CISTER

Research Centre in

Real-Time & Embedded

Computing Systems

Conference Paper

Worst-case Stall Analysis for Multicore Architectures with Two Memory Controllers

Muhammad Ali Awan*

Pedro F. Souto

Konstantinos Bletsas*

Benny Åkesson*

Eduardo Tovar*

*CISTER Research Centre

CISTER-TR-180401

2018/07/03

Worst-case Stall Analysis for Multicore Architectures with Two Memory Controllers

Muhammad Ali Awan*, Pedro F. Souto, Konstantinos Bletsas*, Benny Åkesson*, Eduardo Tovar*

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: muaan@isep.ipp.pt, ksbs@isep.ipp.pt, kbake@isep.ipp.pt, emt@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

In multicore architectures, there is potential for contention between cores when accessing sharedresources, such as system memory. Such contention scenarios are challenging to accurately analyse,from a worst-case timing perspective. One way of making memory contention in multicoresmore amenable to timing analysis is the use of memory regulation mechanisms. It restricts thenumber of accesses performed by any given core over time by using periodically replenished percorebudgets. Typically, this assumes that all cores access memory via a single shared memorycontroller. However, ever-increasing bandwidth requirements have brought about architectureswith multiple memory controllers. These control accesses to different memory regions and arepotentially shared among all cores. While this presents an opportunity to satisfy bandwidthrequirements, existing analysis designed for a single memory controller are no longer safe.This work formulates a worst-case memory stall analysis for a memory-regulated multicorewith two memory controllers. This stall analysis can be integrated into the schedulability analysisof systems under fixed-priority partitioned scheduling. Five heuristics for assigning tasks andmemory budgets to cores in a stall-cognisant manner are also proposed. We experimentallyquantify the cost in terms of extra stall for letting all cores benefit from the memory space offeredby both controllers as well as also evaluate the five heuristics for different system characteristics.

Worst-case Stall Analysis for Multicore Architectures with Two Memory Controllers

Muhammad Ali Awan
CISTER Research Centre and ISEP, Porto, Portugal
muaan@isep.ipp.pt
https://orcid.org/0000-0001-5817-2284

Pedro F. Souto
University of Porto, Faculty of Engineering and CISTER Research Centre, Porto, Portugal
pfs@fe.up.pt
https://orcid.org/0000-0002-0822-3423

Konstantinos Bletsas
CISTER Research Centre and ISEP, Porto, Portugal
ksbs@isep.ipp.pt
https://orcid.org/0000-0002-3640-0239

Benny Akesson
Embedded Systems Innovation, Eindhoven, the Netherlands
benny.akesson@tno.nl
https://orcid.org/0000-0003-2949-2080

Eduardo Tovar
CISTER Research Centre and ISEP, Porto, Portugal
emt@isep.ipp.pt
https://orcid.org/0000-0001-8979-3876

Abstract

In multicore architectures, there is potential for contention between cores when accessing shared resources, such as system memory. Such contention scenarios are challenging to accurately analyse, from a worst-case timing perspective. One way of making memory contention in multicores more amenable to timing analysis is the use of memory regulation mechanisms. It restricts the number of accesses performed by any given core over time by using periodically replenished per-core budgets. Typically, this assumes that all cores access memory via a single shared memory controller. However, ever-increasing bandwidth requirements have brought about architectures with multiple memory controllers. These control accesses to different memory regions and are potentially shared among all cores. While this presents an opportunity to satisfy bandwidth requirements, existing analysis designed for a single memory controller are no longer safe.

This work formulates a worst-case memory stall analysis for a memory-regulated multicore with two memory controllers. This stall analysis can be integrated into the schedulability analysis of systems under fixed-priority partitioned scheduling. Five heuristics for assigning tasks and memory budgets to cores in a stall-cognisant manner are also proposed. We experimentally quantify the cost in terms of extra stall for letting all cores benefit from the memory space offered by both controllers, and also evaluate the five heuristics for different system characteristics.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Computer systems organization → Real-time operating systems, Computer systems organization → Real-time system architecture

Keywords and phrases multiple memory controllers, memory regulation, multicore

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.2

30th Euromicro Conference on Real-Time Systems (ECRTS 2018).
Editor: Sebastian Altmeyer; Article No. 2; pp. 2:1–2:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (CEC/04234).

1 Introduction

The strong trend towards increasing integration in hardware for embedded real-time systems has led to multicores becoming mainstream platforms of choice for such systems. Multicores have significant advantages in terms of computing power, energy usage and weight over single-cores. Yet, one issue with multicores is that worst-case timing analysis becomes more complicated. In particular, the fact that multiple cores contend for the same shared system resources (buses, caches, memory) must be accounted for [8].

Focusing specifically on the problem of main memory contention, we note various research efforts [21, 22, 15, 10, 5, 11, 13, 20, 14, 3] that employ *memory regulation* to make the memory access patterns of the different cores more amenable to worst-case timing analysis. Under memory regulation schemes, each core gets an associated periodically-replenished memory access budget. When a core attempts to issue more memory accesses than its budget, it gets temporarily stalled, until the next replenishment.

However, engineering practice forges ahead and analysis has to catch up. In recent years, in response to memory bandwidth often becoming a performance bottleneck, multicore chips that integrate, not one, but two memory controllers, have become commercially available. In such platforms, both controllers are accessible by all cores, with little to no difference in latency. Examples include various multicore processors from the NXP QorIQ series [16], ranging from the P5020 with 2 cores to the P4080 with 8 cores. For existing approaches to apply to systems with multiple controllers, one could statically map cores to memory controllers and apply the analyses to each partition independently. This simple approach efficiently reduces contention between cores. Still, it may be hard to find a partition such that no tasks depend on data from the memory space of the other memory controller. Core-to-controller partitioning also reduces flexibility in bandwidth allocation, as a partition's bandwidth requirements must be met by just the associated memory controller. In cases when no such partitions can be found, there are currently no good solutions, because existing approaches can be *unsafe* when applied to platforms with two controllers. The reason is that the worst-case memory access pattern for each controller in isolation will not necessarily lead to the worst-case stall, as we demonstrate in Section 5. This reality motivated the present work, whose main contributions are the following:

First, we show via counter-examples that existing techniques for upper-bounding the memory stall, conceived for memory-regulated architectures with a single memory controller, are not necessarily safe in the presence of multiple controllers. Our second and more important contribution is new worst-case memory stall analysis for architectures with two memory controllers, shared by all cores. This analysis, which presumes fixed task-to-core partitioning and fixed-priority scheduling, can then be integrated to the schedulability analysis for the system. Finally, we explore five different stall-cognisant heuristics for combined memory-bandwidth-to-core assignment and task-to-core assignment and evaluate their performance in terms of schedulability via experiments with synthetic task sets capturing different system characteristics. These experiments also highlight the performance implications of having fully shared memory controllers vs. partitioning the controllers to different cores, in cases when the latter arrangement would be viable from the application perspective (i.e., no data sharing across memory domains).

Next, in Section 2, we discuss related work. Section 3 defines our system model and

Section 4 discusses some relevant existing results from the single-controller case. Section 5 contains our analysis. Section 6 describes five proposed stall-cognisant task-to-core assignment heuristics. Section 7 provides an experimental evaluation of our analysis and heuristics in terms of theoretical schedulability using synthetic task sets. Section 8 concludes the paper.

2 Related work

Several software-based approaches for mitigating memory interference in multi-core platforms [21, 22, 15, 10, 5, 11, 3] have been proposed in recent years. These approaches consider a periodic server implemented in software that manages the memory budgets of the cores. This is combined with run-time monitoring through performance counters that keep track of the number of memory accesses and with an enforcement mechanism that suspends tasks whenever they exhaust their budget. Our work is similar to these, as it exploits such a memory throttling mechanism to enforce budgets on memory requests.

The memory regulation techniques used to mitigate the interference on shared memory controllers introduce new stalls and the existing analyses are unsafe unless adapted to account for them. Some efforts in this direction exist for partitioned fixed-priority scheduling [21, 13] and hierarchical scheduling in [5]. Mancuso et al. [13], under their Single-Core Equivalence framework [18], addressed the problem of fixed-priority partitioned schedulability on a multicore. They employ the periodic software-based memory regulation mechanism MemGuard [22] to ensure that each core gets an equal share of memory bandwidth in each regulation interval (or period) and stalls until the end of the regulation period once the budget has been depleted. Such stalls, resulting from the memory regulation together with contention stalls are integrated into the schedulability analysis in [13].

Even if equal sharing of memory bandwidth is simple and facilitates porting applications from a single-core to multi-core platforms (by making the analysis akin to that for a single-core), it is inefficient when the memory requirements of the applications on different cores are diverse. Yao et al. [20], and Pellizzoni and Yun [17] generalise the arrangement along with the analysis to uneven memory budgets per core. The former approach considers round-robin memory arbitration, whereas the latter proposes a new analysis for First-Ready First Come First Served memory scheduling. Recently, Mancuso et al. [14] improved their memory stall analysis by considering the exact memory bandwidth distribution on other cores. However, all these approaches are designed to work with a *single memory controller* and are unsafe with more than one memory controller. The reason is that the worst-case memory access pattern for each controller in isolation no longer necessarily leads to the worst-case stall, as we show in Section 5. In contrast, our work provides a worst-case memory stall analysis for a memory-regulated multicore platform with two memory controllers and incorporates this stall analysis in the schedulability analysis for fixed-priority partitioned preemptive scheduling. We also present five memory bandwidth allocation and task-to-core assignment heuristics.

To summarise, existing works on memory regulation rely on an assumption of a single memory controller. Here, we expand the state-of-the-art by proposing memory stall analysis, when each core can access two controllers, facilitating data sharing among applications and allowing more flexible use of bandwidth. We allow uneven distribution of the memory bandwidth of each controller to available cores. Each core is scheduled under fixed-priority preemptive scheduling, assuming a round-robin memory arbitration policy on both controllers.

3 System Model

We consider a platform with m identical cores (P_1 to P_m) and 2 memory controllers on the same chip, both uniformly accessible by all cores. The sets of memory regions accessible by the two controllers are non-overlapping. Examples of platforms with 2-8 identical cores and two memory controllers include NXP QorIQ P-series P4040, P4080, P5020 and P5040 [16].

Assume a set of n sporadic tasks, τ_1 to τ_n. Each task has a minimum interarrival time T_i, a deadline $D_i \leq T_i$, and a worst-case execution time (WCET) of C_i. Like Yao et al. [20], we assume that CPU computation and memory access do not overlap in time. Each task can access memory via both controllers. Therefore, $C_i = C_i^e + C_i^{m1} + C_i^{m2}$, where C_i^e is the worst-case CPU computation time and C_i^{m1} and C_i^{m2} are the worst-case total memory access times of a task via each respective controller in isolation.

The tasks are partitioned to the cores (no migration) and fixed-priority scheduling is used. For the memory controllers and their interconnects, we assume a round-robin policy [22, 20]. The last-level cache (furthest from the cores) is either private or partitioned to each core. Like Yao et al. [20], we assume that access to main memory is regulated, e.g., by Memguard [22] or in hardware. We also require performance monitoring counters to count the number of memory accesses issued to each controller from each core. As in [20], we assume each memory access takes a constant time L. This allows us to specify P and C_i^e, C_i^{m1} and C_i^{m2} as multiples of L. Our model is agnostic w.r.t. the points in time when memory accesses may occur within the activation of a task and hence imposes no particular programming model.

Memory accesses are regulated as follows. Each core i has a *memory access budget* $Q1_i$ for memory controller 1, which is the maximum allowed memory access time (measured in multiples of L) via that controller, within a *regulation period* of length P. Likewise, it has a budget $Q2_i$ for controller 2. These budgets are set at design time and may be different. A core i that consumes its memory access budget for a given memory controller within a regulation period is *stalled* until the start of the next regulation period[1]. Regulation periods on all cores are synchronised. The *memory bandwidth share* of core i on controller 1 is $b1_i = \frac{Q1_i}{P}$. Similarly for $b2_i$ and controller 2. By design, $\sum_i b1_i \leq 1$ and $\sum_i b2_i \leq 1$, i.e., the bandwidth of any controller is not overcommitted.

4 Relevant existing results from the single-controller case

We now summarise some existing results from [20], for a similar, albeit single-controller, system, in order to later show why those do not apply, and new analysis is needed.

The technique in [20] calculates a worst-case stall term for each task, which is added to the right hand side of the standard worst-case response time (WCRT) recurrence relation for fixed priorities. For ease of presentation, the authors assume that there is a single task running on the core under consideration. Later on, for the case when many tasks are assigned to a core, they explain how to equivalently model the considered task τ_i and all higher-priority tasks as a single synthetic task, in order to apply their stall analysis and derive the worst-case stall term for τ_i. Below, we similarly assume a single task per core.

A memory request may stall either (i) because of requests from other cores, contending for the memory controller simultaneously (a case of **contention stall**) or (ii) because the

---

[1] On practical grounds, we assume that a core is stalled immediately after the Q^{th} memory access in a regulation period via the respective controller is served. Yao et al [20], more generously, assume that it is stalled immediately before attempting a $(Q+1)^{th}$ access within the same regulation period.

issuing core has exhausted its budget for the current regulation period (a **regulation stall**).

Yao et al. identify worst-case patterns for memory accesses and computation within a single regulation period, characterised by maximum stall with the fewest memory accesses. Next, they use these patterns as main "building blocks" for the worst-case pattern for the entire task activation, over multiple regulation periods. In more detail:

Case $b_i \leq 1/m$ (regulation dominant): If $b_i \leq 1/m$, i.e., if the task's bandwidth share is "fair" at most, then a task incurs worst-case stall when all its memory accesses are clustered at the start of its activation, before any computation. Another pessimistic assumption is that the task is released just after a regulation stall, so it waits for $(P - Q_i)$ until the next regulation period. The task will incur a stall of $(P - Q_i)$ within each of the next $\lfloor \frac{C_i^m}{Q} \rfloor$ regulation periods; whether this is entirely due to a regulation stall or partially also due to contention from other cores is irrelevant. Afterwards, any remaining memory accesses (which are too few to trigger a regulation stall), can each incur a worst-case contention stall of (m-1), i.e., one contending access from each other core due to round robin arbitration.

Case $b_i > 1/m$ (contention dominant): In this case, the smallest number of memory accesses per period a core must issue to get the maximum stall is $RBS \stackrel{\text{def}}{=} \frac{P_i - Q_i}{m-1}$, and occurs when the remaining budget is shared evenly among the other cores. From the assumption of the case, $b_i > 1/m$, it follows that $RBS < Q_i$. Therefore, the worst-case pattern for one regulation period involves $c_i^m = RBS$ accesses, each suffering a maximum contention stall of $(m - 1)$, for a total stall of $P - Q_i$. This leaves $Q_i - RBS$ time units not filled by memory accesses or respective stalls. These are filled with computation; if memory accesses were added instead, they would incur no stall. To bound the stall for the entire task activation, this pattern is applied to as many regulation periods as possible. Two subcases exist: either memory accesses or computation will run out first.

Due to space constraints, we refer to [20] for details. Meanwhile, some insights driving Yao's analysis, for single-controller systems, are codified via the following lemmas from [20]:

▶ **Lemma 1.** *Considering the stall of a core due to memory regulation alone, the worst-case memory access pattern of one task is when all accesses within the task are clustered, and the stall is upper bounded by $P - Q_i$ for each regulation period P.*

▶ **Lemma 2.** *If the memory is not overloaded and the regulation periods are the same and synchronized, the stall due to inter-core memory contention alone on each core i with assigned budget Q_i is upper-bounded by $P - Q_i$ for every regulation period P.*

▶ **Lemma 3.** *Considering the contention stall alone, the maximum stall for core i with budget Q_i is obtained when the remaining budget $P - Q_i$ is evenly distributed among all other cores and they generate the maximum amount of accesses.*

5 Analysis

In this section, we formulate the main contribution of this paper: a stall analysis for multicores with two memory controllers, which leverages on Yao et al [20] stall and schedulability analysis for multicores with a single memory controller. First, we look at Lemmas 1 to 3 and Yao's analysis in general, and examine what holds over from [20] and what does not. For readability, we omit the core (task) index, since it is implied. Table 1 summarizes the symbols used.

5.1 What holds over from Yao's analysis and what does not

When we have multiple controllers, with an assigned memory budget Qj for each, Lemma 1 can be generalized as follows:

Table 1 Symbols used in the analysis

$Q1, Q2$	memory budget on controllers 1 and 2, respectively
C^{m1}, C^{m2}	maximum number of memory accesses via controllers 1 and 2, respectively
C^e	worst-case computation time
P	regulation period
m	number of cores
$b1, b2$	core memory bandwidth shared on controllers 1 and 2, respectively
$RBS1, RBS2$	remaining budget share on controllers 1 and 2, respectively
c^{m1*}, c^{m2*}	worst-case number of accesses per period in contention-dominant case
$K1^*$	number of regulation periods of phase 1 in contention-dominant case
$\hat{C}^e, \hat{C}^{m1}, \hat{C}^{m2}$	task computation parameters after phase 1 (in contention-dominant case)
$\Delta\rho^*$	worst-case reduction in regulation stalls w.r.t. maximum regulation stalls in the third case (regulation is dominant only for one controller)
ΔC^e	additional "computation" added to contention-only phase by reducing the number of regulation stalls by 1
ΔC_c^{m2*}	additional number of contention stalls **required** when moving ΔC^e to ensure that the total stall is larger with one less regulation stall on controller 1
$\Delta C_c^{m2}(max)$	maximum number of additional contention stalls obtained by moving ΔC^e to the contention-only phase
$\Delta C_c^{m2}(min)$	minimum number of additional contention stalls obtained by moving ΔC^e to the contention-only phase
$r^m = \frac{C^{m2}}{C^{m1}}$	ratio of memory accesses to each controller
$C_{\bar{c}}^{m2}$	number of memory accesses via controller 2 **without** contention
$single()$	worst-case single controller stall according to Yao's analysis, ignoring the regulation stall at the beginning of the execution

▶ **Lemma 4.** *Considering the stall of a core due to memory regulation alone* ***on controller*** j***, with budget*** Qj*, the worst-case memory access pattern of one task is when all accesses* ***via controller*** j *within the task are clustered, and the stall is upper bounded by* $P - Qj$ *for each regulation period* P*.*

A corollary of this lemma is that the regulation stall on controller j is maximum when there are no memory accesses to a second controller in that period. Note also that a core can only regulation-stall on at most one memory controller in a given regulation period.

With multiple controllers Lemmas 2 and 3 apply to *each controller separately.* Furthermore, because a core may access memory via multiple controllers in a single regulation period, a consequence of Lemma 2 is the following:

▶ **Lemma 5.** *If the memory is not overloaded and the regulation periods are the same and synchronized, the stall due to inter-core memory contention alone on each core* i *with assigned budget* Qj_i *on controller* j *is upper-bounded by* $\min\left(\sum_j (P - Qj_i), \frac{P}{m} \cdot (m-1)\right)$ *for every regulation period* P*.*

When there are multiple memory controllers, the maximum contention stall may occur when there are accesses via more than one controller. The first argument to the min operator in the above expression sums up the contention stall from each controller according to Lemma 2. The second argument expresses the fact that no more than P/m accesses (irrespective via which controller) can all suffer the worst-case per-access contention stall of $(m-1)$ because of round robin arbitration. Both terms independently bound the contention stall.

When there are multiple shared controllers and we try to upper-bound the stall over multiple regulation periods, Yao's analysis may not be safe, i.e., it may underestimate the worst-case stall, as illustrated by the example of Figure 1. Execution i) has the worst-case stall, according to Yao's stall analysis, when in a regulation period all memory accesses are

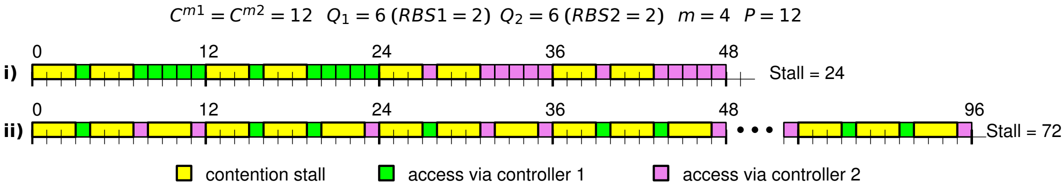

Figure 1 As shown in this example, the worst-case total stall is when there are memory accesses via more than one controller in the same regulation period.

via the same controller. In each period, the first two memory accesses suffer the maximum stall. However the remaining 4 memory accesses suffer no stall, because the maximum stall in every regulation period is 6, $P - Qi$, and it occurs in the first two memory accesses of the respective regulation period. Execution ii) shows the worst-case stall when there are accesses via both controllers in the same period. In each period, we have 2 memory accesses *via each controller* and each of these accesses suffers the maximum contention stall, $m - 1$. This is because the contention stall on accesses via one controller does not affect the contention stall on accesses via the other controller. Thus, in execution ii) all memory accesses suffer the maximum contention stall, whereas in execution i) only a third does.

5.2 Two-controller Task Stall Analysis

Having shown the need for a new analysis, we consider several cases depending on the values of $b1$ and $b2$. Some entail sub-cases. More specifically, we consider 3 cases:

1. $b1 \leq \frac{1}{m} \wedge b2 \leq \frac{1}{m}$
2. $b1 > \frac{1}{m} \wedge b2 > \frac{1}{m}$
3. remaining cases, i.e. $(b1 \leq \frac{1}{m} \wedge b2 > \frac{1}{m}) \vee (b1 > \frac{1}{m} \wedge b2 \leq \frac{1}{m})$

5.2.1 Case 1: $b1 \leq \frac{1}{m} \wedge b2 \leq \frac{1}{m}$

In this case, for each controller, the worst case occurs when there is a regulation stall, as shown in [20]. By Lemma 4, the following execution suffers the worst-case stall. In a first phase, there is the longest sequence of consecutive periods with regulation stalls on controller 1, followed by a second phase consisting of the longest sequence of consecutive periods with regulation stalls on controller 2. Finally, there is a third phase with the remaining memory accesses via each controller, $C^{mi} \bmod Qi$, that suffer the maximum contention stall per memory access, $m - 1$, and any computation. Because in each of the two first phases all memory accesses are via a single controller, we can use Yao's stall analysis to compute an upper bound on the stall in each of these phases. The upper bound of the total stall can then be computed by adding the upper bounds for each phase. I.e.:

$$\begin{aligned} Stall =& single(C^m = \left\lfloor \frac{C^{m1}}{Q1} \right\rfloor \cdot Q1, C^e = 0, Q = Q1, P = P, m = m) \\ &+ single(C^m = \left\lfloor \frac{C^{m2}}{Q2} \right\rfloor \cdot Q2, C^e = 0, Q = Q2, P = P, m = m) \\ &+ (C^{m1} \bmod Q1 + C^{m2} \bmod Q2) \cdot (m - 1) \end{aligned} \tag{1}$$

where $single()$ is the stall based on Yao's (single controller) stall analysis for the respective set of parameter values [20].

5.2.2 Case 2: $b1 > \frac{1}{m} \wedge b2 > \frac{1}{m}$

In this case, according to Yao's analysis, for each controller, the worst case occurs when there is maximum contention stall in a regulation period with the minimum number of memory accesses. However, as shown in Figure 1, in this case the worst-case stall may occur when a task accesses memory via different controllers in the same regulation period. Therefore, the worst-case memory access pattern of a task in this case has 3 phases, as illustrated in Figure 2 i):

Phase 1 In this phase, every regulation period incurs the maximum contention stall. This phase terminates when the task runs out of memory accesses via some controller, and therefore cannot sustain the maximum contention stall any more. In Figure 2 i), this phase spans the two first periods, and, in each period, there are $RBS1$ and $RBS2$ memory accesses via the respective controller.

Phase 3 In this phase, all accesses are via a single controller. This phase may not exist, if the task runs out of memory accesses via both controllers in the same regulation period. In Figure 2 i), this is the 4th and last period and has memory accesses only via controller 1.

Phase 2 This "middle" phase may also not exist, but if it exists, it has only one regulation period. In this phase, we have memory accesses via both controllers, but either there are not enough memory accesses via at least one of the controllers to ensure the maximum contention stall in that period, or there is not enough execution to fill the complete period. In Figure 2 i), this is the 3rd period, and has only one memory access via controller 2.

According to Lemma 5, there are two main cases for the maximum contention stall in a regulation period. We analyse each of these cases separately.

5.2.2.1 Sub-case 1: $(P - Q1) + (P - Q2) < \frac{P}{m} \cdot (m - 1)$

In this case, the maximum contention stall in a regulation period occurs when a task performs $RBS1$ memory accesses via controller 1 and $RBS2$ memory accesses via controller 2. Therefore, the maximum stall per period is $(RBS1+RBS2)\cdot(m-1) = (P-Q1)+(P-Q2)$. Because the task is non preemptive and $(P - Q1) + (P - Q2) < \frac{P}{m} \cdot (m - 1)$, by the definition of the sub-case, there is a "hole" of size $P - (RBS1 + RBS2) \cdot m$ that must be filled with execution, i.e. either computation or memory accesses. An execution in which computation fills as many of these holes as possible suffers the maximum stall, because any additional memory accesses in these periods suffer no contention stall. This will minimize the number of memory accesses without contention in Phase 1, increasing the number of memory accesses in latter phases, and possibly their stall. Similar reasoning can be applied to Phase 2, as well.

Figure 2 illustrates an execution pattern that leads to the worst-case stall, based on the above observations. In execution i) there is enough computation to fill in the holes in Phases 1 (the first two periods) and 2. However, there is not enough computation to ensure that all memory accesses suffer contention: in the 4th and last period, which belongs to Phase 3, there are 4 memory accesses via controller 1 that do not suffer any contention. In execution ii) there is not enough computation to fill the holes in Phase 2, and therefore, we have 6 memory accesses via controller 1 in Phase 2, the 3rd period, that do not suffer any contention, and there is no 3rd Phase. In execution iii) there is no Phase 2, because all memory accesses via controller 2 are used to fill the holes in Phase 1. Phase 3 consists only of a single memory access via controller 1. Finally, in execution iv) there is not enough computation, and Phase 1, like Phase 2, has only one period, and there is no Phase 3.

It can be shown, by case analysis, that in any of these executions swapping any computation or memory access in one regulation period with computation or memory accesses

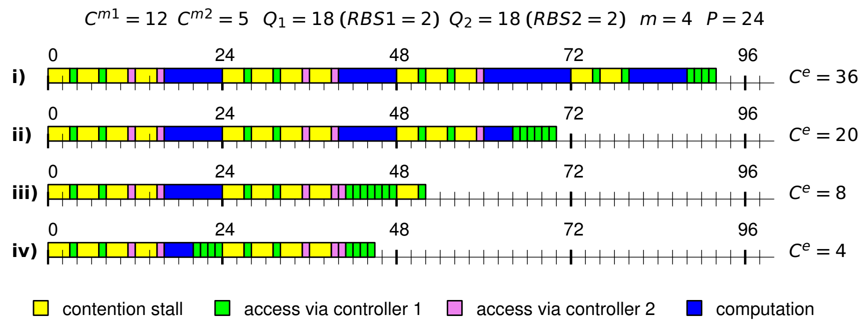

Figure 2 Example execution patterns with worst case stall, for the contention-dominant case when $(P - Q1) + (P - Q2) < \frac{P}{m} \cdot (m - 1)$

in later regulation periods does not lead to an increase in the total stall, and therefore the execution pattern shown suffers the maximum stall. The following stall analysis is based on the execution pattern shown in Figure 2.

In order to reuse the analysis in other cases below, let c^{m1*} and c^{m2*} be the minimum values of c^{m1} and c^{m2}, respectively, that maximize the contention stall in a regulation period, assuming that any holes are filled with computation. Note that by Lemma 5, it must be $c^{m1*} \leq RBS1$ and $c^{m2*} \leq RBS2$. In this sub-case, they are $RBS1$ and $RBS2$, respectively.

In our analysis, we consider Phase 1 separately from the remaining phases, if any.

Phase 1 stall: In Phase 1, the contention stall in every regulation period is maximum and equal to $(c^{m1*} + c^{m2*}) \cdot (m - 1)$. The total stall in this phase is:

$$Stall1 = K1^* \cdot (c^{m1^*} + c^{m2^*}) \cdot (m - 1) \tag{2}$$

$$\text{where: } K1^* = min\left(\left\lfloor \frac{C^{m1}}{c^{m1*}} \right\rfloor, \left\lfloor \frac{C^{m2}}{c^{m2*}} \right\rfloor, \left\lfloor \frac{C^e + C^{m1} + C^{m2}}{P - (c^{m1*} + c^{m2*}) \cdot (m - 1)} \right\rfloor\right) \tag{3}$$

is the number of regulation periods in Phase 1. Indeed, to sustain maximum contention stall in every regulation period of Phase 1, the task must have both:

1. Enough memory accesses via controller 1, i.e. $K1^* \leq \left\lfloor \frac{C^{m1}}{c^{m1*}} \right\rfloor$.
2. Enough memory accesses via controller 2, i.e. $K1^* \leq \left\lfloor \frac{C^{m2}}{c^{m2*}} \right\rfloor$.
3. Enough execution, since when a core is not stalled it must be either computing or accessing memory, i.e. in every Phase 1 period a task must execute for $P - (c^{m1*} + c^{m2*}) \cdot (m - 1)$. Therefore, $K1^* \leq \left\lfloor \frac{C^e + C^{m1} + C^{m2}}{P - (c^{m1*} + c^{m2*}) \cdot (m-1)} \right\rfloor$.

We use the minimum of these 3 values, because this is the largest possible number of periods in Phase 1 and, as argued above, this leads to the worst-case stall.

Remaining stall: Without loss of generality, let $\left\lfloor \frac{C^{m1}}{c^{m1*}} \right\rfloor \geq \left\lfloor \frac{C^{m2}}{c^{m2*}} \right\rfloor$, i.e. controller 2 runs out of memory accesses entirely in Phase 2 the latest. (The other case is symmetric.)

To analyse the stall in Phases 2 and 3, if any, we consider the stall of each controller separately. Since memory accesses via controller 2 occur only in Phase 2 (which has at most one regulation period) and not in Phase 3, the contention stall on controller 2 can be upper bounded by $min(\hat{C}^{m2}, RBS2) \cdot (m - 1)$, where $\hat{C}^{m2}$ is the number of memory accesses via controller 2 in Phase 2, if any. Observe that these memory accesses and respective stall can be taken into account as computation in the analysis of the stall of memory accesses via controller 1, in Phase 2. Furthermore, in Phase 3, if any, all memory accesses are via controller 1, only. Therefore, we apply Yao's stall analysis to compute the stall of memory accesses via controller 1 in Phases 2 and 3, if they exist.

So, to complete analysis of this case, we compute $\hat{C}^{m2}$, as well as parameters for Yao's single controller stall analysis. Since in the latter we consider the remaining memory accesses via controller 2, $\hat{C}^{m2}$, and respective stall, if any, as computation, C^e is obtained by adding to that value the remaining computation, $\hat{C}^e$, i.e. the task computation that was not performed in Phase 1. Finally, the value of C^m to use in the single controller analysis is the number of memory accesses via controller 1 that were not performed in Phase 1, $\hat{C}^{m1}$, if any. Thus,

$$\begin{aligned} Stall =& Stall1 + min(\hat{C}^{m2}, RBS2) \cdot (m-1) \\ & + single(C^e = \hat{C}^{m2} + min(\hat{C}^{m2}, RBS2) \cdot (m-1) + \hat{C}^e, \\ & C^m = \hat{C}^{m1}, Q = Q1, P = P, m = m) \end{aligned} \tag{4}$$

where $Stall1$ is given by (2). Next, we derive the expressions for $\hat{C}^e, \hat{C}^{m1}$ and $\hat{C}^{m2}$.

In every Phase 1 period a task must execute, i.e. either compute or access memory, when it is not stalled. Thus, in addition to the $c^{m1*} + c^{m2*}$ memory accesses that lead to the maximum stall in a regulation period, a task may have to execute for the remaining time: $P - (c^{m1*} + c^{m2*}) \cdot m$. As we have argued, the total stall will be maximum in executions where computation fills as many of these "holes" as possible. Thus:

$$\hat{C}^e = max\left(0, C^e - K1^* \cdot \left(P - (c^{m1*} + c^{m2*}) \cdot m\right)\right) \tag{5}$$

If there is enough computation to fill all these holes, $C^e \geq K1^* \cdot \left(P - (c^{m1*} + c^{m2*}) \cdot m\right)$, then $\hat{C}^{m1} = C^{m1} - K1^* \cdot c^{m1*}$ and $\hat{C}^{m2} = C^{m2} - K1^* \cdot c^{m2*}$.

If there is not enough computation to fill all these holes, then the remaining holes, $K1^* \cdot (P - (c^{m1*} + c^{m2*}) \cdot m) - C^e$, will be filled with memory accesses. Thus, the total number of memory accesses that will occur in the remaining phases, if any, is:

$$\begin{aligned} \hat{C}^m &= C^{m1} + C^{m2} - K1^* \cdot (c^{m1*} + c^{m2}) - (K1^* \cdot (P - (c^{m1*} + c^{m2*}) \cdot m) - C^e) \\ &= C^{m1} + C^{m2} - (K1^* \cdot (P - (c^{m1*} + c^{m2*}) \cdot (m-1)) - C^e) \end{aligned} \tag{6}$$

To determine $\hat{C}^{m1}$ and $\hat{C}^{m2}$, we distinguish two cases, depending on the value of $K1^*$.

If $K1^* = \left\lfloor \frac{C^{m2}}{c^{m2*}} \right\rfloor \left(\leq \left\lfloor \frac{C^{m1}}{c^{m1*}} \right\rfloor\right)$, then an execution that has at least $min(C^{m1} - K1^* \cdot c^{m1*}, RBS1, \hat{C}^m)$ controller 1 memory accesses in the first period of the remaining phases, will suffer maximum stall, because all these memory accesses suffer maximum contention stall. The first bound is the number of memory accesses not used to ensure maximum stall in Phase 1, the second bound is the maximum number of accesses via controller 1 that can suffer maximum stall in a regulation period, and the third bound is the number of memory accesses in the remaining phases. This ensures that controller 2 runs out of memory accesses before controller 1, as shown in Figure 2 iii). Thus the number of memory accesses via controller 2 in Phase 2 is $\hat{C}^{m2} = min\left(\hat{C}^m - min(C^{m1} - K1^* \cdot c^{m1*}, RBS1, \hat{C}^m), C^{m2} - K1^* \cdot c^{m2*}\right)$ i.e. the number of memory accesses via controller 2 in Phase 2 is the number of memory accesses not used to fill the holes in Phase 1, discounted by the minimum number of memory accesses via controller 1 that suffer maximum contention in Phase 2, and upper-bounded by the maximum number of controller 2 memory accesses that are not necessary to ensure maximum stall in Phase 1. Finally, $\hat{C}^{m1} = \hat{C}^m - \hat{C}^{m2}$.

If $K1^* = \left\lfloor \frac{C^e + C^{m1} + C^{m2}}{P - (c^{m1*} + c^{m2*}) \cdot (m-1)} \right\rfloor$, there is not enough execution to complete the $K1^*$+1st regulation period, if any – the execution has at most one regulation period after Phase 1.

In this case, the total stall is maximum in executions where the number of contention stalls in the last period is maximum. However, there cannot be more than $RBS1$ ($RBS2$)

contention stalls on controller 1 (2, respectively) in this period. Like in the previous sub-case, an execution with at least $min(C^{m1} - K1^* \cdot c^{m1*}, RBS1, \hat{C}^m)$ controller 1 memory accesses in Phase 2, guarantees that controller 2 runs out of memory accesses no later than controller 1, and suffers maximum stall, because all these memory accesses suffer maximum contention stall. Thus, the expressions we derived for $\hat{C}^{m1}$ and $\hat{C}^{m2}$ in the previous sub-case, are also valid for this one. Summarizing, we get the following expressions:

$$\hat{C}^{m2} = \begin{cases} C^{m2} - K1^* \cdot c^{m2*}, & \text{if } C^e \geq K1^* \cdot \big(P - (c^{m1*} + c^{m2*}) \cdot m\big) \\ min(\hat{C}^m - min(C^{m1} - K1^* \cdot c^{m1*}, RBS1, \hat{C}^m), C^{m2} - K1^* \cdot c^{m2*}), & \text{o.w} \end{cases} \tag{7}$$

$$\hat{C}^{m1} = \begin{cases} C^{m1} - K1^* \cdot c^{m1*} & \text{if } C^e \geq K1^* \cdot \big(P - (c^{m1*} + c^{m2*}) \cdot m\big) \\ \hat{C}^m - \hat{C}^{m2} & \text{otherwise} \end{cases} \tag{8}$$

5.2.2.2 Sub-case 2: $(P - Q1) + (P - Q2) \geq \frac{P}{m} \cdot (m - 1)$

In this case (by the definition of RBS), $RBS1 + RBS2 \geq \frac{P}{m}$, and therefore it is possible to guarantee maximum contention stall in a period, without any computation or memory accesses without contention. To ensure the maximum stall, the memory accesses should be distributed in a "balanced" way so that both controllers run out of memory access at more or less the same time, thus ensuring that all C^m memory access suffers the maximum stall.

Let c^{m1*} and c^{m2*} be the number of memory accesses via controllers 1 and 2 per regulation period that maximize the contention stall in a period. The goal is then to ensure:

$$\frac{C^{m1}}{c^{m1*}} = \frac{C^{m2}}{c^{m2*}} \Rightarrow c^{m2*} = \frac{C^{m2}}{C^{m1}} \cdot c^{m1*} \Rightarrow c^{m2*} = r^m c^{m1*}, \text{where: } r^m \stackrel{\text{def}}{=} \frac{C^{m2}}{C^{m1}} \tag{9}$$

Without loss of generality, assume $r^m < 1$; the other case is symmetrical. Then it must be:

$$c^{m1*} + c^{m2*} = \frac{P}{m} \Rightarrow (1 + r^m) \cdot c^{m1*} = \frac{P}{m} \Rightarrow c^{m1*} = \frac{P}{m \cdot (1 + r^m)} \tag{10}$$

$$c^{m2*} = r^m \cdot c^{m1*} \Rightarrow c^{m2*} = r^m \cdot \frac{P}{m \cdot (1 + r^m)} \tag{11}$$

We now consider three sub-cases:

Sub-case $c^{m1*} \leq RBS1 \wedge c^{m2*} \geq 1$: In this case it is possible to ensure that all memory accesses suffer the maximum contention stall, even without any computation. Thus:

$$Stall = (C^{m1} + C^{m2}) \cdot (m - 1) \tag{12}$$

Note that even though c^{m1*} or c^{m2*} may be fractional, these are average values. This means that in an execution with worst-case stall, the number of memory accesses via any controller may not be the same across all the regulation periods. However, there is an execution such that $c^{m1} + c^{m2} = \frac{P}{m}$, in all but possibly the last regulation period, and $c^{m1} \leq RBS1$ and $c^{m2} \leq RBS2$ in every regulation period.

Sub-case $c^{m1*} > RBS1$: In this case, both controllers would run out of computation at the same time only if the number of memory accesses via controller 1 exceeded $RBS1$, and therefore there would be memory accesses without any contention. An execution following the pattern illustrated in Figure 2, with $c^{m1*} = RBS1$ and $c^{m2*} = min(\frac{P}{m} - RBS1, RBS2)$ will have the worst-case stall, and therefore we can apply the analysis in Section 5.2.2.1.

Sub-case $c^{m2*} < 1$: In this case, both controllers would run out of computation at the same time only if there are some periods without memory accesses via controller 2. An execution following the pattern illustrated in Figure 2, with $c^{m2*} = 1$ and $c^{m1*} = min(\frac{P}{m} - 1, RBS1)$ will have the worst-case stall, and therefore we can apply the analysis in Section 5.2.2.1.

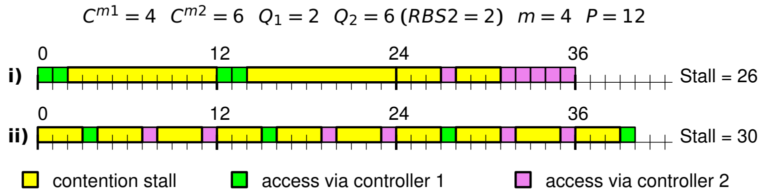

Figure 3 Maximizing the number of regulation stalls may not lead to the worst-case stall.

5.2.3 Case 3: $(b1 \leq \frac{1}{m} \wedge b2 > \frac{1}{m}) \vee (b1 > \frac{1}{m} \wedge b2 \leq \frac{1}{m})$

In this case, executions with the maximum number of regulation stalls do not always lead to the worst-case stall. This is shown in Figure 3. In execution i), all memory accesses via controller 1 are clustered, causing two regulation stalls on controller 1, in the first two regulation periods. All the memory accesses via controller 2, occur in the third regulation period. Of these, only the first two suffer the maximum contention stall. The remainder suffer no contention, because the memory budget of the remaining cores, $P - Qi$, is exhausted by the stalls of the first 2 memory accesses. In execution ii), there is one memory access via controller 1 in each period, and thus there is no regulation stall on controller 1, but each of these accesses suffers the maximum contention stall. Furthermore, in each of the first 3 periods, there are 2 memory accesses via controller 2, each of which suffers the maximum contention stall. Thus all memory accesses via both controllers suffer the maximum contention stall, and the total stall for execution ii) exceeds that of execution i). This is counter-intuitive, because the contention stall by accesses via controller 1 in execution ii), 12, is smaller than the regulation stall, 20, caused by the same number of accesses via controller 1 in execution i). However, this loss is more than compensated by the contention stall in execution ii) of the 4 memory accesses via controller 2 that suffer no contention stall in execution i). I.e., although we are trading off a regulation stall, $P - Qi$, for contention stalls, presumably with maximum contention stall, $Qi \cdot (m - 1) < P - Qi$, we may also be adding stall to memory accesses via the second controller that previously suffered no stall.

Depending on whether $b1 \leq \frac{1}{m} \wedge b2 > \frac{1}{m}$ or $b1 > \frac{1}{m} \wedge b2 \leq \frac{1}{m}$, there are two sub-cases. Because they are symmetrical, we analyse only the former.

5.2.3.1 Sub-case 3.1: $b1 \leq \frac{1}{m} \wedge b2 > \frac{1}{m}$

Figure 3 shows that the maximum number of regulation stalls does not always lead to the worst-case stall. Furthermore, it can be shown that the total stall is maximum if there are no memory accesses via the second controller in periods with a regulation stall. Thus, the following memory access pattern with two phases leads to the worst-case stall: in the first phase, there is a number, possibly 0, of consecutive periods with regulation stalls; in the second phase, the contention-only phase, there is a number of consecutive periods, possibly only 1, with contention stalls only. Thus, the problem of finding the worst-case stall reduces to that of determining the number of regulation stalls that maximizes that stall. Actually, to simplify the mathematical expressions, we use the difference, $\Delta\rho^*$, between this number and the maximum number of regulation stalls, $\left\lfloor \frac{C^{m1}}{Q1} \right\rfloor$. The total stall can then be determined

Algorithm 1 Compute stall for each task.

Input: Parameters: C^{m1}, C^{m2}, m, C^e, $Q1$, $Q2$ and P (omitting task's index for simplicity)
Output: Stall

$b1 = \frac{Q1}{P}$, $b2 = \frac{Q2}{P}$, $RBS1 = \frac{P-Q1}{m-1}$, $RBS2 = \frac{P-Q2}{m-1}$ and $C = C^e + C^{m1} + C^{m2}$
if $(b1 \le \frac{1}{m} \;\wedge\; b2 \le \frac{1}{m})$ **then** ▷ Regulation stall is dominant for both controllers
  Stall = Equation (1)
else if $(b1 > \frac{1}{m} \;\wedge\; b2 > \frac{1}{m})$ **then** ▷ Contention stall is dominant for both controllers
  if $((P - Q1) + (P - Q2) < \frac{P}{m} \cdot (m-1))$ **then**
    $c^{m1*} = RBS1$, $c^{m2*} = RBS2$
    Compute Stall with Algorithm 2
  else ▷ $(P - Q1) + (P - Q2) \ge \frac{P}{m} \cdot (m-1)$
    $r^m = \frac{C^{m2}}{C^{m1}}$, c^{m1*} = Equation 10, c^{m2*} = Equation 11
    if $(r^m < 1)$ **then**
      if $(c^{m1*} \le RBS1 \wedge c^{m2*} \ge 1)$ **then**
        Stall = Equation 12
      else if $(c^{m1*} > RBS1)$ **then**
        $c^{m1*} = RBS1$, $c^{m2*} = min(RBS2, \frac{P}{m} - RBS1)$
        Compute Stall with Algorithm 2
      else ▷ $c^{m2*} < 1$
        $c^{m1*} = min(RBS1, \frac{P}{m} - 1)$ $c^{m2*} = 1$
        Compute Stall with Algorithm 2
      end if
    else ▷ $r^m \ge 1$: symmetric of previous case, swap indices
    end if
  end if
else ▷ Regulation stall is dominant for only one controller
  if $(b1 \le \frac{1}{m} \;\wedge\; b2 > \frac{1}{m})$ **then**
    Compute $\Delta\rho^*$ using Algorithm 3
    Stall = Equation 13
  else ▷ $b2 \le \frac{1}{m} \;\wedge\; b1 > \frac{1}{m}$: symmetric of previous case
  end if
end if
return Stall $+= (P - \min(Q1, Q2))$ ▷ This adds the stall when the task arrives.

Algorithm 2 Compute stall for contention dominant case.

Input: Parameters: c^{m1*}, c^{m2*}, C^{m1}, C^{m2}, m, C^e, $Q1$, $Q2$ and P (omitting task's index)
Output: Stall

$b1 = \frac{Q1}{P}$, $b2 = \frac{Q2}{P}$, $RBS1 = \frac{P-Q1}{m-1}$, $RBS2 = \frac{P-Q2}{m-1}$ and $C = C^e + C^{m1} + C^{m2}$
$K1^*$ = Equation 3 ,
Stall1 = Equation 2
$\hat{C}^e$ = Equation 5, $\hat{C}^{m1}$ = Equation 8, $\hat{C}^{m2}$ = Equation 7
Stall23 = $single(C^e = \hat{C}^{m2} \cdot m + \hat{C}^e, C^m = \hat{C}^{m1}, Q = Q1, P = P, m = m)$
return Stall = Stall1 + $min(\hat{C}^{m2}, RBS2) \cdot (m-1)$ + Stall23 ▷ Equation 41

using Yao's stall analysis:

$$
\begin{aligned}
Stall = {} & single(Q = Q1, C^m = C^{m1} - C^{m1} \bmod Q1 - \Delta\rho^* Q1, C^e = 0) \\
& + ((C^{m1} \bmod Q1) + \Delta\rho^* \cdot Q1) \cdot (m-1) \\
& \quad + single(Q = Q2, C^m = C^{m2}, C^e = C^e + ((C^{m1} \bmod Q1) + \Delta\rho^* \cdot Q1) \cdot m) \qquad (13)
\end{aligned}
$$

where, for computing the stall on the memory accesses via controller 2 in the second phase, we account the memory accesses via controller 1 in the second phase and respective contention

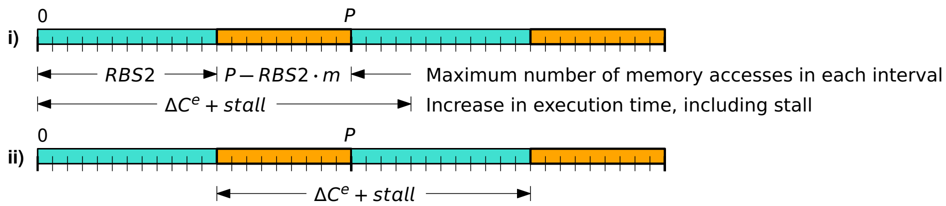

Figure 4 Upper (i) and lower (ii) bounds on ΔC_c^{m2}.

stalls as computation, assuming that each of them suffers the maximum contention stall under round-robin, $m-1$. Algorithms 1 an 2 detail the case analysis that we have described so far in this section. In the following, we determine the value of $\Delta\rho^*$.

We consider two main sub-cases depending on whether there is enough computation, including residual memory accesses via controller 1, to ensure that every memory access via controller 2 suffers maximum contention.

5.2.3.2 Sub-case 1: Enough computation

If $C^e \geq \left\lfloor \frac{C^{m2}}{RBS2} \right\rfloor \cdot (P - m \cdot RBS2) - (C^{m1} \bmod Q1) \cdot m$, then every memory access in the contention-only phase suffers maximum contention, and therefore the total stall is maximum when the number of regulation stalls is maximum, i.e. $\Delta\rho^* = 0$.

5.2.3.3 Sub-case 2: Not enough computation

In this case, as illustrated in Figure 3, if there are memory accesses in the contention-only phase that suffer no contention, the worst-case stall may occur when the number of regulation stalls is not maximum.

When the number of regulation stalls is decremented by one, the regulation stall reduction by $P - Q1$ is partially compensated by an increase of the contention stall via controller 1 by $Q1 \cdot (m-1)$. If the increase in contention stall via controller 2, $\Delta stall_c^2$ is such that:

$$\Delta stall_c^2 > \Delta stall_c^{2^*} \stackrel{\text{def}}{=} P - Q_1 - Q_1 \cdot (m-1) = P - Q_1 \cdot m \tag{14}$$

then reducing the number of regulation stall leads to a larger total stall. In other words, the total stall will be worse if the increase in the number of memory accesses with maximum stall, ΔC_c^{m2}, satisfies the following inequality:

$$\Delta C_c^{m2} > \Delta C_c^{m2^*} \stackrel{\text{def}}{=} \frac{\Delta stall_c^{2^*}}{m-1} = \frac{P - Q_1 \cdot m}{m-1} \tag{15}$$

Like in the analysis in Section 5.2.2, to compute the stall on memory accesses via controller 2, we can view the memory accesses via controller 1 and respective contention stall as computation. Thus, we need to determine ΔC_c^{m2} when the computation in the contention-only phase increases by $\Delta C^e = Q_1 \cdot m$. The challenge is that this value, ΔC_c^{m2}, may not be constant. I.e., when we increase the computation by $\Delta C^e = Q_1 \cdot m$, ΔC_c^{m2} may have different values depending on other parameter values.

Our solution is to compute the maximum and minimum values of ΔC_c^{m2}, $\Delta C_c^{m2}(max)$ and $\Delta C_c^{m2}(min)$, respectively, and then finding $\Delta\rho^*$ by case analysis, as described below.

When we increase the computation of the contention-only phase by ΔC^e, the total execution of that phase, including any contention, will increase at least by that much. This execution can replace memory accesses via controller 2 that did not have any contention, i.e memory accesses in excess of $RBS2$ accesses per period, which can then be shifted towards the end of the execution. ΔC_c^{m2} will be maximum if the shifted memory accesses are added to a regulation period with no memory accesses via controller 2, up to a limit of $RBS2$ memory accesses per regulation period, as shown in Figure 4 i). Thus, in this case, as a result of adding ΔC^e memory accesses we get:

$$\begin{aligned}\Delta C_c^{m2}(max) &= RBS2 \cdot \left\lfloor \frac{\Delta C^e}{RBS2 + P - RBS2 \cdot m} \right\rfloor \\ &\quad + min(RBS2, \Delta C^e \bmod (RBS2 + P - RBS2 \cdot m)) \\ &= RBS2 \cdot \left\lfloor \frac{\Delta C^e}{Q2} \right\rfloor + min(RBS2, \Delta C^e \bmod Q2) \end{aligned} \tag{16}$$

The first term corresponds to the number of additional periods with $RBS2$ memory accesses. (Note that ΔC^e is used both to shift memory accesses via controller 2, and to fill the "hole" in the remaining of the period, $P - RBS2 \cdot m$.) The second term corresponds to the number of memory accesses in the last incomplete regulation period, if any: essentially, the memory accesses that can be replaced with the remaining of ΔC^e that was not used for the additional full periods, upper-bounded by $RBS2$.

On the other hand, ΔC_c^{m2} will be minimum, if, before adding ΔC^e, the execution ended immediately after the $RBS2$ accesses with contention. This is shown in Figure 4 ii). In this case, the analysis is similar to the one above, and therefore we can also use (16), except that rather than using ΔC^e, we need to use $max(\Delta C^e - (P - RBS2 \cdot m), 0)$, because the remainder of the period at which the execution ended needs to be filled with "computation" before an earlier memory access via controller 2 without contention stall can experience the maximum contention stall by shifting it towards the end of the execution.

We can now distinguish there sub-cases, depending on the relative values of ΔC_c^{m2*}, $\Delta C_c^{m2}(max)$ and $\Delta C_c^{m2}(min)$.

Sub-case $\Delta C_c^{m2*} \geq C_c^{m2}(max)$: In this case, the increase in the number of memory accesses with contention cannot make up for the eliminated regulation stall, so $\Delta\rho^* = 0$.

Sub-case $\Delta C_c^{m2*} < C_c^{m2}(min)$: In this case, the increase in the number of memory accesses with contention suffices to make up for the eliminated regulation stall. Therefore, the worst-case stall increases as we reduce the number of regulation stalls until one of the following 3 cases occurs: 1) there are no more regulation stalls; 2) there are not enough memory accesses via controller 2, ΔC_c^{m2*}, without the maximum contention stall, to compensate for the loss in the regulation stall; or 3) the number of memory accesses via controller 1 in at least one period of the second phase exceeds $Q1 - 1$, in which case we would have a regulation stall, and therefore there would be no reduction in the number of regulation stalls.

Because ΔC^{m2} varies, we do not know a closed form expression for the number of regulation periods to reduce. Thus, we use the iterative procedure shown in Algorithm 3. We hence start with $\Delta\rho^* = 0$ and keep increasing it by one until one of the above 3 stop conditions is satisfied. Specifically, while there are still enough memory accesses via controller 2 without maximum contention stall, $C_{\bar{c}}^{m2}$, and there is still one regulation stall (line 15), $\Delta\rho^*$ is tentatively increased by one. In each iteration, we tentatively compute the total stall using Yao's analysis with the appropriate parameters (line 18) and the number of memory accesses via controller 2 that suffer no contention (line 20), for the tentative value of $\Delta\rho^*$. If the number of memory accesses via controller 1 in all periods of the contention-only phase (line 21) does not exceed $Q1 - 1$, then the tentative values become definitive (line 22), and the algorithm loops again, otherwise it exits the loop and terminates.

All other cases, i.e. $C_c^{m2}(min) \leq \Delta C_c^{m2*} < C_c^{m2}(max)$: In this case, the total stall

Algorithm 3 Compute $\Delta\rho^*$

Input: Parameters: C^{m1}, C^{m2}, m, C^e, $Q1$, $Q2$ and P (omitting task index for simplicity)
Output: $\Delta\rho^*$

1: $RBS1 = \frac{P-Q1}{m-1}$, $RBS2 = \frac{P-Q2}{m-1}$ and $C = C^e + C^{m1} + C^{m2}$
2: $\Delta C^e = m \cdot Q1$
3: $\Delta C_c^{m2}(max) =$ Equation 16
4: $\Delta C_c^{m2}(min) =$ Equation 16, but replacing ΔC^e with $\max(\Delta C^e - (P - m \cdot RBS2), 0)$
5: $\Delta C_c^{m2*} = \left\lfloor \frac{P-m \cdot Q1}{m-1} \right\rfloor$
6: **if** $(C^e \geq \left\lfloor \frac{C^{m2}}{RBS2} \right\rfloor \cdot (P - m \cdot RBS2) - (C^{m1} \bmod Q1) \cdot m)$ **then**
7: $\quad \Delta\rho^* = 0$ ▷ There is enough "computation"
8: **else if** ($\Delta C_c^{m2}(max) \leq \Delta C_c^{m2*}$) **then** ▷ Which implies $\Delta C_c^{m2}(min) \leq \Delta C_c^{m2*}$
9: $\quad \Delta\rho^* = 0$ ▷ Maximize regulation stalls on Controller one
10: **else if** ($\Delta C_c^{m2}(min) > \Delta C_c^{m2*}$) **then** ▷ Which implies $\Delta C_c^{m2}(max) > \Delta C_c^{m2*}$
11: $\quad \Delta\rho^* = 0$
12: $\quad stall = single(Q = Q2, C^m = C^{m2}, C^e = C^e + (C^{m1} \bmod Q1) \cdot m)$
13: $\quad R = C^{m2} + C^e + (C^{m1} \bmod Q1) \cdot m + stall$
14: $\quad C_{\bar{c}}^{m2} = C^{m2} - \left\lfloor \frac{R}{P} \right\rfloor \cdot RBS2 - min\left(\left\lfloor \frac{R \bmod P}{m} \right\rfloor, RBS2\right)$
15: $\quad$ **while** $\left(C_{\bar{c}}^{m2} > \Delta C_c^{m2*} \wedge \Delta\rho^* < \left\lfloor \frac{C^{m1}}{Q1} \right\rfloor \right)$ **do**
16: $\quad\quad \Delta\rho_t^* = \Delta\rho^* + 1$
17: $\quad\quad \hat{C}^{m1} = C^{m1} \bmod Q1 + \Delta\rho_t^* \cdot Q1$ ▷ Accesses via controller 1 in second phase
18: $\quad\quad stall = single(Q = Q2, C^m = C^{m2}, C^e = C^e + \hat{C}^{m1} \cdot m)$
19: $\quad\quad R = C^{m2} + C^e + \hat{C}^{m1} \cdot m + stall$
20: $\quad\quad C_{\bar{c}t}^{m2} = max\left(C^{m2} - \left\lfloor \frac{R}{P} \right\rfloor \cdot RBS2 - min\left(\left\lfloor \frac{R \bmod P}{m} \right\rfloor, RBS2\right), 0\right)$
21: $\quad\quad$ **if** $\left(\hat{C}^{m1} - min\left(Q1 - 1, max\left(0, \left\lfloor \frac{R \bmod P}{m} \right\rfloor - RBS2\right)\right) \leq (Q1 - 1) \cdot \frac{R}{P}\right)$ **then** ▷ Enough reg. periods to ensure that there is no reg. stall in periods with accesses via both controllers.
22: $\quad\quad\quad \Delta\rho^* = \Delta\rho_t^*, C_{\bar{c}}^{m2} = C_{\bar{c}t}^{m2}$
23: $\quad\quad$ **else** break
24: $\quad\quad$ **end if**
25: $\quad$ **end while**
26: **else** ▷ $\Delta n_c^2(min) \leq \Delta n_c^{2*} < \Delta n_c^2(max)$
27: $\quad \Delta\rho(max) = 0, stall(max) = 0$ ▷ Variables for maximum stall
28: $\quad$ **for** $\Delta\rho^* = 0$ **to** $\left\lfloor \frac{C^{m1}}{Q1} \right\rfloor$ **do** ▷ Do exhaustive search
29: $\quad\quad \hat{C}^{m1} = C^{m1} \bmod Q1 + \Delta\rho_t^* \cdot Q1$
30: $\quad\quad stall = single(Q = Q2, C^m = C^{m2}, C^e = C^e + \hat{C}^{m1} \cdot m)$ ▷ Cont. stall on both controllers
31: $\quad\quad R = C^{m2} + C^e + \hat{C}^{m1} \cdot m + stall$ ▷ Duration of contention-only phase
32: $\quad\quad$ **if** $stall + \left(\left\lfloor \frac{C^{m1}}{Q1} \right\rfloor - \Delta\rho^*\right) \cdot (P - Q1) > \text{stall(max)}$ $\wedge \left(\hat{C}^{m1} - min\left(Q1 - 1, max\left(0, \left\lfloor \frac{R \bmod P}{m} \right\rfloor - RBS2\right)\right) \leq (Q1 - 1) \cdot \left\lfloor \frac{R}{P} \right\rfloor\right)$ **then**
33: $\quad\quad\quad stall(max) = stall + \left(\left\lfloor \frac{C^{m1}}{Q1} \right\rfloor - \Delta\rho^*\right) \cdot (P - Q1)$
34: $\quad\quad\quad \Delta\rho^*(max) = \Delta\rho^*$
35: $\quad\quad$ **end if**
36: $\quad$ **end for**
37: $\quad \Delta\rho^* = \Delta\rho^*(max)$
38: **end if**
39: **return** $\Delta\rho^*$

sometimes increases when the number of regulations stalls decreases by one and sometimes it does not. Thus in this case, our approach to find the value of $\Delta\rho^*$ is to compute the stall for every possible value of $\Delta\rho^*$ and pick the one that leads to the maximum stall. Algorithm 3, lines 27-37, details the computation of $\Delta\rho^*$ in this case.

Algorithm 4 Sensitivity analysis to reclaim memory bandwidth from both controllers

Input: $b1$, $b2$, m, Δ (threshold to stop the algorithm)) and τ
Output: Minimum required memory bandwidth of both controllers

$b^1_{min} = 0$, $b^1_{max} = b1$, $b^2_{min} = 0$, $b^2_{max} = b2$
while $(b^1_{max} - b^1_{min} > \Delta \vee b^2_{max} - b^2_{min} > \Delta)$ **do**
  for each controller $j \in \{1, 2\}$ **do**
    if $(b^j_{max} - b^j_{min} > \Delta)$ **then**
      $X^j = \lfloor \frac{b^j_{min} + b^j_{max}}{2} \rfloor$
      if $(j == 1)$ **then**
        Schedulability = MultiControllerSchedulabilityAnalysis(X^j, b^2_{max}, m, τ)
      else
        Schedulability = MultiControllerSchedulabilityAnalysis(b^1_{max}, X^j, m, τ)
      end if
      if (Schedulability == true) **then** $b^j_{max} = X^j$
      else $b^j_{min} = X^j$
      end if
    end if
  end for
end while
return $\{b^1_{max}$ and $b^2_{max}\}$

5.3 Schedulability analysis

Until now, we assumed one task per core. When many tasks are assigned to a core, the task in consideration and those of higher priority can be modelled by one synthetic task, using the approach in [20], and schedulability analysis can be performed as summarized in Section 4.

6 Bandwidth Allocation and Task-to-core Assignment Heuristics

We propose 5 heuristics for allocating tasks and memory bandwidth of both controllers to the cores. They are evaluated in terms of system schedulability. We use Audsley's algorithm [1] to assign task priorities, even if it is no longer necessarily optimal in the presence of stalls.

Even: The total memory bandwidth of both controllers is equally distributed among all cores. Subject to this even share, the task-to-core assignment is performed using first-fit.

Uneven: Initially, this heuristic also distributes both controller's bandwidth evenly among cores and employs the first-fit for task-to-core assignment. However, instead of declaring failure whenever a task does not fit on any core, it sets that task aside, and moves on to consider the next task. Any tasks that remain unassigned after considering all tasks, are handled in-order as follows. Each core's memory bandwidth from both controllers is "trimmed" to the minimum value that preserves schedulability, via the sensitivity analysis of Algorithm 4, explained later in this section. Let the total reclaimed bandwidth from all cores be $B1$ and $B2$ from controllers 1 and 2, respectively. A second round of first-fit tries to assign the remaining tasks, assuming that the bandwidth of the target core i is increased by $B1$ and $B2$ for controllers 1 and 2, respectively. Upon successfully assigning such a task, we trim anew the target cores's memory budgets via sensitivity analysis, adjust the available reclaimed budgets and move on to the next task in a similar manner.

Greedy-fit: Initially, the total memory bandwidth of both controllers is assigned to the first core and the task-set is iterated over once (in a given order) to assign the maximum number of tasks to this core; if a task does not fit, we try the next one. Afterwards, the spare bandwidth from each controllers on this core is reclaimed via sensitivity analysis, and is fully assigned to next core. And so on, until all tasks are assigned or the cores run out.

Humble-fit: Similar to greedy-fit, except that when a task assignment fails, we move to the next core (attempting no more task assignments on the current one).

Memory-fit: Initially, $b1_i = b2_i = 0$, for every core i, where bx_i is the allocated memory bandwidth of controller x on core i. Each task is assigned to the core i that requires the least increase to $b1_i + b2_i$ to accommodate it, subject to existing task assignments.

"Uneven" explores a larger solution space than "Even. "Greedy-fit" and "Humble-fit" aggressively optimise for processing capacity use foremost. Conversely, "Memory-fit" optimises for bandwidth instead. Hence, all heuristics sample the solution space in different ways.

Sensitivity analysis: Algorithm 4 presents the sensitivity analysis that trims the unused memory bandwidth from both controllers and outputs the least required memory bandwidth from each controller. This sensitivity analysis, used for bandwidth optimisation, is an adaptation of binary interval search ([19, 2]). It gives both controllers an equal chance to preserve their bandwidth in a round-robin fashion. By comparison, completely optimizing one controller followed by the second one, may lead to an imbalanced approach, hence avoided.

7 Evaluation

Experimental Setup We developed a Java tool for our experiments. Its first module generates the synthetic task sets and sets up a platform with the given input parameters. A second module performs task-to-core allocation and feasibility analysis with two controllers.

We generate the task-set with a given target $U = x \cdot m : x \in (0, 1]$ using UUnifast-discard algorithm [6, 9] for unbiased distribution of task utilisations. The task-set size is given as input. Task periods are log-uniform-distributed, in the range 10-100 ms. We assume implicit deadlines, even if our analysis also holds for constrained deadlines. The WCET of a task is derived as $C_i = U_i \cdot T_i$. The total memory accesses of each task are randomly selected in the range $[0, \Gamma \cdot C_i]$, with memory intensity factor $\Gamma \in (0, 1]$ user-defined. The total memory accesses are randomly divided between the two memory controllers. By default the task-set is sorted in descending order of utilisation. For each set of input parameters, we generate 1000 task-sets. We use independent pseudo-random number generators for the utilisations, minimum inter-arrival times/deadlines, memory accesses and reuse their seeds [12]. Table 2 summarises all parameters, with default values underlined. We observed that size of the regulation period has no effect on the schedulability ratio.

To avoid having hundreds of plots, in each experiment we vary only one parameter, with others conforming to the defaults from Table 2 and present the results as plots of *weighted schedulability*. This performance metric, adopted from [4], condenses what would have been three-dimensional plots into two dimensions. It is a weighted average that gives more weight to task-sets with higher utilisation, which are supposedly harder to schedule. Specifically, using notation from [7], let $S_y(\tau, p)$ represent the result (0 or 1) of the schedulability test y for a given task-set τ with an input parameter p. Then $W_y(p)$, the weighted schedulability for that test y as a function p, is $W_y(p) = \sum_{\forall \tau} \left(\bar{U}(\tau) \cdot S_y(\tau, p)\right) / \sum_{\forall \tau} \bar{U}(\tau)$. Here, $\bar{U}(\tau) \stackrel{\text{def}}{=} U(\tau)/m$ is the system utilisation, normalised by the number of cores m.

No other stall analysis with two controllers exists in the literature to compare with. We therefore compare our approach against a system where the two controllers are partitioned among cores that can only make requests to their assigned controller. The benefit of such partitioning is that it roughly cuts contention in half. On the other hand, tasks assigned to one controller cannot access data addressable by the other controller.

For the comparison, half the cores are assigned to each controller. Since each core

Table 2 Overview of Parameters

Parameters	Values	Default
Number of cores (m)	$\{4, 8, 12, 16\}$	4
Task-set size (n)	$\{8, 16, 24, 32, 40, 48\}$	16
Regulation period (P)	$\{1us, 10us, 100us, 1ms\}$	$100us$
Inter-arrival time (T_i)	$10ms$ to $100ms$	N/A
Nominal utilisation ($\bar{U} = \frac{U}{m}$)	$\{0.1 : 0.01 : 1\}$	N/A
Memory intensity (Γ)	$\{0.1 : 0.1 : 1\}$	0.5

accesses only one controller, the feasibility of the tasks assigned to it can be tested with Yao's analysis [20]. We adapt the task-to-core assignment heuristics and bandwidth allocation schemes presented in Section 6 for the partitioned case: The even heuristic equally divides a controller's bandwidth among its associated cores. Similarly, in the uneven heuristic, the readjustment of the controllers bandwidth is performed only among the controller's associated cores. In the greedy-fit/humble-fit, all bandwidth of a given controller is only assigned to its first associated core with an objective to maximise the number of tasks assigned to it. The trimmed-off bandwidth from this controller is assigned to its remaining associated cores. If the task is not feasible on the cores associated to the first controller, its feasibility is next checked on the set of cores associated with the second controller. In the memory-fit, a task is assigned to the core with the lowest bandwidth requirement of its controller. We use Yao- and MC- prefixes to denote the partitioned and our approach, respectively, followed by the name of the heuristic (even, uneven, greedy-fit, humble-fit and memory-fit).

Results Figure 5 presents the weighted schedulability for different number of cores for both systems with partitioned and shared controllers (our approach) using the proposed heuristics. The first important result is that all heuristics under partitioning perform better than their corresponding heuristic under shared controllers, which is due to the stall being roughly cut in half in the former approach. This difference ranges around $10\% - 30\%$ in absolute terms of weighted schedulability. Of course, this expected result applies only when there are no dependencies across partitions. However, in many systems, there is always some sharing/communication of data among tasks and this might make such partitioning impossible. In other cases, a single controller cannot deliver enough bandwidth. This may become more frequent in the future, as applications getting more demanding. Therefore safe analysis for predictable access to both controllers, like the one proposed here, is needed.

In terms of heuristics, memory-fit, uneven, even, humble-fit and greedy-fit is the descending ordered list w.r.t. weighted schedulability ratio. The memory-fit heuristic, which optimises the use of memory bandwidth, performing best, implies that memory bandwidth is typically the scarce resource for the given set of parameters. The uneven and even heuristics are more balanced in terms of bandwidth and processing capacity distribution and hence, perform close to memory-fit. Humble-fit and greedy-fit are too aggressive in construction to optimise the use of processing capacity at the cost of memory resources and hence underperform the other heuristics in a memory-scarce setup. Greedy-fit manages the memory resources comparatively better than humble-fit and hence, outperforms it. Yet, if the applications are compute-intensive and the system is not scarce w.r.t. memory resource, the heuristics that optimise for processing resources may become handy and outperform their counterparts.

With more cores, the contention from other cores increases and hence, the schedulability of the system decreases. Figure 6 presents the effect of memory intensity over the proposed

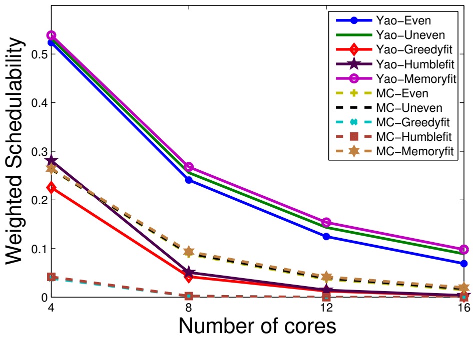

Figure 5

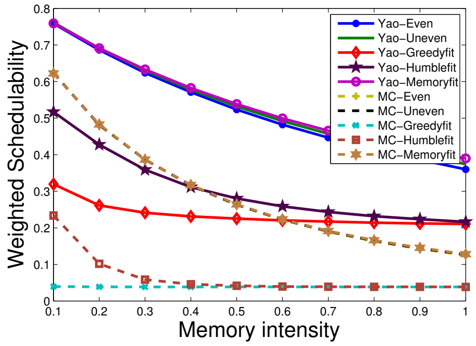

Figure 6

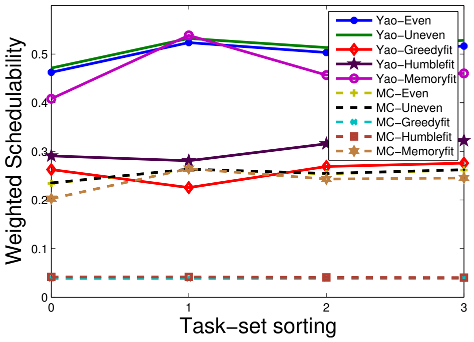

Figure 7

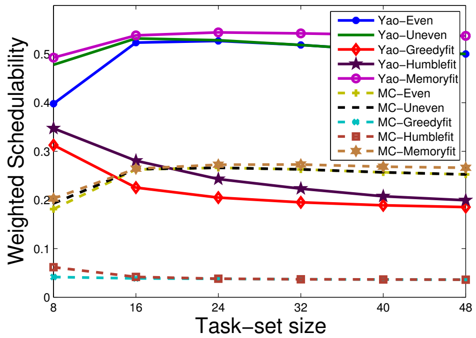

Figure 8

heuristics. Obviously, higher memory intensity increases the contention on the shared controllers, consequently decreasing the schedulability. We also compared the effect of the task indexing over the different heuristics as shown in Figure 7. The numbers 0, 1, 2 and 3 on the X-axis correspond to task-set ordering w.r.t. descending order of deadlines, utilisation, total memory requests and memory density (i.e. total memory requests divided by the T_i), respectively. Task-set indexing w.r.t. utilisation benefits the memory-fit, even and uneven heuristics. Figure 8 shows that task-set size has very limited effect on the memory-fit, uneven and even approaches and they scale well when that increases. Conversely, the performance of humble-fit and greedy-fit degrade with greater task-set sizes due to their aggressive optimisation of processor usage at the expense of memory bandwidth.

8 Conclusion

This paper demonstrated that worst-case memory stall analyses for single-memory-controller multicores with memory regulation are unsafe if applied to multicores with multiple memory controllers. We overcome this limitation by proposing a new memory stall analysis for multicore platforms with two memory controllers that captures the cases where all cores can access both controllers. We also proposed five memory allocation heuristics, each specialising in optimising processing capacity and/or memory bandwidth. The experimentally quantified cost of allowing all cores to flexibly access the memory space of two controllers is $10 - 30\%$ in terms of weighted schedulability. Results further show that the proposed memory-fit heuristic performs well if bandwidth is scarce. The even and uneven heuristics are suitable for balanced systems, while greedy-fit and humble-fit are handy for compute-intensive systems.

References

1. N. C. Audsley. On priority asignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.
2. M. A. Awan, K. Bletsas, P. F. Souto, and E. Tovar. Semi-partitioned mixed-criticality scheduling. In *Proceedings of the 30th International Conference on the Architecture of Computing Systems (ARCS 2017)*, pages 205–218, 2017. `doi:10.1007/978-3-319-54999-6_16`.
3. Muhammad Ali Awan, Pedro Souto, Konstantinos Bletsas, Benny Akesson, and Eduardo Tovar. Mixed-criticality scheduling with memory bandwidth regulation. In *Proceedings of the 55th IEEE/ACM Conference on Design Automation and Test in Europe (DATE 2018)*, March 2018.
4. A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of the OSPERT*, pages 33–44, 2010.
5. M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory-bus contention. *ACM SIGBED Review*, 10(3):35–42, 2013. `doi:10.1145/2544350.2544354`.
6. E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Journal of Real–Time Systems*, 30(1-2):129–154, May 2005. `doi:10.1007/s11241-005-0507-9`.
7. A. Burns and R. I. Davis. Adaptive mixed criticality scheduling with deferred preemption. In *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS 2014)*, pages 21–30, Dec 2014. `doi:10.1109/RTSS.2014.12`.
8. D. Dasari, B. Akesson, V. Nélis, M. A. Awan, and S. M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, pages 39–48, June 2013.
9. R. I. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 398–409, Dec 2009. `doi:10.1109/RTSS.2009.31`.
10. J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 151–159, June 2014. `doi:10.1109/SIES.2014.6871199`.
11. Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, and Mikael Sjodin. Multi-core composability in the face of memory-bus contention. In *Proceedings of the 20th IEEE Real-Time Technology and Applications Symposium (RTAS 2014)*, 2014.
12. Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.
13. R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, pages 174–183, July 2015. `doi:10.1109/ECRTS.2015.23`.
14. Renato Mancuso, Rodolfo Pellizzoni, Neriman Tokcan, and Marco Caccamo. WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:23, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
15. J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In

Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014), pages 109–118, 2014. doi:10.1109/ECRTS.2014.20.

16 NXP. QorIQ Layerscape Processors Based on Arm Technology, 2018. www.nxp.com/products/processors-and-microcontrollers/applications-processors/qoriq-platforms/p-series.

17 R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*, pages 97–108, April 2016. doi:10.1109/RTAS.2016.7461339.

18 Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, Bradford Richard, et al. Single core equivalent virtual machines for hard real—time computing on multicore processors. Technical report, Univ. of Illinois at Urbana Champaign, 2014.

19 Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, Pedro Souto, and Benny Åkesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. *Journal of Real–Time Systems*, 50(5-6):680–735, 2014.

20 G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha. Schedulability analysis for memory bandwidth regulated multicore real-time systems. *IEEE Transactions on Computers*, 65(2):601–614, Feb 2016.

21 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 299–308, July 2012. doi:10.1109/ECRTS.2012.32.

22 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, pages 55–64, April 2013. doi:10.1109/RTAS.2013.6531079.