Schedulability Analysis for Multi-core Systems Accounting for Resource Stress and Sensitivity

Robert I. Davis
Department of Computer Science, University of York, UK.

David Griffin
Department of Computer Science, University of York, UK.

Iain Bate
Department of Computer Science, University of York, UK.

Abstract

Timing verification of multi-core systems is complicated by contention for shared hardware resources between co-running tasks on different cores. This paper introduces the Multi-core Resource Stress and Sensitivity (MRSS) task model that characterizes how much stress each task places on resources and how much it is sensitive to such resource stress. This model facilitates a separation of concerns, thus retaining the advantages of the traditional two-step approach to timing verification (i.e. timing analysis followed by schedulability analysis). Response time analysis is derived for the MRSS task model, providing efficient context-dependent and context independent schedulability tests for both fixed priority preemptive and fixed priority non-preemptive scheduling. Dominance relations are derived between the tests, and proofs of optimal priority assignment provided. The MRSS task model is underpinned by a proof-of-concept industrial case study.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

Keywords and phrases real-time, multi-core, scheduling, schedulability analysis, cross-core contention, resource stress, resource sensitivity

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2021.7

Funding Innovate UK HICLASS project (113213) and the ESPRC grants STRATA (EP/N023641/1) and MARCH (EP/V006029/1). EPSRC Research Data Management: No new primary data was created during this study.

Acknowledgements The authors would like to thank Rolls-Royce PLC for providing the object code for one of their aero-engine controllers for use in real-time systems research.

1 Introduction

1.1 Background

The survey published by Akesson et al. in 2020 [1], shows that about 80% of industry practitioners developing real-time systems are using multi-core processors, about twice the number that are using single-cores. On a single-core processor, when a task executes without interruption or pre-emption it has exclusive access to the hardware resources that it needs. The execution time of the task therefore depends *only* on its own behavior and the initial state of the hardware. This is in marked contrast to what happens when a task executes on one core of a multi-core processor. Multi-core processors are typically designed to provide high average-case performance at low cost, with hardware resources shared between cores. These shared hardware resources typically include, the interconnect, caches, and main memory, as well as other platform specific components. As a consequence, the execution time of a task running on one core of a multi-core system can be extended by *interference* due to contention for shared hardware resources emanating from co-running tasks on the other cores.

This problem of cross-core contention and interference has led to timing verification of multi-core systems becoming a hot topic of real-time systems research in the decade to 2020. The survey published by Maiza et al. in 2019 [32] classifies approximately 120 research papers in this area. Much of this research relies on detailed information about shared hardware resources and the policies used to arbitrate access to them. This information is then used to derive analytical bounds on the maximum interference possible due to contending tasks running on the other cores. In practice, however, there can be substantial difficulties in obtaining and using such detailed low-level information, since it is not typically disclosed by hardware vendors. This is because the complex resource arbitration policies and low-level hardware design features employed comprise valuable intellectual property. Further, even if such information is available, then the overall behavior can be so complex as to preclude a static analysis that provides meaningful bounds, as opposed to substantial overestimates.

The predominant industry practice is to use measurement-based timing analysis techniques to estimate worst-case execution times[1] (WCETs). However, the simple extension of measurement-based techniques to multi-core systems cannot provide an adequate solution that bounds the impact of cross-core interference. This is because cross-core interference is highly dependent on the timing of accesses to shared hardware resources by both the task under analysis and its co-runners. In practice, it is not possible to choose the worst-case combination of behavior (inputs, paths, and timing) for co-running tasks that will result in the maximum interference occurring [33]. A potential solution to this problem, which is being taken up commercially [37], is to employ a more nuanced measurement-based approach using *micro-benchmarks* [36, 16, 33, 24]. These micro-benchmarks sustain a high level of resource accesses, ameliorating the timing alignment issues inherent in the naive approach discussed above. Micro-benchmarks can be used to characterize tasks in terms of the interference that they can cause, or be subject to, due to contention over a particular hardware resource.

The timing verification of single-core systems has traditionally been solved via a *two-step* approach [32]. First context-independent WCET estimates are obtained, either via static or measurement-based timing analysis. Second, these estimates are used as parameter values in a task model, with schedulability analysis employed to determine if all of the tasks can meet their timing constraints when executed under a specific scheduling policy. This separation of concerns between timing analysis and schedulability analysis brings many benefits; however, its effectiveness is greatly diminished in multi-core systems due to the fact that execution times heavily depend on co-runner behavior and the cross-core interference that they bring. Inflating individual task execution time estimates to account for the maximum amount of context-independent interference that could potentially occur during the time interval in which each task executes can result in gross over-estimates that are not viable in practice [27]. Rather, research [2, 10] has shown that it is more effective to consider contention over the longer time frame of task response times.

1.2 Contribution and Organization

In Section 2, we introduce the *Multi-core Resource Stress and Sensitivity* (MRSS) task model that characterizes how much each task *stresses* shared hardware resources and how much each task is *sensitive* to such resource stress. The MRSS task model provides a simple interface and a separation of concerns between timing analysis and schedulability analysis, thus retaining

[1] About 66% of the industry practitioners surveyed by Akesson et al. [1] used some form of measurement-based timing analysis, whereas only about 33% used some form of static timing analysis.

the advantages of the traditional two-step approach to overall timing verification. The MRSS task model relies on timing analysis, either measurement-based or static, to provide task parameter values characterizing stand-alone (i.e. no contention) WCETs, resource stresses, and resource sensitivities. Thus, it provides the information needed by schedulability analysis to integrate cross-core interference into the computation of bounds on task response times, and hence determine the schedulability of tasks running on multi-core systems. The MRSS task model is generic and versatile. It supports different types of interference that occur via cross-core contention for shared hardware resources, as follows:

(i) *Limited interference* where contention for the resource is ameliorated by parallelism in the hardware. Here, the interference is *sub-additive*, i.e. less than the time that the co-running task on another core spends accessing the resource.

(ii) *Direct interference* where the bandwidth of the resource is shared between contending cores, for example with Round-Robin bus. Here, the interference is *additive*, directly matching the time that co-running tasks spend accessing the resource.

(iii) *Indirect interference* where contention causes additional interference, over and above the bandwidth consumed by co-running tasks (i.e. a *super-additive* effect), due to changes in the state of the resource that cause further delays to subsequent accesses. An example of indirect interference occurs with main memory (DRAM) [22] when interleaved accesses target different rows, resulting in additional row close and row open operations, increasing memory access latency.

The MRSS task model is not however a panacea, it cannot support *unbounded interference* where task execution is disproportionately impacted by contending accesses. This includes cases where contenders can effectively lock a resource for an extended or unbounded amount of time, or change the information stored in the resource in such a way that it needs to be obtained from elsewhere. Problems of cache thrashing [36], cache coherence [17], and cache miss status handling registers [41] can all cause effectively unbounded interference, and need to be eliminated from systems aimed at providing real-time predictability.

Section 3 introduces schedulability analysis for the MRSS task model, considering task sets scheduled according to partitioned fixed priority preemptive scheduling (pFPPS) and partitioned fixed priority non-preemptive scheduling (pFPNS) policies[2]. Two types of schedulability test are derived: (i) *context-dependent* tests that make use of information about the co-running tasks on the other cores, and (ii) *context-independent* tests that use only information about the tasks running on the same core. The latter are less precise, but *fully composable*, meaning that if the tasks on one core are changed, then only those tasks need have their schedulability re-assessed; task schedulability on the other cores is unaffected. *Composability* is an important issue for industry, particularly when different companies or departments are responsible for the sub-systems running on different cores.

In systems that use fixed priority scheduling, appropriate priority assignment is a crucial aspect of achieving a schedulable system [14]. Section 4 investigates optimal priority assignment, proving that Deadline Monotonic [31] priority ordering is optimal for both the context-independent and the simpler context-dependent schedulability tests for pFPPS. Similarly, Audsley's optimal priority assignment algorithm [4] is proven to be applicable and optimal for the equivalent tests for pFPNS. The more complex and precise context-dependent tests are proven incompatible with Audsley's algorithm [4].

Section 5, provides a systematic evaluation of the effectiveness of the schedulability tests derived in Section 3. The results of this evaluation follow the dominance relationships

[2] The most commonly used real-time scheduling polices in industry practice [1]

demonstrated earlier, indicating the superiority of the more complex context-dependent schedulability tests, while also highlighting the additional contention that adding further cores brings. Section 6 concludes with a summary and directions for future work.

The appendix presents the findings from a case study examining 24 tasks from a Rolls-Royce aero-engine control system. These tasks were assessed using measurement-based timing analysis to obtain broad-brush estimates of their stand-alone WCETs, as well as characterizing their resource stress and resource sensitivity parameters. The purpose of the case study was not to try to determine definitive values for these parameters, in itself a challenging research problem, rather our aim was to obtain proof-of-concept data to act as an exemplar underpinning the MRSS task model and its analysis.

1.3 Related Work

Prior publications that relate to the research presented in this paper include work on micro-benchmarks [36, 16, 33, 24, 37] that can be used to stress resources in multi-core systems, and work on the integration of interference effects into schedulability analysis. Many of the latter papers are summarized in Section 4 of the survey [32] by Maiza et al. Unlike the analysis presented in this paper, which uses a generic task model that is applicable to many different types of interference and a variety of different shared hardware resources, most of these prior works focus on the details of one or more specific hardware resources. They require detailed information about the resource arbitration policy used, the number of resource accesses made by each task, and in some cases the timing of those accesses. By contrast, this paper takes a more abstract, but nonetheless valid view, that interference can be modeled in terms of its execution time impact via resource sensitivity and resource stress parameters for each task. This approach requires less detail about the resource behavior, and is more amenable to practical use, since it can still be used when full details of shared resource behavior are not available from the hardware vendor.

Early work on the integration of interference effects into schedulability analysis by Schliecker et al. [39] used arrival curves to model the resource accesses of each task, and hence how resource access delays due to contention impact upon task response times. Schliecker's work focused on contention over the memory bus. Further work in this area by Schranzhofer et al. [40], Pellizzoni et al. [35], Giannopoulou et al. [19], and Lampka et al. [28] used the superblock model that divides each task into a sequence of blocks, and uses information about the number of resource accesses within different phases of these blocks.

Dasari et al. [9] used a request function to model the maximum number of resource accesses from each task in a given time interval, and integrated this request function into response time analysis. Kim et al. [26] and Yun et al. [42] provided a detailed analysis of contention caused by DRAM accesses, accounting for access scheduling and variations in latencies due to differing states e.g. open and closed rows. The delays due to contention were then integrated into response time analysis. Altmeyer et al. [2, 10] introduced a multi-core response time analysis framework, aimed at combining the demands that tasks place on difference types of resources (e.g. CPU, memory bus, and DRAM) with the resource supply provided by those hardware resources. The resulting explicit interference was then integrated directly into response time analysis. Rihani et al. [38] built on this framework, using it to analyze complex bus arbitration policies on a many-core processor. Huang et al. [23] and Cheng et al. [8] leveraged the symmetry between processing and resource access, viewing tasks as executing and then suspending execution while accessing a shared resource. Using this suspension model in the schedulability analysis, they obtained results that were broadly comparable to those of Altmeyer et al. [2].

Paolieri et al. [34] proposed using a WCET-matrix and WCET-sensitivity values to characterize the variation in task execution times in different execution environments (e.g. with different numbers of contending cores, and different cache partition sizes). This information was then used to determine efficient task partitioning and task allocation strategies. Andersson et al. [3] presented a schedulability test where tasks have different execution times dependent on their co-runners. Here, tasks are represented by a sequence of segments, each of which has execution requirements and co-runner slowdown factors with respect to sets of other segments that could execute in parallel with it. The schedulability test involves solving a linear program to bound the longest response time given the possible ways in which different segments could execute in parallel and the slowdown in execution that this would entail. The method has significant scalability issues that effectively limit the total number of tasks it can handle to approximately 32 tasks on a 4 core system (i.e. 8 tasks per core).

1.4 Inspiration

The research presented in this paper was inspired by the desire to combine a practical approach to characterizing contention via micro-benchmarks and measurement-based techniques with a generic form of schedulability analysis that can be applied to a wide range of homogeneous multi-core systems with different types of shared hardware resources. The aim being to provide an effective form of timing verification that, while retaining the traditional two-step approach, is able to avoid undue pessimism by accounting for interference over long time intervals equating to task response times rather than short time intervals equating to task execution times. With industry practice in mind, the schedulability analysis derived includes context-dependent (non-composable), context-independent (fully composable), and partially composable schedulability tests. The overall method enables task timing behavior on multi-cores to be assessed without necessitating recourse to detailed information about the hardware behavior, something that most chip vendors do not make publicly available.

2 System Model and Assumptions

We assume a multi-core system with m homogeneous cores that executes tasks under either partitioned fixed priority preemptive (pFPPS) or partitioned fixed priority non-preemptive (pFPNS) scheduling. With partitioning, tasks are assigned to a specific core and do not migrate. The tasks are assumed to be independent, but may access a set of shared hardware resources $r \in H$, thus causing interference on the execution of tasks on other cores via cross-core contention. We omit from consideration the effects of resource contention between tasks on the same core, since they are executed sequentially rather than in parallel. We assume that appropriate techniques are used to avoid substantial preemption effects when preemptive scheduling is employed, for example cache partitioning can be used to eliminate cache-related preemption delays. The costs of scheduling decisions and any context switching are assumed to be subsumed into the task execution times. Each task τ_i is characterised by: the minimum inter-arrival time or period between releases of its jobs, T_i, its relative deadline, D_i, and its WCET, C_i, when executing stand-alone, i.e. with no co-runners. All task deadlines are assumed to be *constrained* i.e. $D_i \le T_i$.

Further aspects of the model are based on the concept of *resource sensitive contenders* and *resource stressing contenders*. A resource stressing contender maximizes the stress on a resource r by repeatedly making accesses to it that cause the most contention. Hence, running a resource stressing contender in parallel with a task creates the maximum increase in execution time for the task due to contention over resource r from any single co-runner.

A resource sensitive contender for a resource r, suffers the maximum possible interference by repeatedly making accesses to the resource that suffer from the most contention. Hence, running a resource sensitive contender in parallel with a task creates the maximum increase in execution time for any single co-running contender due to contention over resource r from the task. Note, resource stressing and resource sensitive contenders for a given resource are not necessarily one and the same.

Each task is further characterised by its *resource sensitivity* X_i^r and *resource stress* Y_i^r for each shared hardware resource $r \in H$. X_i^r captures the increase in execution time of task τ_i (from C_i to $C_i + X_i^r$) when it is executed in parallel with a resource stressing contender for resource r. Thus X_i^r models how much task τ_i behaves like a resource sensitive contender. Similarly, Y_i^r captures the increase in execution time of a resource sensitive contender (from C to $C + Y_i^r$) for resource r, when it is executed in parallel with task τ_i. Hence Y_i^r models how much task τ_i behaves like a resource stressing contender. With this model, the execution time of a task τ_i running on one core, subject to interference via shared hardware resource r from task τ_k running in parallel on another core, is increased by at most $\min(X_i^r, Y_k^r)$ i.e. from C_i to $C_i + \min(X_i^r, Y_k^r)$. The notation Γ_x is used to denote the set of tasks that execute on the same core (with index x) as the task of interest τ_i. Similarly, Γ_y is used to denote the set of tasks that execute on some other core (with index y). Each task τ_i is assumed to have a unique priority. $hp(i)$ (resp. $lp(i)$) is used to denote the set of tasks with higher (resp. lower) priority than task τ_i. Similarly, $hep(i)$ (resp. $lep(i)$) is used to denote the set of tasks with higher (resp. lower) than or equal priority to task τ_i.

The schedulability tests introduced in this paper are named using the following convention: **Cp*Sched*-*m*-*X***, where **C** indicates a contention-based test for **p** partitioned scheduling, using scheduling policy ***Sched***, which is either **FPPS** or **FPNS**. The test is applicable to systems with m cores, and makes use of information ***X***, which is either **D** or **R** meaning the deadlines or the response times of the tasks on other cores, or **fc** meaning fully composable, i.e. the test does not rely on any information about the tasks running on the other cores.

The MRSS task model assumes that the resource sensitivity X_i^r and resource stress Y_i^r parameters for each task τ_i are provided by timing analysis. Obtaining precise bounds for these parameters is a challenging timing analysis problem that is beyond the scope of this paper; nevertheless, below we give a brief overview of how such values could be estimated.

Using measurement-based timing analysis techniques, the resource sensitivity X_i^r can be obtained by capturing the maximum difference between the execution time of task τ_i when it runs in parallel with a resource stressing contender, and the corresponding execution time when it runs stand-alone, assuming that the same inputs and initial state are used in each case. Similarly, the resource stress Y_i^r can be obtained by capturing the maximum difference between the execution time of a resource sensitive contender when it runs in parallel with task τ_i, and the corresponding execution time of the contender when it runs stand-alone. As with measurement-based WCET estimation, such an approach needs to explore a representative set of inputs and initial states in order to obtain valid estimates. Further, resource stressing and resource sensitive contenders need to be carefully designed to meet their requirements in terms of creating/suffering the maximum amount of interference via contention over the resource [24]. Bounds on resource sensitivity and resource stress can also be obtained via static timing analysis. Static analysis first needs to compute an upper bound on the maximum number of accesses A_i^r that task τ_i can make to the resource. The resource sensitivity X_i^r can then be computed by determining the maximum increase in the execution time of task τ_i assuming that A_i^r accesses are made in contention with an arbitrary number of accesses emanating from one other core. Similarly, the resource stress Y_i^r equates to the maximum increase in the execution time of any arbitrary resource sensitive contender, due to contention over the resource caused by A_i^r accesses emanating from one other core.

The schedulability analysis presented in Section 3 assumes that the total interference occurring via multiple different resources can be upper bounded by the sum of the interference occurring via each of those resources individually. This assumption can reasonably be expected to hold provided that the resource contention is independent. In other words, that contention over one resource does not create additional contention over another resource. An example that breaks this assumption occurs with a cache that is shared between cores. In this case, cache thrashing [36] can result in additional accesses to main memory, and hence further contention and interference over that disparate resource. Cache partitioning (per core) would be an effective way of addressing this issue, thus improving timing predictability.

The analysis also assumes that the total interference occurring due to co-running tasks on multiple other cores can be upper bounded by the sum of the interference occurring due to co-running tasks on each of those cores individually. This assumption can reasonably be expected to hold provided that there are no discontinuities in the amount of interference that can occur that can be triggered by co-running tasks on a multiple cores, but not by co-runners on just one core. An example that breaks this assumption occurs with cache miss status handling registers (MSHR) [41]. In this case, contention from tasks on multiple cores can exhaust all of the available MSHRs, resulting in substantial blocking delays. Depending on the local memory level parallelism, utilizing all of the MSHRs is typically not be possible with just one contending core. Increasing the number of MSHRs, or reducing the local memory level parallelism, such that contention from all m cores cannot exhaust the set of MSHRs, are effective ways of addressing this problem [41] and restoring timing predictability. To validate the use of the analysis given in Section 3, each of the above assumptions needs to be assessed for the hardware architecture considered.

3 Schedulability Analysis

In this section, we introduce schedulability tests for the MRSS task model, assuming partitioned fixed priority preemptive scheduling (pFPPS) (Section 3.1), and partitioned fixed priority non-preemptive scheduling (pFPNS) (Section 3.2). In Section 3.3 we consider *composability* and derive context-independent schedulability tests for both pFPPS and pFPNS. The dominance relationships between the various tests are derived in Section 3.4.

3.1 pFPPS Schedulability Analysis

In the absence of any interference via shared hardware resources, the worst-case response time of task τ_i under pFPPS is given via standard response time analysis [25, 5]:

$$R_i = C_i + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1}$$

Adding cross-core interference considering each resource $r \in H$, we may compute the worst-case response time as follows:

$$R_i = C_i + \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \sum_{r \in H} I_i^r(R_i) \tag{2}$$

where $I_i^r(R_i)$ is an upper bound on the interference that may occur within the response time of task τ_i, via shared hardware resource r, due to tasks executing on the other cores.

The interference term $I_i^r(R_i)$ depends on: (i) the total resource sensitivity for resource r, denoted by $S_i^r(R_i, x)$, for the tasks executing on the same core x as task τ_i within its response time R_i; and (ii) the total resource stress on resource r, denoted by $E_i^r(R_i, y)$, that can be produced by tasks executing on each of the other cores y within an interval of length R_i. The total resource sensitivity $S_i^r(R_i, x)$ is computed based on the jobs that may execute within the worst-case response time of task τ_i, hence with reference to (1) we have:

$$S_i^r(R_i, x) \;=\; X_i^r \;+ \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil X_j^r \tag{3}$$

The total resource stress $E_i^r(R_i, y)$ due to tasks that execute on another core y in the interval R_i can be upper bounded as follows. Here, unlike in (3), the worst-case does not occur when these tasks are released synchronously, but rather when the resource contention occurs as late as possible for one job of a task, and then as early as possible for subsequent jobs. Further, tasks of any priority can cause interference when executing on other cores. Thus we have:

$$E_i^r(R_i, y) \;=\; \sum_{j \in \Gamma_y} \left\lceil \frac{R_i + D_j}{T_j} \right\rceil Y_j^r \tag{4}$$

The analysis in (4) does not make any assumptions about how long task τ_j needs to execute in order to cause an increase in execution time of up to Y_j^r in a task running on another core. In particular, there is no assumption that task τ_j needs to run for at least Y_j^r, since Y_j^r is a measure of the maximum increase in execution time of another task due to contention from task τ_j, not a measure of the time for which task τ_j needs to execute to cause that contention. Assuming that the execution causing contention can occur instantaneously, as is done in (4), is potentially pessimistic; however, it ensures that the analysis is sound even when there is considerable asymmetry in the (small) execution time required to stress a resource and the (large) increase in execution time of another task, which is sensitive to that resource stress. Since X_k^r represents the maximum sensitivity of a task τ_k when subject to continuous interference via resource r from a maximally resource stressing contender on one single other core, the maximum interference from other cores that can impact the response time of task τ_i via resource r can be upper bounded by:

$$I_i^r(R_i) = \sum_{\forall y \neq x} \min(E_i^r(R_i, y), S_i^r(R_i, x)) \tag{5}$$

This is the case, since the maximum interference due to contention from each core y cannot exceed the total resource stress $E_i^r(R_i, y)$ emanating from that core within a time R_i.

We refer to the schedulability test given by (2), (3), (4), and (5) as the **CpFPPS-m-D test**, since this test uses information about the *deadlines* of the tasks running on other cores.

A more precise analysis may be obtained by substituting R_j for D_j in (4) as follows, since a schedulable job of task τ_j cannot execute beyond its worst-case response time.

$$E_i^r(R_i, y) \;=\; \sum_{j \in \Gamma_y} \left\lceil \frac{R_i + R_j}{T_j} \right\rceil Y_j^r \tag{6}$$

Using this formulation, the response times of the tasks become interdependent. This problem can be solved via fixed point iteration. Here, an outer iteration starts with $R_i = C_i$, $R_j = C_j$ etc. for all tasks in the system, and repeatedly computes the response times for all tasks on all cores. This is done using the R_j values in the right hand side of (6) from the previous round, until all response times either converge (i.e. are unchanged from the previous round) or one of them exceeds the associated deadline. Since $E_i^r(R_i, y)$ in (6) is a monotonically non-decreasing function of each R_j, then on each round, each R_j value can only increase or remain the same, it cannot decrease. Thus, the outer fixed point iteration is guaranteed to either converge giving the set of schedulable $R_i \le D_i$ for all tasks in the system, or to result in some $R_i > D_i$, in which case that task and the system as a whole is unschedulable. We refer to the schedulability test given by (2), (3), (5), and (6) as the **CpFPPS-m-R test**, since it uses information about the *response times* of the tasks running on the other cores.

3.2 pFPNS Schedulability Analysis

In the absence of any cross-core contention and interference via shared hardware resources, the worst-case response time of task τ_i under pFPNS can be upper bounded via a sufficient response time analysis [13]:

$$R_i \;=\; \max_{k \in \Gamma_x \wedge k \in \mathbf{lep}(i)} (C_k) \;+\; \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left(\left\lfloor \frac{R_i - C_i}{T_j} \right\rfloor + 1 \right) C_j +\; C_i \tag{7}$$

Here, we have reformulated the sufficient analysis for FPNS [13] into a single equation. The changes involve compacting the blocking term (max()), and bringing the execution time C_i of the task under analysis into the equation. To compensate for the latter, the time interval in which higher priority tasks can execute is changed to $(R_i - C_i)$. This excludes the time at the end of the interval when task τ_i is executing non-preemptively. We also use a $\lfloor\ \rfloor + 1$ formulation rather than $\lceil\ \rceil$ to avoid the need for a term equal to the time unit granularity.

Similar to the case for pFPPS in (2), adding cross-core interference considering each resource $r \in H$, we may compute an upper bound on the worst-case response time as follows:

$$R_i \;=\; \max_{k \in \Gamma_x \wedge k \in \mathbf{lep}(i)} (C_k) \;+\; \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left(\left\lfloor \frac{R_i - C_i}{T_j} \right\rfloor + 1 \right) C_j +\; C_i \;+ \sum_{r \in H} I_i^r(R_i) \tag{8}$$

where $I_i^r(R_i)$ is an upper bound on the interference that may occur within the response time of task τ_i, via shared hardware resource r, due to tasks executing on other cores. Here, we make the sound, but potentially pessimistic, assumption that even though the execution time of task τ_i may be increased to more than C_i due to contention, only during the final C_i time units of the task's response time are other tasks on core x precluded from executing (i.e. we continue to use $(R_i - C_i)$ in the $\lfloor\ \rfloor$ function). Further, we use R_i in the final term, since cross-core contention still occurs during non-preemptive execution.

The interference term $I_i^r(R_i)$ depends on: (i) the total resource sensitivity for resource r, denoted by $S_i^r(R_i, x)$, for the tasks executing on the same core x as task τ_i within its response time R_i; and (ii) the total resource stress on resource r, denoted by $E_i^r(R_i, y)$, that can be produced by tasks executing on each of the other cores y within an interval of length R_i. The total resource sensitivity $S_i^r(R_i, x)$ is computed based on the jobs that may execute within the worst-case response time of task τ_i, hence with reference to (7) we have:

$$S_i^r(R_i, x) \;=\; \max_{k \in \Gamma_x \wedge k \in \mathbf{lep}(i)} (X_k^r) \;+\; \sum_{j \in \Gamma_x \wedge j \in \mathbf{hp}(i)} \left(\left\lfloor \frac{R_i - C_i}{T_j} \right\rfloor + 1 \right) X_j^r +\; X_i^r \tag{9}$$

The two equations (4) and (6) for the total resource stress $E_i^r(R_i, y)$ due to tasks that execute on another core y in the interval R_i depend only on the tasks parameters and response times, but not the scheduling policy per se. Thus by redefining $S_i^r(R_i, x)$ according to (9) for the non-preemptive case, we obtain the following pFPNS schedulability tests for the MRSS task model.

The **CpFPNS-m-D** test given by (8), (9), (4), and (5) makes use of the *deadlines* of the tasks running on the other cores.

The **CpFPNS-m-R** test given by (8), (9), (6), and (5) makes use of the *response times* of the tasks running on the other cores.

3.3 Composability

The schedulability analyses derived in Sections 3.1 and 3.2 make use of information about the resource contention due to tasks executing on other cores. In other words, these analyses requires that the resource stress (Y_j^r) values are known for all tasks executing on the other cores, as well as their other parameters i.e. T_j, D_j, R_j. While this results in tighter response time bounds, it also means that the analyses are not *fully composable*, since the schedulability of the tasks running on one core depend on the parameters of the tasks running on the other cores. A *fully composable* analysis can, however, be obtained by redefining (5) as follows:

$$I_i^r(R_i) = \sum_{\forall y \neq x} S_i^r(R_i, x) = (m-1) \cdot S_i^r(R_i, x) \tag{10}$$

This equates to assuming a worst-case scenario of resource stressing contenders for each resource r running on every core. This may be pessimistic on two counts: Firstly, the resource stressing contenders may cause significantly more interference than the tasks actually running on the other cores, and secondly, with more than one resource it may not be possible to maximally stress all resources simultaneously.

Using (10) results in *fully composable* context-independent schedulability tests. These tests are able to check the schedulability of task sets on each of the m cores in a system, without needing to know any of the parameters of the tasks on the other cores. We refer to the schedulability test given by (2), (3), and (10) as the **CpFPPS-m-fc** test. Similarly, we refer to the schedulability test given by (8), (9), and (10) as the **CpFPNS-m-fc** test.

Finally, an intermediate *partially composable* analysis can be provided if resource access regulation mechanisms or budgets are employed to limit the amount of contention emanating from each core. Let $F_i^r(t, y)$ be the maximum increase in execution time of a resource sensitive contender on another core that can occur due to contention over resource r caused by a resource stressing contender running on core y for a time period of t, subject to resource regulation. Partially composable analysis can be obtained by redefining (5) as follows:

$$I_i^r(R_i) = \sum_{\forall y \neq x} \min(F_i^r(R_i, y), S_i^r(R_i, x)) \tag{11}$$

Note, this analysis only holds if the resource regulation on each core y does not actually limit the accesses to each resource r made by tasks on that core over any time interval. Provided that is guaranteed, no actual runtime enforcement is necessary, the budget function $F_i^r(t, y)$ simply acts as an intermediate value that permits a separation of concerns and composition.

3.4 Dominance Relations

A schedulability test S is said to *dominate* another test Z for a given task model and scheduling algorithm, if every task set that is deemed schedulable according to test Z is also deemed schedulable by test S, and there exists some task sets that are schedulable according to test S, but not according to test Z.

Comparing the definitions of $E_i^r(R_i, y)$ given by (6) for the **CpFPPS-m-R** and **CpFPNS-m-R** tests and by (4) for the **CpFPPS-m-D** and **CpFPNS-m-D** tests, it is evident that each of the former tests deems schedulable all task sets that are schedulable according to the corresponding latter test. This is the case, since in any schedulable system, the response time of a task is no greater than its deadline ($R_j \leq D_j$), and hence the $E_i^r(R_i, y)$ term for the former tests, given by (6), is less then or equal to the equivalent term, given by (4), for the latter tests. Further, it is easy to see that there exist task sets that are schedulable according to the each of the former tests, but not according to the corresponding latter test due to a larger contention contribution emanating from the larger $E_i^r(R_i, y)$ term. Hence the **CpFPPS-m-R** test dominates the **CpFPPS-m-D** test, and the **CpFPNS-m-R** test dominates the **CpFPNS-m-D** test.

Comparing the definitions of $I_i^r(R_i)$ given by (5) for the **CpFPPS-m-D** and **CpFPNS-m-D** tests and by (10) for the **CpFPPS-m-fc** and **CpFPNS-m-fc** tests, it is evident that the former tests deems schedulable all task sets that are schedulable according to the corresponding latter test. Further, it is easy to see that there exist task sets that are schedulable according to the each of the former tests, but not according to the corresponding latter test due to a larger contention contribution emanating from the larger $I_i^r(R_i)$ term. Hence the **CpFPPS-m-D** test dominates the **CpFPPS-m-fc** test, and the **CpFPNS-m-D** test dominates the **CpFPNS-m-fc** test.

As dominance is transitive, we have: **CpFPPS-m-R** $\rightarrow$ **CpFPPS-m-D** $\rightarrow$ **CpFPPS-m-fc** and **CpFPNS-m-R** $\rightarrow$ **CpFPNS-m-D** $\rightarrow$ **CpFPNS-m-fc** where $S \rightarrow Z$ indicates that test S dominates test Z.

Finally, comparing a system of m cores to one with $m+1$ cores, where in each case the first m cores execute exactly the tasks, and the $m+1$ core system has extra tasks that executes on core $m+1$, then there is a dominance relationship between the systems as analysed by any of the schedulability tests. In other words, adding a core and the contention that it brings cannot improve schedulability for the tasks running on the existing cores, but may make their schedulability worse. Schedulability for m cores thus dominates that for $m+1$ cores with added tasks: **Cp*Sched*-m-*X*** $\rightarrow$ **Cp*Sched*-$(m+1)$-*X***

4 Priority Assignment

To maximize schedulability it is necessary to assign task priorities in an optimal way [14]. This section considers optimal priority assignment for the schedulability tests introduced in Section 3.

4.1 pFPPS Priority Assignment

Leung and Whitehead [31] showed that Deadline Monotonic Priority Ordering (DMPO) is optimal for constrained-deadline task sets with parameters (C, D, T) under fixed priority preemptive scheduling. We observe that this result also holds for constrained-deadline MRSS task sets compliant with model described in Section 2 and analysed according to the **CpFPPS-m-fc** test introduced in Section 3.3. This is because that formulation can be re-arranged to match the basic response time analysis (1), with the execution time of each task τ_k increased by $\sum_{r \in H}(m-1)X_k^r$. DMPO is also optimal for constrained-deadline MRSS task sets analysed according to the **CpFPPS-m-D** test, introduced in Section 3.1. Proof is given below using the standard apparatus for proving the optimality of such priority orderings, as described in section IV of [14]. This proof technique is applicable in cases where task priorities can be defined directly from fixed task parameters, for example periods and deadlines. To show that a priority assignment policy P (i.e. DMPO) is optimal, it suffices to prove that any task set that is schedulable according to the schedulability test considered using some priority assignment policy Q is also schedulable using priority ordering P. Proof is obtained by transforming priority ordering Q into priority ordering P, while ensuring that no tasks become unschedulable during the transformation. The proof proceeds by induction.

▶ **Theorem 1.** *Deadline Monotonic Priority Ordering is optimal for constrained-deadline MRSS task sets compliant with the model described in Section 2 and analysed according to the* ***CpFPPS**-m-**D*** *test introduced in Section 3.1.*

Proof. *Base case:* The task set is schedulable with priority order $Q = Q^k$, where k is the iteration count.

Inductive step: We select a pair of tasks that are at adjacent priorities i and j where $j = i + 1$ in priority ordering Q^k, but out of Deadline Monotonic relative priority order. Let these tasks be τ_A and τ_B, with τ_A having the higher priority in Q^k. Note that $D_A > D_B$ as the tasks are out of Deadline Monotonic relative order. Let i be the priority of task τ_A in Q^k and j be the priority of task τ_B. We need to prove that all of the tasks remain schedulable with priority order Q^{k-1}, which switches the priorities of these two tasks. There are four groups of tasks to consider:

hp(i): tasks in this set have higher priorities than both τ_A and τ_B in both Q^k and Q^{k-1}. Since the schedulability of these tasks is unaffected by the relative priority ordering of τ_A and τ_B, they remain schedulable in Q^{k-1}.

τ_A: Let $w = R_B$ be the response time of task τ_B in priority order Q^k. Since task τ_B is schedulable in Q^k, we have $w = R_B \leq D_B < D_A \leq T_A$, hence in (2), the contribution from τ_A within the response time of τ_B is exactly one job (i.e. C_A), and there is also a contribution of C_B from task τ_B itself. Considering interference, the total resource sensitivity $S^r_B(w, x)$ given by (3) depends only on the value w and fixed parameters of the set of tasks with priorities higher than or equal to task τ_B in Q^k that is τ_A, τ_B, and *hp*(i). Further, the total resource stress $E^r_B(w, y)$ due to tasks executing on some other core y depends only on the value of w and the fixed parameters of the tasks executing on that core. It follows that the interference term $I^r_B(w)$ given by (5) and used in (2) depends only on the value of w and the fixed parameters of the set of tasks τ_A, τ_B, and *hp*(i), as well as the fixed parameters of the tasks executing on all other cores. Now consider the response time of task τ_A under priority order Q^{k+1}. This response time is $R_A = w$, as there is exactly the same contribution from tasks τ_A, τ_B and all the higher priority tasks, and further the interference due to resource contention is the same, in other words $I^r_B(w)$ for Q^k equates to $I^r_A(w)$ for Q^{k+1}, since the value of w is the same, and the set of tasks that this term is dependent upon is unchanged (τ_A, τ_B, and *hp*(i) on core x, and all of the task on the other cores). Since $w < D_A$, task τ_A remains schedulable.

τ_B: as the priority of τ_B has increased its response time is no greater in Q^{k+1} than in Q^k. This is the case because the only change to the response time calculation for τ_B is the removal of the contribution from task τ_A, and also the removal of its contribution to the total resource sensitivity, and hence from the interference term $I^r_B(w)$. Thus τ_B remains schedulable.

lp(j) : tasks in this set have lower priorities than tasks τ_A and τ_B in both Q^k and Q^{k+1}. Since the schedulability of these tasks is unaffected by the relative priority ordering of tasks τ_A and τ_B, they remain schedulable.

All tasks therefore remain schedulable in Q^{k+1}.

At most $k = n(n-1)/2$ steps are required to transform priority ordering Q into P without any loss of schedulability ◀

Next, we consider optimal priority assignment with respect to the **CpFPPS-m-R** test introduced in Section 3.1. Davis and Burns proved in [12] that it is both sufficient and necessary to show that a schedulability test meets three simple conditions in order for Audsley's Optimal Priority Assignment (OPA) algorithm [4] algorithm to be applicable.

Condition 1: The schedulability of a task according to the test must be independent of the relative priority order of higher priority tasks.

Condition 2: The schedulability of a task according to the test must be independent of the relative priority order of lower priority tasks.

Condition 3: The schedulability of a task according to the test must not get worse if the task is moved up one place in the priority order (i.e. its priority is swapped with that of the task immediately above it in the priority order).

▶ **Theorem 2.** *The* ***CpFPPS-m-R*** *test, given in Section 3.1, is not compatible with Audsey's Optimal Priority Assignment (OPA) algorithm [4], and hence that algorithm cannot be used to obtain an optimal priority assignment with respect to the test.*

Proof. To prove non-compatibility, it suffices to show that any one of the three conditions set out in [12] and listed above is broken by the test. In this case, we show that Condition 1 does not hold. According to the **CpFPPS-m-R** test, the schedulability of a task τ_i on core x can depend on the response time of task τ_j on a different core y via $E^r_j(R_i, y)$ given by (6). In turn, the response time of task τ_j can depend on the response time of some higher priority task τ_k on the same core x as task τ_i via $E^r_k(R_j, x)$ also given by (6). Since the response time of task τ_k depends on its relative priority order among those tasks with higher priority than task τ_i, Condition 1 does not hold and therefore the **CpFPPS-m-R** test is not compatible with Audsley's OPA algorithm ◀

Although the **CpFPPS-m-R** test is not compatible with Audsley's OPA algorithm, the form of the test, with its dependence on the response times of other tasks, means that a back-tracking search, as proposed in [11], could potential be used to obtain a schedulable priority assignment without having to explore all possible priority orderings. The same applies to the **CpFPNS-m-R** test discussed in Section 4.2 below.

4.2 pFPNS Priority Assignment

George et al. [18] showed that Deadline Monotonic Priority Ordering (DMPO) is not optimal for constrained-deadline task sets with parameters (C, D, T) under fixed priority non-preemptive scheduling, and proved that Audsley's algorithm [4] is able to provide an optimal priority ordering in this case. We observe that this result also holds for constrained-deadline MRSS task sets compliant with the model described in Section 2 and analysed according to the **CpFPNS-m-fc** test introduced in Section 3.3. This is the case because the formulation can be re-arranged to match the basic response time analysis (7), with the execution time of each task τ_k increased by $(m-1)X^r_k$. Audsley's algorithm [4] is also optimal with respect to the **CpFPNS-m-D** test, as proved below.

▶ **Theorem 3.** *Audsley's algorithm [4] is optimal for constrained-deadline MRSS task sets compliant with the model described in Section 2 and analysed according to the* ***CpFPNS-m-D*** *test introduced in Section 3.2.*

Proof. It suffices to show that the schedulability test meets the three conditions, given in [12] and set out in Section 4.1. With respect to **Condition 1** and **Condition 2**, inspection of (8) shows that the first two terms are dependent on the set of lower and equal priority tasks $lep(i)$ and the set of higher priority tasks $hp(i)$ respectively, but do not depend on the relative priority order of the tasks within those sets. Considering the fourth term in (8), $I_i^r(t)$ is given by (5). In the definition of $I_i^r(t)$, the total resource sensitivity $S_i^r(t,x)$ is given by (9), which is dependent on the set of tasks $lep(i)$ and the set of tasks $hp(i)$, but does not depend on the relative priority order of the tasks within those sets. Finally, the total resource contention $E_i^r(t,y)$ given by (4) has no dependence on the relative priority order of the tasks in the sets $hp(i)$ and $lep(i)$ (or $lp(i)$), thus **Condition 1** and **Condition 2** hold.

With respect to **Condition 3**, moving task τ_i up one place in the priority order is equivalent to moving another task τ_h that also executes on core x from the set $hp(i)$ to the set $lep(i)$. Considering (8), such a change may increase the first term by no more than C_h, but is guaranteed to also reduce the second term by at least C_h. Further, with respect to the total resource sensitivity $S_i^r(t,x)$, given by (9), such a change may increase the first term by no more than X_h^r, but is guaranteed to also reduce the second term by at least X_h^r. There is no change to the total resource stress $E_i^r(t,y)$ given by (4). Hence the schedulability of task τ_i cannot get worse if the task is moved up one place in the priority order ◀

Finally, we note that the **CpFPNS-m-R** test is not compatible with Audsley's OPA algorithm, since it breaks Condition 1, as proven below.

▶ **Theorem 4.** *The **CpFPNS-m-R** test given in Section 3.1, is not compatible with Audsey's Optimal Priority Assignment (OPA) algorithm [4], and hence that algorithm cannot be used to obtain an optimal priority assignment with respect to the test.*

Proof. Proof follows via exactly the same argument as given in the proof of Theorem 2 in Section 4.1, replacing the **CpFPPS-m-R** test with the **CpFPNS-m-R** test ◀

5 Evaluation

In this section, we present an empirical evaluation of the schedulability tests introduced in Section 3 for MRSS task sets executing on a multi-core system, assuming a single hardware resource shared between all cores. (Note, multiple shared hardware resources resulting in the same total interference would have the same impact on schedulability, due to the summation terms in (2) and (8)). Experiments were performed for 1, 2, 3, and 4 cores[3], with the single core case considered for comparison purposes.

5.1 Task Set Parameter Generation

The task set parameters used in our experiments were generated as follows:

- Task utilizations ($U_i = C_i/T_i$) were generated using the Dirichlet-Rescale (DRS) algorithm [21] (open source Python software [20]) providing an unbiased distribution of utilization values that sum to the total utilization U required.
- Task periods T_i were generated according to a log-uniform distribution [15] with a factor of 100 difference between the minimum and maximum possible period. This represents a spread of task periods from 10ms to 1 second, as found in many real-time applications. (When considering non-preemptive scheduling, a factor of 10 difference was used, otherwise most task sets would not be schedulable).

[3] The analysis scales to more than 4 cores; however, we limited consideration to this range, since 4 cores represents a typical cluster size beyond which sharing hardware resources can become a significant performance bottleneck.

- Task deadlines D_i were set equal to their periods T_i.
- The stand-alone execution time of each task was given by: $C_i = U_i \cdot T_i$.
- Task resource sensitivity values X_i^r were determined as follows. The DRS algorithm was used to generate task resource sensitivity utilization values V_i^r, such that the total resource sensitivity utilization was SF (the Sensitivity Factor, default $SF = 0.25$) times the total task utilization (i.e. $\sum_{\forall i} V_i^r = U \cdot SF$), and each individual task resource sensitivity utilization was upper bounded by the corresponding task utilization (i.e. $V_i^r \leq U_i$). Each task resource sensitivity value was then given by $X_i^r = V_i^r \cdot T_i$.
- Task resource stress values Y_i^r were set to a fixed proportion of the corresponding resource sensitivity value $Y_i^r = X_i^r \cdot RF$, where RF is the Stress Factor, default $RF = 0.5$.

The default value for the Sensitivity Factor ($SF = 0.25$) was set to approximately twice the average value (13.6%) obtained for the tasks in the industry case study described in the Appendix. This is justified since the case study considers a single shared hardware resource, whereas in practice contention would likely occur via multiple shared hardware resources, resulting in higher levels of interference. The default value for the Stress Factor ($RF = 0.5$) was set within the range found in the case study (0.23 to 1.58). Further, specific experiments were also used to evaluate performance over a wide range of values for these parameters.

5.2 Experiments

The experiments considered systems with 1, 2, 3, and 4 cores, with a different task set (generated according to the same parameters) assigned to each core. The per core task set utilization U (shown on x-axis of the graphs) was varied from 0.05 to 0.95. For each utilization value examined, 1000 task sets were generated for each core considered, (100 in the case of experiments using the weighted schedulability measure [6]). The default cardinality of the task sets on each core was $n = 10$. In the experiments, a system was deemed schedulable if and only if the different task sets assigned to each of its m cores were schedulable, i.e. if all $m \cdot n$ tasks in the system were schedulable. With a single core, there is no cross-core resource contention and hence no interference, and so task set schedulability can be determined via standard response time analysis. With multiple cores, contention and the resulting interference was modelled as described in Section 2. The experiments investigated the performance of the following schedulability tests for partitioned fixed priority preemptive and non-preemptive scheduling:

- **No-CpFPPS-m**: The exact analysis of pFPPS [25, 5] with no contention, recapped in Section 3.1, and given by (1).
- **CpFPPS-m-R**: The response time based analysis of pFPPS with contention, introduced in Section 3.1, and given by (2), (3), (5), and (6).
- **CpFPPS-m-D**: The deadline based analysis of pFPPS with contention, introduced in Section 3.1, and given by (2), (3), (4), and (5).
- **CpFPPS-m-fc**: The fully composable analysis of pFPPS with contention, introduced in Section 3.3, and given by (2), (3), and (10).
- **No-CpFPNS-m**: The sufficient analysis of pFPNS [13] with no contention, recapped in Section 3.2, and given by (7)).
- **CpFPNS-m-R**: The response time based analysis of pFPNS with contention, introduced in Section 3.2, and given by (8), (9), (6), and (5).
- **CpFPNS-m-D**: The deadline based analysis of pFPNS with contention, introduced in Section 3.2, and given by (8), (9), (4), and (5).

- **CpFPNS-m-fc**: The fully composable analysis of pFPNS with contention, introduced in Section 3.3, and given by (8), (9), and (10).

For consistency of comparison, Deadline Monotonic Priority Ordering (DMPO) [31] was used to assign priorities to tasks on the individual cores. As shown in Section 4, DMPO is optimal with respect to the **No-CpFPPS-m**, **CpFPPS-m-fc**, and **CpFPPS-m-D** tests, but only a heuristic policy with respect to the **CpFPPS-m-R** test and the tests for fixed priority non-preemptive scheduling.

Note, the results for the fully composable analyses (tests **CpFPPS-m-fc** and **CpFPNS-m-fc**) equate to the performance obtained via the use of resource sensitivity information only, as outlined in prior works [36, 16, 33, 24].

5.3 Results

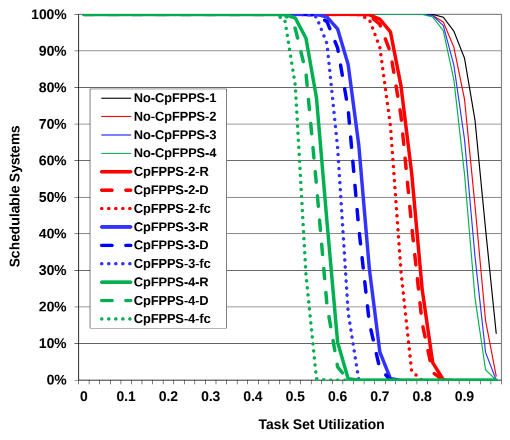
In the first experiment, we compared the performance of the various schedulability tests, assuming 1, 2, 3, and 4 cores, using the default parameters given in Section 5.1. The *Success Ratio*, i.e. the percentage of systems generated that were deemed schedulable, is shown for each of the pFPPS schedulability tests in Figure 1a, and for the pFPNS schedulability tests in Figure 1b. The dominance relationships between the tests, discussed in Section 3.4, are evidenced by the lines on the graphs. Note, schedulability depends on the number of cores even when contention is not taken into account. This is because for an m-core system the task sets on all m cores have to be schedulable for the system to be deemed schedulable.

Observe, that the performance advantage that the context-independent tests have over their context-dependent counterparts is more pronounced with pFPPS than with pFPNS. The reason for this is that the increased response times due to the blocking factor with pFPNS mean that the critical task(s) (those that become unschedulable as utilization is increased) are much more likely to be medium or high priority tasks than is the case with pFPPS. For higher priority tasks, the balance between total resource sensitivity $S_i^r(R_i, x)$ and total resource stress $E_i^r(R_i, y)$ tends towards the latter being larger, since $E_i^r(R_i, y)$ includes a contribution from all of the tasks on core y, while $S_i^r(R_i, x)$ only includes a contribution from a single lower priority (blocking) task in the case of pFPNS, and no lower priority tasks at all in the case of pFPPS. When $E_i^r(R_i, y)$ exceeds $S_i^r(R_i, x)$ then the performance of the context-independent tests is reduced to that of their context-dependent counterparts.

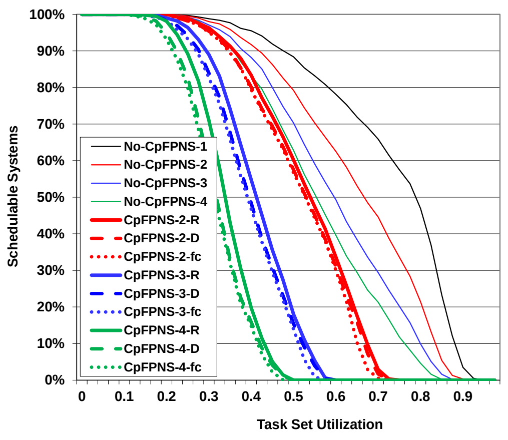
In the second set of experiments, we used the weighted schedulability measure [6] to assess schedulability test performance, while varying an additional parameter. In these experiments, the other parameters were set to their default values given in Section 5.1. In all of the weighted schedulability experiments the relative performance of the different tests follows the pattern illustrated in the first experiment, as dictated by the dominance relationships.

The results of varying the Sensitivity Factor SF from 0.05 to 0.5 in steps of 0.05, are shown in Figure 2a for pFPPS, and Figure 2b for pFPNS. Recall that the Sensitivity Factor determines the ratio of the total resource sensitivity utilization to the total task utilization. As expected, increasing the Sensitivity Factor and hence the amount of interference that tasks can be subject to due to cross-core contention for resources results in a rapid decline in the weighted schedulability measure for all of the tests that take into account contention.

The results of varying the Stress Factor RF from 0 to 1.2 in steps of 0.1 are shown in Figure 3a for pFPPS, and Figure 3b for pFPNS. Recall that the Stress Factor determines the ratio of the resource stress for each task to its resource sensitivity. Here, it is interesting to note that interference effective saturates once the Stress Factor reaches 1.0. By then, the total resource stress $E_i^r(t, y)$, given by (4) or (6), emanating from each additional core y in a

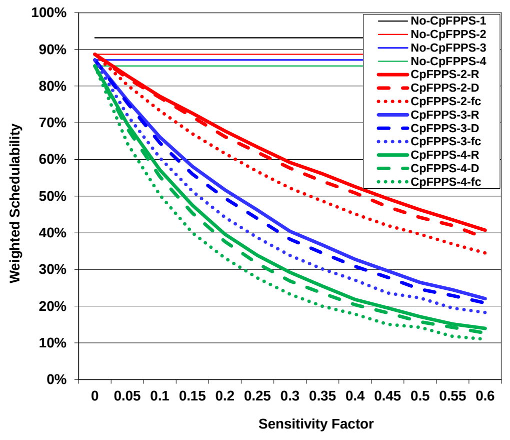


(a) pFPPS

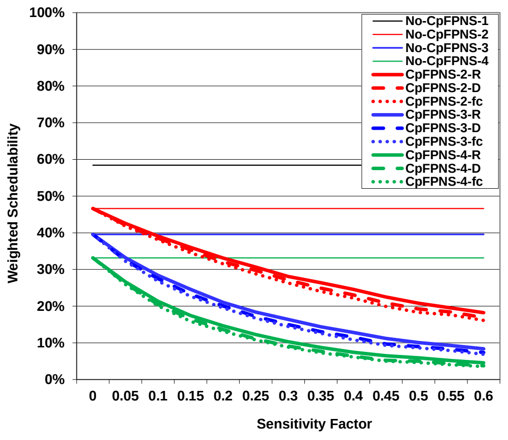


(b) pFPNS

Figure 1 Success Ratio: Varying task set utilization



(a) pFPPS

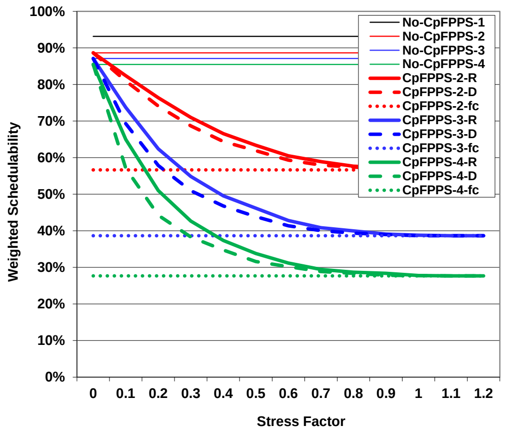


(b) pFPNS

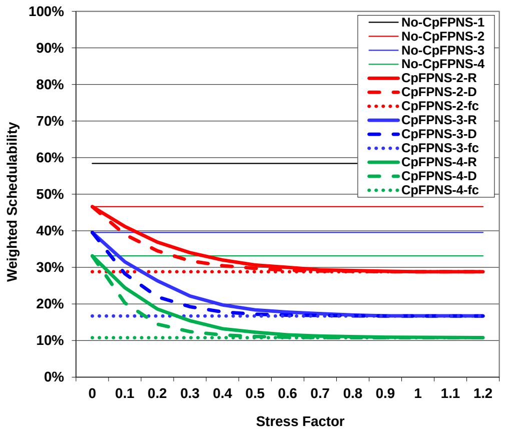
Figure 2 Weighted Schedulability: Varying Sensitivity Factor (SF)

time interval t tends to exceed the total resource sensitivity $S_i^r(t,x)$, given by (3), for core x in that same time interval. Hence, for pFPPS the **CpFPPS-m-R** and **CpFPPS-m-D** tests reduce to exactly the same performance as the **CpFPPS-m-fc** test. Similarly, for pFPNS the **CpFPNS-m-R** and **CpFPNS-m-D** tests reduce to exactly the same performance as the **CpFPNS-m-fc** test. This is because the $\min(E_i^r(t,y), S_i^r(t,x))$ term in (5) ceases to reduce the value in the summation below $S_i^r(t,x)$. At the other extreme a Stress Factor RF of zero means that $E_i^r(t,y) = 0$ whether computed via (4) or (6). For pFPPS, the **CpFPPS-m-R** and **CpFPPS-m-D** tests therefore have the same performance as the no contention **No-CpFPPS-m** test, and similarly for pFPNS the **CpFPNS-m-R** and **CpFPNS-m-D** tests have the same performance as the **No-CpFPNS-m** test. Between the two extremes, the smaller values of $E_i^r(t,y)$ given by (6) as opposed to (4) mean that the **CpFPPS-m-R** test outperforms the **CpFPPS-m-D** test, and similarly the **CpFPNS-m-R** test outperforms the **CpFPNS-m-D** test.

The results of varying the cardinality of task sets running on each core from 2 to 32 in

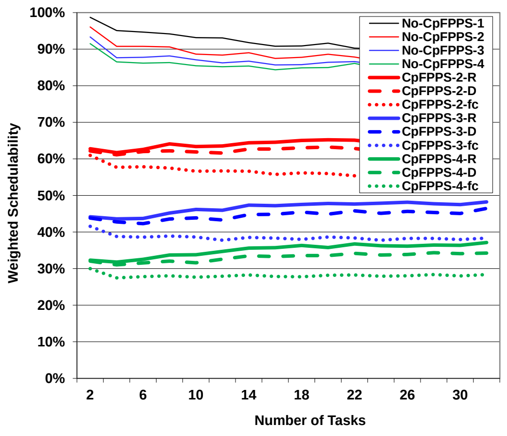


(a) pFPPS

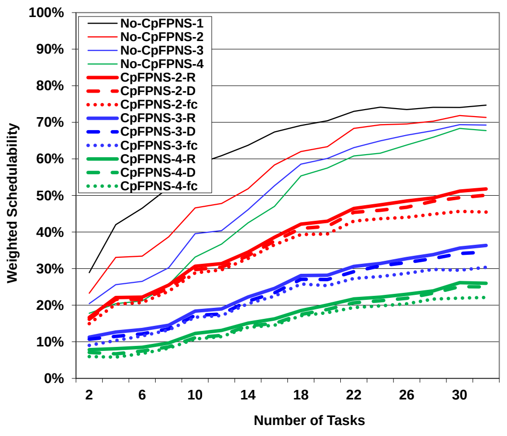


(b) pFPNS

Figure 3 Weighted Schedulability: Varying Stress Factor (RF)



(a) pFPPS

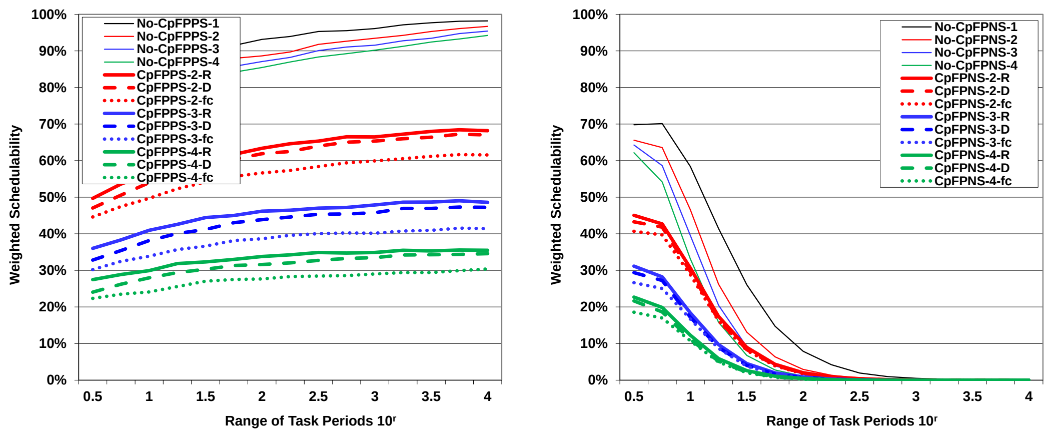


(b) pFPNS

Figure 4 Weighted Schedulability: Varying number of tasks in each task set

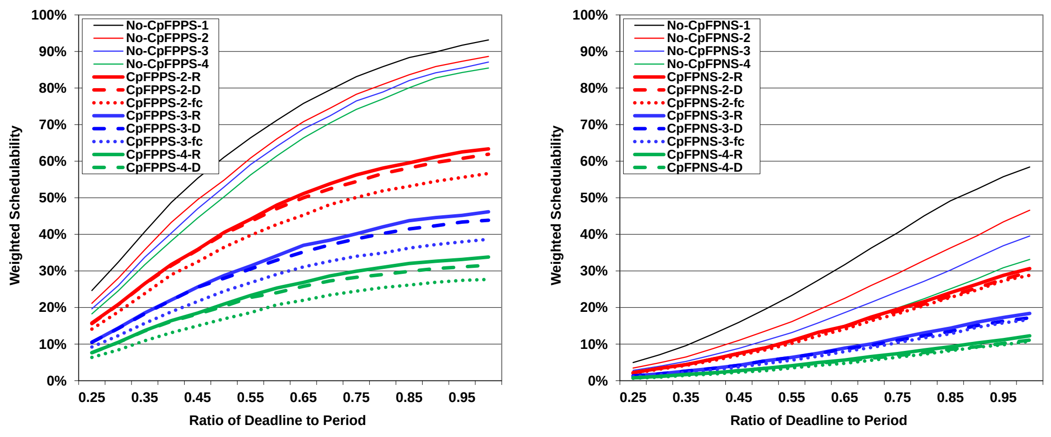
steps of 2 are shown in Figure 4a for pFPPS, and Figure 4b for pFPNS. In the preemptive case, task set cardinality typically has only a limited effect on schedulability test performance; however, in the non-preemptive case (Figure 4b), larger task sets equate to smaller execution times for each task and hence smaller blocking factors. Thus schedulability improves with increasing cardinality for all of the pFPNS schedulability tests. In the preemptive case (Figure 4a) the gap between the **CpFPPS-m-R** and **CpFPPS-m-D** tests and the **CpFPPS-m-fc** test increases with larger numbers of tasks. This is due to changes in the shape of the total resource stress function $E_i^r(t,y)$, which typically consists of many small steps when there are a large number of tasks, and fewer larger steps when there are a smaller number of tasks. As the function $E_i^r(t,y)$ is above the same gradient line in both cases, this difference acts to improve schedulability for the **CpFPPS-m-R** and **CpFPPS-m-D** tests at higher task set cardinality. The same effect is also evident in Figure 4a for the pFPNS schedulability tests.

The effects of varying the range of task periods (ratio of the max/min possible task period) from $10^{0.5} \approx 3$ to $10^4 = 10,000$ are shown in Figure 5a for pFPPS, and Figure 5b for



(a) pFPPS **(b)** pFPNS

Figure 5 Weighted Schedulability: Varying range of task periods



(a) pFPPS **(b)** pFPNS

Figure 6 Weighted Schedulability: Varying ratio of deadlines to periods

pFPNS. As expected, increasing the range of task periods results in a gradual improvement in pFPPS schedulability test performance, a well-known effect with fixed priority preemptive scheduling. In contrast, with non-preemptive scheduling, once the range of task periods exceeds 100 (i.e. $r = 2$), hardly any task sets are schedulable. This happens because tasks with short periods (and deadlines) cannot tolerate being blocked by tasks with long periods and commensurate large execution times.

Finally, the results of varying task deadlines from 25% to 100% of the task's period are shown in Figure 6a for pFPPS, and Figure 6b for pFPNS. As expected, schedulability improves for all approaches as task deadlines are increased. Further, the performance advantage of the **CpFPPS-m-R** test over the **CpFPPS-m-D** test increases with increasing deadlines. This occurs because larger deadlines provide a more pessimistic approximation of response times for schedulable tasks, impacting the total resource stress as assumed by the **CpFPPS-m-D** test.

6 Conclusions

The main contributions of this paper are the Multi-core Resource Stress and Sensitivity (MRSS) task model and its accompanying schedulability analyses. The MRSS task model:

- Characterizes how much each task stresses shared hardware resources and how much it is sensitive to such resource stress.
- Provides a simple yet effective interface between timing analysis and schedulability analysis, facilitating a separation of concerns that retains the advantages of the traditional two-step approach to timing verification.
- Caters for a variety of different shared hardware resources in a way that is both generic and versatile.

The accompanying schedulability analyses:

- Provide efficient context-dependent and context independent schedulability tests for both fixed priority preemptive and fixed priority non-preemptive scheduling.
- Exhibit dominance relationships illustrating the trade-off between context independence and schedulability.
- Were proven compatible or incompatible with efficient optimal priority assignment algorithms.
- Were subject to a systematic evaluation illustrating their effectiveness across a wide range of parameter values.

In future, we aim to investigate task allocation strategies for partitioned fixed priority scheduling of MRSS tasks. Details of a preliminary case study that explores the resource stress and resource sensitivity of tasks from a Rolls-Royce aero-engine control system are given in the appendix. This case study provides an underpinning proof-of-concept for the MRSS task model.

A Case Study

In this appendix, we present a preliminary case study that investigates the resource stress and resource sensitivity of tasks from a real-time industrial application. The purpose of this case study is not to try to determine definitive values for task WCETs, resource sensitivities and resource stresses, in itself a challenging research problem that is beyond the scope of this work. Rather our aim is to obtain proof-of-concept data to act as an exemplar underpinning the MRSS task model and its accompanying schedulability analysis.

The case study focuses on a set of 24 tasks from a Rolls-Royce aero engine control system or FADEC (Full Authority Digital Engine Controller). The industrial software was developed in SPARK-Ada and has been verified according to DO-178C standards (level A). The software was provided in an anonymized object code format, derived from that used in the case studies reported in [29] and [30]. The tasks have object code libraries ranging in size from 300 KBytes to 40 MBytes, including compiled in data structures, but not including any framework or Linux additions. The software was originally designed to run on a Rolls-Royce specific packaged processor that integrates a single core, memory, I/O, and tracing interfaces; however, for research purposes, it was ported to run on a Raspberry Pi 3B+ [30], along with a framework that facilitates taking timing measurements [7].

The Raspberry Pi 3B+ uses a Broadcom BCM2837 System-on-Chip with a quad-core ARM Cortex-A53 processor. It has a 16 KByte L1 data cache, 16 KByte L1 instruction cache, 512 KByte L2 shared cache, and 1 GByte of DDR2-DRAM. The L2 cache was, as is the default, configured for use as local memory for the GPU[4], and so was not available to the four CPUs. The experimental hardware set-up involved a cluster of Raspberry Pi 3B+s, configured to run at a clock frequency of 600MHz, so as to eliminate any possible disruption due to thermal throttling. The cluster was powered by specialized power rails to ensure a stable supply voltage and frequency. The Pi 3B+s used the Raspberry Pi OS Lite (updated on 01/25/2021) and the Linux Kernel 5.10.11-v7+. The `isolcpus` Linux option was used to minimize activity on the two cores used for the experiments. Timing measurements were obtained using a nanosecond clock, and cross-referenced against a 600MHz cycle counter. Prior to each run of a task, the L1 data and L1 instruction caches were flushed. Given that the L2 cache was not accessible to the CPUs, the case study focussed on the key shared hardware resource, main memory (DDR2-DRAM).

A.1 Case Study Experiments

For each of the 24 tasks, we considered 10,000 randomly selected traces of execution. When a task was called for a specific trace, each of its input parameters was set to a random value based on the type (float, integer, or boolean) and the range of values permitted. The inputs were thus randomized, but nevertheless reproducible via the trace number, which controlled the random seed used. In the following, for brevity we use *trace* to mean a task with a specific set of input parameters corresponding to the trace number.

In **Experiment A.1**, for each trace we obtained the stand-alone execution time, the resource sensitivity, and the resource stress as measured against each of the three contenders described below. These values were obtained by: (i) running the trace stand alone, (ii) running the trace in parallel with the contender, (iii) running the contender stand alone. In (i) and (ii) the execution time of the trace was recorded. In addition, in (ii) the number of times L that the contender looped while the trace was running was recorded, along with the execution time of the contender for that number of loops. Finally, in (iii) the stand-alone execution time of the contender was recorded for L loops. The loop count L thus enabled comparable measurements to be made irrespective of the trace execution times. The stand-alone execution time of the trace came directly from (i), while the resource sensitivity (per contender) for the trace was given by the difference between the trace execution times in (i) and (ii), and the resource stress for the trace by the difference in the contender's execution times in (ii) and (iii).

We repeated the runs for each trace 9 times to ensure consistency. Post processing of the raw timing data was used to eliminate anomalies caused by the kernel scheduler tick and the clock synchronization interrupt, neither of which could be disabled. The cycle counter was configured to pause when the scheduler was running, and so we were able to detect and eliminate anomalies due to the scheduler by comparing nanosecond clock and cycle counter readings. Measurement noise caused by the clock synchronization interrupt was more difficult to detect; however, we were able to filter out these anomalies by taking the median value for the 9 repeated runs for each trace, and by using the 95th percentile value (over the 10,000 traces) as the reference "maximum" increase in execution time for each task and contender.

Three contenders were used that cause contention by repeatedly accessing main memory. The contenders both stress the resource and are sensitive to contention. The three contenders have a similar structure, they differ only in the instruction patterns used: Read-Read (RR), Read-Write (RW), and Write-Write (WW). The read and write operations both compile down

[4] The case study software does not use the GPU.

to a single instruction. Each contender loop body included 100 memory access instructions, ensuring that the loop overhead, i.e. checking when the contender should stop, was small in comparison to the loop body. Hence each contender achieved close to the maximum possible load in terms of instructions that access memory and cause contention. The addresses used were such that the accesses had to go to memory, rather than being satisfied by the L1 cache. A handshaking protocol was used between task and contender to ensure that the contender started before and finished after the task. Further, dummy loops with no memory accesses were added before and after each task, to ensure that the experimental framework did not cause extra interference on the contender when it was running but the task was not.

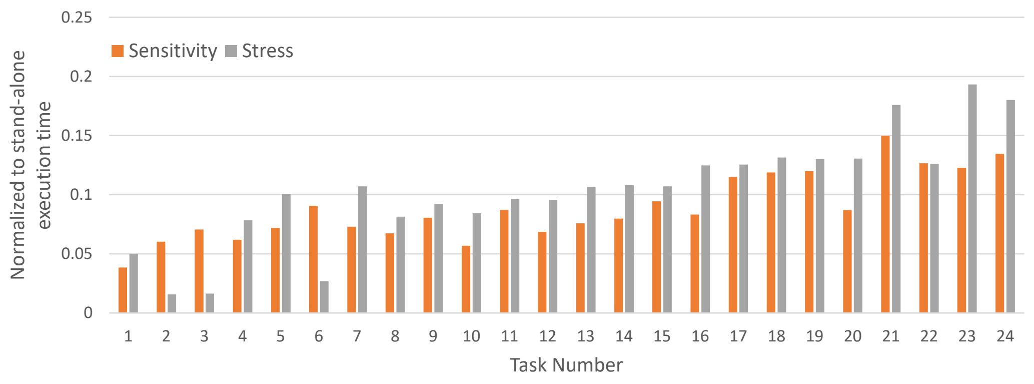


Figure 7 Estimated resource stress and resource sensitivity values for 24 tasks from a Rolls-Royce aero-engine control systems normalized to the task's estimated stand-alone WCET

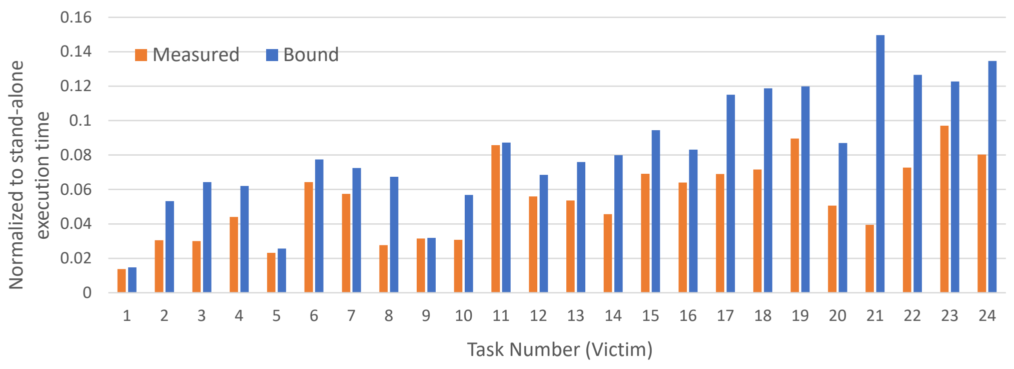


Figure 8 Increase in execution time of a (victim) task co-running with its paired task. Maximum observed value and computed bound derived from resource sensitivity and resource stress values, normalized to the stand-alone execution time of the victim task.

Figure 7 shows the results of Experiment A.1, for 24 tasks from the Rolls-Royce aero-engine application, giving their maximum resource sensitivity and maximum resource stress normalized to the task's maximum stand-alone execution time. Note, the tasks appear in the figure ordered by their maximum stand-alone execution time, largest first. The RW contender was responsible for the maximum increase in task execution time (resource sensitivity) in all 24 cases. However, in terms of which contender suffered the maximum increase in execution time due to the task (i.e. resource stress), this was the RR contender in 2 cases, the RW contender in 3 cases, and the WW contender in 19 cases. Running a contender in parallel with a task increased the task's execution time by between 3.8% and 15.0% compared to stand-alone execution, thus characterizing the tasks' resource sensitivity. Further, the contender's execution time increased by between 1.5% and 19.3% of the task's stand-alone execution time, thus characterizing the tasks' resource stress. The ratio of resource stress to resource sensitivity for each task varied from 0.23 to 1.58. Some negative correlation can be observed between the stand-alone execution time and the percentage resource sensitivity and resource stress, with longer running tasks often having lower percentage values for these metrics. This is to be expected, since longer tasks typically spend more of their execution time in loops, running code that is cached, and therefore causes less resource contention.

As well as running tasks (traces) in parallel with the synthetic contenders, we also conducted **Experiment A.2**, running pairs of tasks in parallel on different cores. For each pair of tasks, we ran 10,000 pairs of their traces in parallel, with the inputs randomly selected as described previously. Figure 8 shows the maximum increase in execution time for each (victim) task due to cross-core contention from the task it was paired with. (The tasks were sorted by stand-alone execution time and then paired 1-2, 3-4, 5-6 and so on). The values shown are the maximum over the 10,000 pairs of traces, and are normalized to the stand-alone execution time of the victim task. Also shown is the bound computed from the minimum of (i) the resource sensitivity for the victim task and (ii) the resource stress for the task it was paired with, both obtained via Experiment A.1 using the synthetic contenders. The maximum measured increase in execution time is no greater than the computed bound. On average it is approx. 69% of the bound, and varies between 26% and 99%.

This preliminary case study underpins the MRSS task model, illustrating the relevance of using both resource sensitivity and resource stress to characterize cross-core contention, and thus bound interference.

References

1 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 3–11. IEEE, 2020. URL: `https://doi.org/10.1109/RTSS49844.2020.00012`, `doi:10.1109/RTSS49844.2020.00012`.

2 Sebastian Altmeyer, Robert I. Davis, Leandro Soares Indrusiak, Claire Maiza, Vincent Nélis, and Jan Reineke. A generic and compositional framework for multicore response time analysis. In Julien Forget, editor, *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 129–138. ACM, 2015. URL: `https://doi.org/10.1145/2834848.2834862`, `doi:10.1145/2834848.2834862`.

3 Björn Andersson, Hyoseung Kim, Dionisio de Niz, Mark H. Klein, Ragunathan Rajkumar, and John P. Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Trans. Embed. Comput. Syst.*, 17(3):71:1–71:29, 2018. URL: `https://doi.org/10.1145/3203407`, `doi:10.1145/3203407`.

4 Neil C. Audsley. On priority assignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44, 2001. URL: `https://doi.org/10.1016/S0020-0190(00)00165-4`, `doi:10.1016/S0020-0190(00)00165-4`.

5 Neil C. Audsley, Alan Burns, Michael Richardson, Kenneth W. Tindell, and Andrew J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292(8), September 1993. URL: `https://digital-library.theiet.org/content/journals/10.1049/sej.1993.0034`.

6 Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *International*

Workshop on Operating Systems Platforms for Embedded Real-Time Applications, pages 33–44, 2010.

7 Iain Bate, David Griffin, and Benjamin Lesage. Establishing confidence and understanding uncertainty in real-time systems. In Liliana Cucu-Grosjean, Roberto Medina, Sebastian Altmeyer, and Jean-Luc Scharbarg, editors, *28th International Conference on Real Time Networks and Systems, RTNS 2020, Paris, France, June 10, 2020*, pages 67–77. ACM, 2020. URL: `https://doi.org/10.1145/3394810.3394816`, `doi:10.1145/3394810.3394816`.

8 Sheng-Wei Cheng, Jian-Jia Chen, Jan Reineke, and Tei-Wei Kuo. Memory bank partitioning for fixed-priority tasks in a multi-core system. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 209–219. IEEE Computer Society, 2017. URL: `https://doi.org/10.1109/RTSS.2017.00027`, `doi:10.1109/RTSS.2017.00027`.

9 Dakshina Dasari, Björn Andersson, Vincent Nélis, Stefan M. Petters, Arvind Easwaran, and Jinkyu Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2011, Changsha, China, 16-18 November, 2011*, pages 1068–1075. IEEE Computer Society, 2011. URL: `https://doi.org/10.1109/TrustCom.2011.146`, `doi:10.1109/TrustCom.2011.146`.

10 Robert I. Davis, Sebastian Altmeyer, Leandro Soares Indrusiak, Claire Maiza, Vincent Nélis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real Time Syst.*, 54(3):607–661, 2018. URL: `https://doi.org/10.1007/s11241-017-9285-4`, `doi:10.1007/s11241-017-9285-4`.

11 Robert I. Davis and Alan Burns. On optimal priority assignment for response time analysis of global fixed priority pre-emptive scheduling in multiprocessor hard real-time systems. Technical Report YCS-2010-451, University of York, Computer Science Dept., 2010.

12 Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real Time Syst.*, 47(1):1–40, 2011. URL: `https://doi.org/10.1007/s11241-010-9106-5`, `doi:10.1007/s11241-010-9106-5`.

13 Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real Time Syst.*, 35(3):239–272, 2007. URL: `https://doi.org/10.1007/s11241-007-9012-7`, `doi:10.1007/s11241-007-9012-7`.

14 Robert I. Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. A review of priority assignment in real-time systems. *J. Syst. Archit.*, 65:64–82, 2016. URL: `https://doi.org/10.1016/j.sysarc.2016.04.002`, `doi:10.1016/j.sysarc.2016.04.002`.

15 Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, July 2010. URL: `http://retis.sssup.it/waters2010/waters2010.pdf`.

16 Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the suitability of the NGMP multi-core processor in the space domain. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 175–184. ACM, 2012. URL: `https://doi.org/10.1145/2380356.2380389`, `doi:10.1145/2380356.2380389`.

17 Rudolf Fuchsen. How to address certification for multi-core based IMA platforms: Current status and potential solutions. In *29th Digital Avionics Systems Conference*, pages 5.E.3–1–5.E.3–11, 2010. `doi:10.1109/DASC.2010.5655461`.

18 Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and nonpreemptive real-time uniprocessor scheduling. Technical report, INRIA Research Report, No. 2966, 1996. URL: `https://hal.inria.fr/inria-00073732`.

19 Georgia Giannopoulou, Kai Lampka, Nikolay Stoimenov, and Lothar Thiele. Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 63–72. ACM, 2012. URL: `https://doi.org/10.1145/2380356.2380372`, `doi:10.1145/2380356.2380372`.

20 David Griffin, Iain Bate, and Robert I. Davis. Dirichlet-Rescale (DRS) algorithm software: dgdguk/drs: v1.0.0 available at `https://doi.org/10.5281/zenodo.4118059`, December 2020.

21 David Griffin, Iain Bate, and Robert I. Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 76–88. IEEE, 2020. URL: `https://doi.org/10.1109/RTSS49844.2020.00018`, `doi:10.1109/RTSS49844.2020.00018`.

22 Mohamed Hassan. On the off-chip memory latency of real-time systems: Is DDR DRAM really the best option? In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 495–505. IEEE Computer Society, 2018. URL: `https://doi.org/10.1109/RTSS.2018.00062`, `doi:10.1109/RTSS.2018.00062`.

23 Wen-Hung Huang, Jian-Jia Chen, and Jan Reineke. MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 158:1–158:6. ACM, 2016. URL: `https://doi.org/10.1145/2897937.2898046`, `doi:10.1145/2897937.2898046`.

24 Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. Slow and steady: Measuring and tuning multicore interference. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*, pages 200–212. IEEE, 2020. URL: `https://doi.org/10.1109/RTAS48715.2020.000-6`, `doi:10.1109/RTAS48715.2020.000-6`.

25 Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986. URL: `https://doi.org/10.1093/comjnl/29.5.390`, `doi:10.1093/comjnl/29.5.390`.

26 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark H. Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and reducing memory interference in cots-based multi-core systems. *Real Time Syst.*, 52(3):356–395, 2016. URL: `https://doi.org/10.1007/s11241-016-9248-1`, `doi:10.1007/s11241-016-9248-1`.

27 Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, James H. Anderson, and F. Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real Time Syst.*, 53(5):709–759, 2017. URL: `https://doi.org/10.1007/s11241-017-9272-9`, `doi:10.1007/s11241-017-9272-9`.

28 Kai Lampka, Georgia Giannopoulou, Rodolfo Pellizzoni, Zheng Wu, and Nikolay Stoimenov. A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. *Real Time Syst.*, 50(5-6):736–773, 2014. URL: `https://doi.org/10.1007/s11241-014-9211-y`, `doi:10.1007/s11241-014-9211-y`.

29 Stephen Law and Iain Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 189–199. IEEE Computer Society, 2016. URL: `https://doi.org/10.1109/ECRTS.2016.21`, `doi:10.1109/ECRTS.2016.21`.

30 Benjamin Lesage, Stephen Law, and Iain Bate. TACO: an industrial case study of test automation for coverage. In Yassine Ouhammou, Frédéric Ridouard, Emmanuel Grolleau, Mathieu Jan, and Moris Behnam, editors, *Proceedings of the 26th International Conference on Real-Time Networks and Systems, RTNS 2018, Chasseneuil-du-Poitou, France, October 10-12, 2018*, pages 114–124. ACM, 2018. URL: `https://doi.org/10.1145/3273905.3273910`, `doi:10.1145/3273905.3273910`.

31 Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Evaluation*, 2(4):237–250, 1982. URL: `https://doi.org/10.1016/0166-5316(82)90024-4`, `doi:10.1016/0166-5316(82)90024-4`.

32 Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019. URL: `https://doi.org/10.1145/3323212`, `doi:10.1145/3323212`.

33 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In Cristian Constantinescu and Miguel P. Correia, editors, *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 132–143. IEEE Computer Society, 2012. URL: `https://doi.org/10.1109/EDCC.2012.27`, `doi:10.1109/EDCC.2012.27`.

34 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Robert I. Davis, and Mateo Valero. Iaˆ3: An interference aware allocation algorithm for multicore hard real-time systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 280–290. IEEE Computer Society, 2011. URL: `https://doi.org/10.1109/RTAS.2011.34`, `doi:10.1109/RTAS.2011.34`.

35 Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In Giovanni De Micheli, Bashir M. Al-Hashimi, Wolfgang Müller, and Enrico Macii, editors, *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, pages 741–746. IEEE Computer Society, 2010. URL: `https://doi.org/10.1109/DATE.2010.5456952`, `doi:10.1109/DATE.2010.5456952`.

36 Petar Radojkovic, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, 2012. URL: `https://doi.org/10.1145/2086696.2086713`, `doi:10.1145/2086696.2086713`.

37 Rapita Systems. Multicore timing analysis for do-178c. `https://www.rapitasystems.com/downloads/multicore-timing-analysis-do-178c`.

38 Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I. Davis, and Sebastian Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor. In Alain Plantec, Frank Singhoff, Sébastien Faucou, and Luís Miguel Pinho, editors, *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 67–76. ACM, 2016. URL: `https://doi.org/10.1145/2997465.2997472`, `doi:10.1145/2997465.2997472`.

39 Simon Schliecker and Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, 2010. URL: `https://doi.org/10.1145/1880050.1880058`, `doi:10.1145/1880050.1880058`.

40 Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In Sachin S. Sapatnekar, editor, *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 332–337. ACM, 2010. URL: `https://doi.org/10.1145/1837274.1837359`, `doi:10.1145/1837274.1837359`.

41 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 161–172. IEEE Computer Society, 2016. URL: `https://doi.org/10.1109/RTAS.2016.7461361`, `doi:10.1109/RTAS.2016.7461361`.

42 Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 184–195. IEEE Computer Society, 2015. URL: `https://doi.org/10.1109/ECRTS.2015.24`, `doi:10.1109/ECRTS.2015.24`.