
Article:
Davis, Robert Ian orcid.org/0000-0002-5772-0928, Griffin, David Jack orcid.org/0000-0002-4077-0005 and Bate, Iain John orcid.org/0000-0003-2415-8219 (2022) A Framework for Multi-core Schedulability Analysis Accounting for Resource Stress and Sensitivity. Real-Time Systems. ISSN 1573-1383

https://doi.org/10.1007/s11241-022-09377-8

Real-Time Syst manuscript No.
(will be inserted by the editor)

A Framework for Multi-core Schedulability Analysis Accounting for Resource Stress and Sensitivity

Robert I. Davis · David Griffin · Iain Bate

Received: May 2021 / Revised January 2022 / Accepted: February 2022

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, along with complexity results, and proofs of optimal priority assignment policies. The MRSS task model is underpinned by a proof-of-concept industrial case study. The problem of task allocation is considered in the context of the MRSS task model, with Simulated Annealing shown to provide an effective solution.

Robert I. Davis
Dept. of Computer Science
University of York
United Kingdom
E-mail: rob.davis@york.ac.uk

David Griffin
Dept. of Computer Science
University of York
United Kingdom
E-mail: david.griffin@york.ac.uk

Iain Bate
Dept. of Computer Science
University of York
United Kingdom
E-mail: iain.bate@york.ac.uk

Extended Version

This paper builds upon and extends the ECRTS 2021 paper *Schedulability Analysis for Multi-core Systems Accounting for Resource Stress and Sensitivity* (Davis et al, 2021). Section 3.5 derives complexity results for the various schedulability tests. Section 7 considers the issues involved in allocating tasks to cores in a way that optimizes system schedulability and robustness under the MRSS task model. The difficulties of task assignment are highlighted via a worked example, based on data from the industrial case study. Simulated Annealing is then proposed as a potential solution, and its effectiveness demonstrated via experimental evaluation.

1 Introduction

1.1 Background

The survey published by Akesson et al (2020, 2021), 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 (2019) 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 (Nowotsch and Paulitsch, 2012). A potential solution to this problem, which is being taken up commercially (Rapita Systems, 2019), is to employ a more nuanced measurement-based approach using *micro-benchmarks* (Radojkovic et al, 2012; Fernández et al, 2012; Nowotsch and Paulitsch, 2012; Iorga et al, 2020). 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 shared hardware resource.

The timing verification of single-core systems has traditionally been solved via a *two-step* approach (Maiza et al, 2019). 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 (Kim et al, 2017). Rather, research (Altmeyer et al, 2015; Davis et al, 2018) 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

[1] About 66% of the industry practitioners surveyed by Akesson et al (2020, 2021) used some form of measurement-based timing analysis, whereas only about 33% used some form of static timing analysis.

stress. The MRSS task model provides a simple interface and a separation of concerns between timing analysis and schedulability analysis, thus retaining 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 a 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) (Hassan, 2018) when interleaved accesses target different rows, resulting in additional row close and row open operations that increase 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. Further, it cannot support *dependent interference* where contention can change the information stored in a resource in such a way that it needs to be obtained from elsewhere, potentially creating additional (dependent) interference via another shared resource. Problems of cache thrashing (Radojkovic et al, 2012), cache coherence (Fuchsen, 2010), and cache miss status handling registers (Valsan et al, 2016) can all cause unbounded and/or dependent interference. These issues 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

[2] The most commonly used real-time scheduling polices in industry practice (Akesson et al, 2020, 2021).

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. The section ends by deriving the dominance relations between the schedulability tests, and assessing their complexity.

In systems that use fixed priority scheduling, appropriate priority assignment is a crucial aspect of achieving a schedulable system (Davis et al, 2016). Section 4 investigates optimal priority assignment, proving that Deadline Monotonic (Leung and Whitehead, 1982) 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 (Audsley, 2001) 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 (Audsley, 2001).

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 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 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, but rather to obtain proof-of-concept data to act as an exemplar underpinning the MRSS task model and its analysis.

Section 7 considers the issues involved in allocating tasks to cores in a way that optimizes system schedulability and robustness under the MRSS task model. The difficulties of task assignment are highlighted via a worked example, based on data from the industrial case study. This example shows that overall interference can typically be reduced by partitioning tasks such that those with high resource stress and sensitivity are assigned to a subset of the available cores, while those with low resource stress and sensitivity are assigned to the remaining cores. However, minimizing interference does not necessarily optimize schedulability and robustness. Simulated Annealing is proposed as a potential solution, and its effectiveness demonstrated via experimental evaluation.

Section 8 concludes with a summary and directions for future work.

1.3 Related work

Prior publications that relate to the research presented in this paper include work on micro-benchmarks (Radojkovic et al, 2012; Fernández et al, 2012; Nowotsch and Paulitsch, 2012; Iorga et al, 2020; Rapita Systems, 2019) 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 by Maiza et al (2019). 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 and Ernst (2010) 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 (2010), Pellizzoni et al (2010), Giannopoulou et al (2012), and Lampka et al (2014) 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 (2011) 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 (2016) and Yun et al (2015) 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 (2015); Davis et al (2018) 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 (2016) built on this framework, using it to analyze complex bus arbitration policies on a many-core processor.

Huang et al (2016) and Cheng et al (2017) 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 (2015).

Paolieri et al (2011) 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 (2018) 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. $D_i \leq T_i$.

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 either measurement-based or static timing analysis techniques.

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 (Iorga et al, 2020).

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 (Radojkovic et al, 2012) 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 (Altmeyer et al, 2014, 2016), 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 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) (Valsan et al, 2016). 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 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 (Valsan et al, 2016) and hence 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.

First, we give a simple example. Consider four tasks executing on two cores under partitioned fixed priority preemptive scheduling, with all four tasks accessing the same shared hardware resource r. Tasks τ_1 and τ_2 execute on core 1 and tasks τ_3 and τ_4 execute on core 2. The stand-alone worst-case execution times C_i of the tasks are 100, 200, 150, and 150, their resource sensitivity values X_i^r are 16, 12, 10, and 10, and their resource stress values Y_i^r are 24, 12, 10, and 5 respectively. Further, the periods and deadlines of the tasks are much larger than their execution times. Considering the higher priority task τ_1 on core 1. During the response time of a single job of task τ_1, it could be subject to interference due to cross-core contention from one job of each of tasks τ_3 and τ_4 executing on core 2. This interference is bounded by the minimum of the resource sensitivity of the job of task τ_1, $X_1^r = 16$, and the total resource stress due to one job of each of tasks τ_3 and τ_4, $Y_3^r + Y_4^r = 10 + 5 = 15$. Hence the worst-case response time of task τ_1 is bounded by $R_1 = 100 + \min(16, 15) = 115$. Considering the lower priority task τ_2 on core 1. During the response time of a single job of task τ_2, one job of τ_1 and one job of τ_2 can execute on core 1.

These jobs could be subject to interference due to cross-core contention from one job of each of tasks τ_3 and τ_4 executing on core 2. This interference is bounded by the minimum of the total resource sensitivity of the jobs of tasks τ_1 and τ_2, $X_1^r + X_2^r = 16 + 12 = 28$, and the total resource stress due to the jobs of tasks τ_3 and τ_4, $Y_3^r + Y_4^r = 10 + 5 = 15$. Hence the worst-case response time of task τ_2 is bounded by $R_2 = 100 + 200 + \min(28, 15) = 315$. Similar analysis for tasks τ_3 and τ_4 on core 2 yields bounds on their worst-case response times of $R_3 = 150 + \min(10, 36) = 160$ and $R_4 = 150 + 150 + \min(20, 36) = 320$. Note that this instructive example, and the detailed schedulability analysis given below, makes no assumptions about exactly when jobs execute within the response times considered, nor any assumptions about when within those time intervals cross-core resource contention can actually occur. Rather bounds on worst-case response times are derived using only the task timing parameters: stand-alone worst-case execution times, resource stress and sensitivity values, periods and deadlines.

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 (Joseph and Pandya, 1986; Audsley et al, 1993):

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

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 $R_i \leq D_i$ 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 (Davis et al, 2007):

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

Here, we have reformulated the sufficient analysis for FPNS (Davis et al, 2007) 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:

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

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 (Yun et al, 2013) 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. Stated otherwise, the budget function $F_i^r(t, y)$ becomes a requirement that any set of tasks assigned to core y must guarantee not to exceed. This guarantee is relied upon by the schedulability analysis for tasks executing on the other cores. Hence, the analysis is partially composable, the tasks on core y may be changed or modified provided that the rely-guarantee is respected.

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 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 same sets of tasks, and the $m+1$ core system has extra tasks that execute 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***

3.5 Complexity

The standard response time analysis (Joseph and Pandya, 1986; Audsley et al, 1993) for FPPS on a single-core processor, given by (1), has pseudo-polynomial complexity of $O(n^2 D^{max})$, where n is the number of tasks and D^{max} is the longest deadline of any task in the system. This can be seen by observing that there are n tasks for which response times need to be determined, and on each fixed-point iteration there is a summation over at most n tasks. Further, on each fixed-point iteration the response time can either increase by at least 1, or remain the same, in which case iteration terminates. Since iteration also terminates when the deadline is exceeded, the maximum number of iterations is bounded by D^{max}. Hence the overall complexity of the test is $O(n^2 D^{max})$. The sufficient response time test for FPNS (Davis et al, 2007) similarly has pseudo-polynomial complexity of $O(n^2 D^{max})$. Considering partitioned scheduling on multi-core systems, with m cores, at most n tasks per core, and no cross-core contention or interference, these tests have $O(mn^2 D^{max})$ complexity.

The schedulability tests for the MRSS task model are derived from the above tests; however, they also consider cross-core contention and interference over $|H|$ shared hardware resources.

The **CpFPPS-m-fc** and **CpFPNS-m-fc** tests have pseudo-polynomial complexity of $O(m|H|n^2 D^{max})$. This can be seen by observing that there are at most mn tasks for which response times need to be determined, and on each fixed-point iteration of (2) or (8) the interference term involves a nested summation over $|H|$ resources, no summation over m cores – see (10), and lastly summation over n tasks within the expression for $S_i^r(R_i, x)$. Finally, the maximum number of fixed point iterations is again bounded by D^{max}.

The **CpFPPS-m-D** and **CpFPNS-m-D** tests have pseudo-polynomial complexity of $O(m^2|H|n^2D^{max})$. This can be seen by observing that there are at most mn tasks for which response times need to be determined, and on each fixed-point iteration of (2) or (8) the interference term involves a nested summation over $|H|$ resources, m cores, and lastly over n tasks within the expressions for $E_i^r(R_i, y)$ and $S_i^r(R_i, x)$. Finally, the maximum number of fixed point iterations is again bounded by D^{max}.

The **CpFPPS-m-R** and **CpFPNS-m-R** tests were described in Sections 3.1 and 3.2 as requiring nested fixed point iterations to compute the interdependent response times. The efficiency of these tests can however be improved by considering the monotonicity of the expressions on the right hand side of each of the equations with respect to the values of both R_i and R_j. This means that the tests can be implemented via an outer loop that performs fixed point iteration, combined with a simple inner loop that iterates over all of the tasks in the system. Below, we describe this implementation in more detail, followed by the complexity of the **CpFPPS-m-R** and **CpFPNS-m-R** tests.

In an efficient implementation of the **CpFPPS-m-R** test (resp. **CpFPNS-m-R** test), the outer loop iteration starts with $R_i = C_i$, $R_j = C_j$ etc. for all tasks. The inner loop iterates over all tasks, for each task it computes an updated response time by evaluating (2) (resp. (8)) just once using the R_i and R_j values from the previous outer loop iteration. Due to the right hand sides of (2), (3), (5), and (6) (resp. (8), (9), (5), and (6)) being monotonically non-decreasing functions of R_i and R_j, then on each outer loop iteration, the response time value for each task can only increase or remain the same, it cannot decrease. Hence, the outer loop fixed point iteration is guaranteed to terminate, either due to convergence (i.e. all response times are unchanged from the previous iteration) indicating a schedulable system, or because the response time of at least one task has exceeded its deadline.

Observe that on each iteration of the outer loop, the response time of at least one task must increase by at least 1, otherwise the response times have converged and the test terminates. The maximum number of outer loop iterations is therefore upper bounded by mnD^{max}. The inner loop evaluates (2) (resp. (8)) once for each of at most mn tasks, with each such evaluation requiring $O(m|H|n)$ operations. It follows that the **CpFPPS-m-R** test and the **CpFPNS-m-R** test have pseudo-polynomial complexity of $O(m^3|H|n^3D^{max})$.

Comparing the complexity of the tests for the MRSS task model to those for partitioned fixed priority scheduling with no contention, we observe that: (i) the complexity of the **CpFPPS-m-fc** and **CpFPNS-m-fc** tests is higher by a factor of $|H|$, (ii) the complexity of the **CpFPPS-m-D** and **CpFPNS-m-D** tests is higher by a factor of $m|H|$, and (iii) the complexity of the **CpFPPS-m-R** and **CpFPNS-m-R** tests is higher by a factor of $m^2|H|n$. Given the high performance of the standard response time tests for fixed priority scheduling (Davis et al, 2008), in practice, all of the tests for the MRSS task model scale well to realistic system sizes.

4 Priority Assignment

To maximize schedulability it is necessary to assign task priorities in an optimal way (Davis et al, 2016). This section considers optimal priority assignment for the schedulability tests introduced in Section 3.

4.1 pFPPS Priority Assignment

Leung and Whitehead (1982) 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 the review by Davis et al (2016). 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 tasks 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 (2011) proved 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 (Audsley, 2001) 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 (Audsley, 2001), 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 by Davis and Burns (2011) 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 $E_j^r(R_i, y)$ 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 $E_k^r(R_j, x)$ 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 by Davis and Burns (2010), could potentially 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 (1996) 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 (Audsley, 2001) 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 $(m-1)X_k^r$. Audsley's algorithm (Audsley, 2001) is also optimal with respect to the **CpFPNS-m-D** test, as proved below.

Theorem 3 *Audsley's algorithm (Audsley, 2001) 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 by Davis and Burns (2011) 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 (Audsley, 2001), 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, by replacing the words "**CpFPPS-m-R** test" with the words "**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:

[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 utilizations ($U_i = C_i/T_i$) were generated using the Dirichlet-Rescale (DRS) algorithm (Griffin et al, 2020b) (open source Python software (Griffin et al, 2020a)) 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 (Emberson et al, 2010) 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).
- 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 proof of concept industry case study described in Section 6. 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 (Bastoni et al, 2010)). 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 (Joseph and Pandya, 1986; Audsley et al, 1993) 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 (Davis et al, 2007) 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) (Leung and Whitehead, 1982) 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 (Radojkovic et al, 2012; Fernández et al, 2012; Nowotsch and Paulitsch, 2012; Iorga et al, 2020).

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 1, and for the pFPNS schedulability tests in Figure 2. 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 (Bastoni et al, 2010) 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 3 for pFPPS, and Figure 4 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 5 for pFPPS, and Figure 6 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 effectively 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 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).

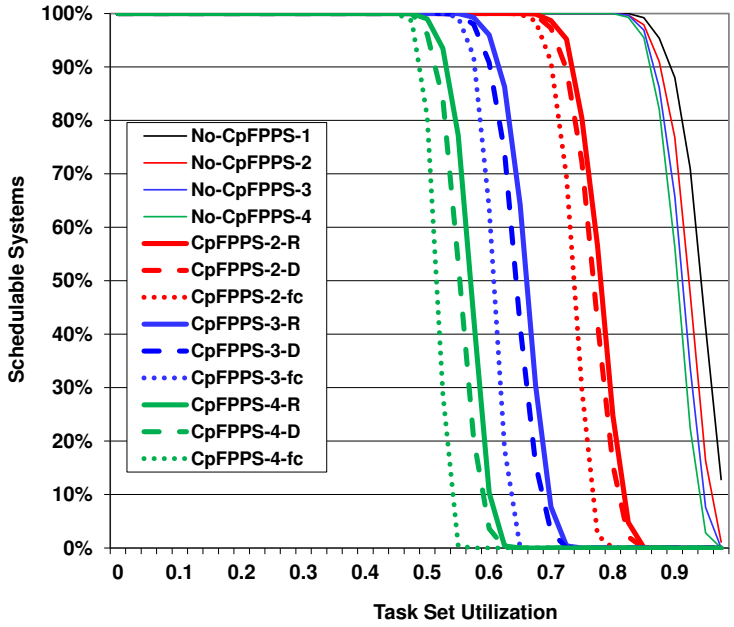


Fig. 1 pFPPS: Success Ratio: Varying task set utilization

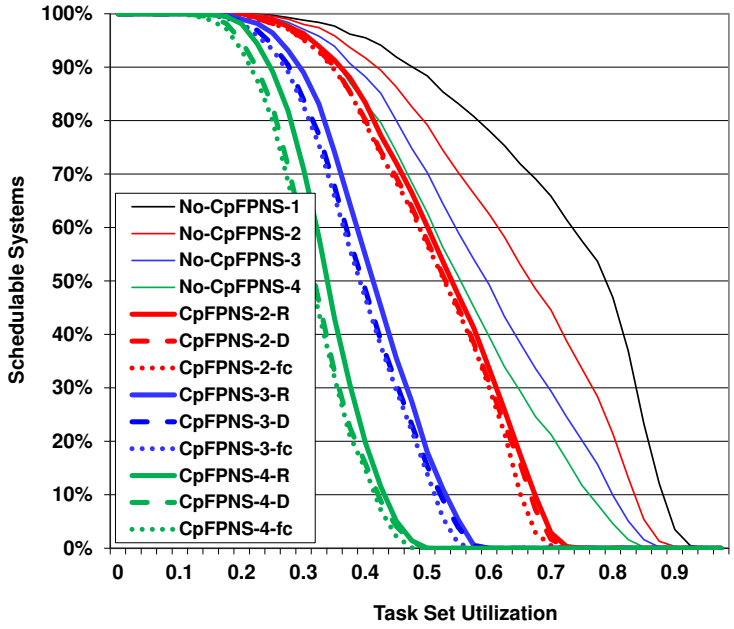


Fig. 2 pFPNS: Success Ratio: Varying task set utilization

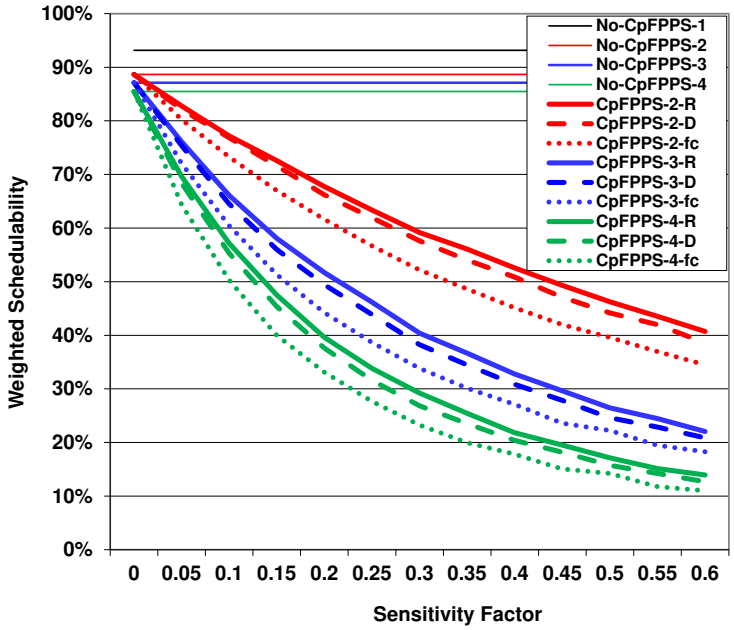


Fig. 3 pFPPS:Weighted Schedulability: Varying Sensitivity Factor (SF)

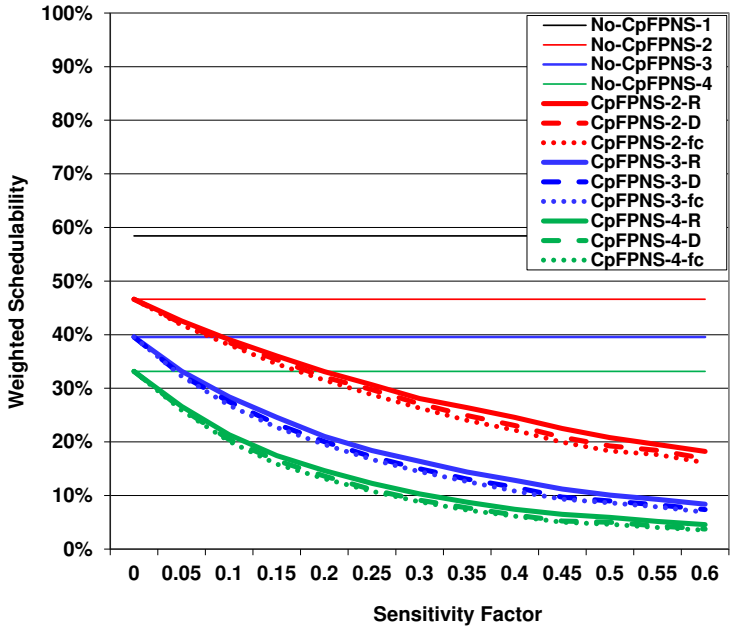


Fig. 4 pFPNS: Weighted Schedulability: Varying Sensitivity Factor (SF)

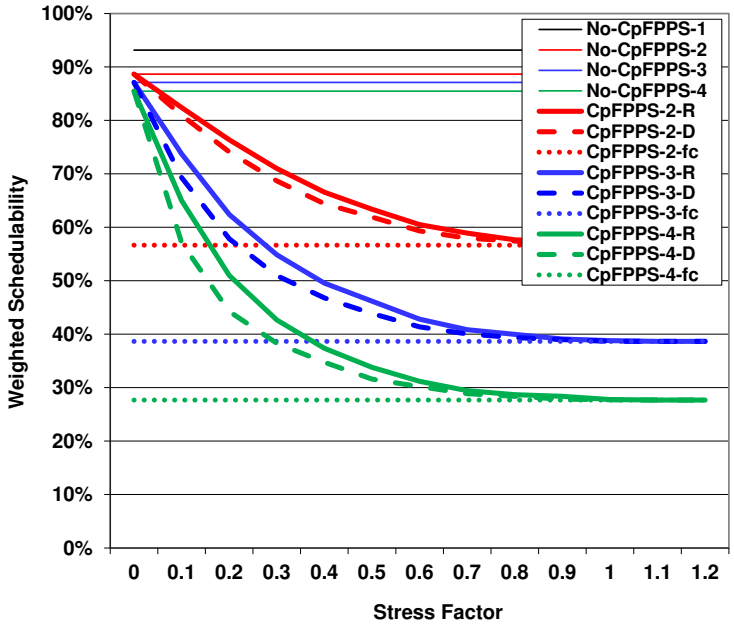


Fig. 5 pFPPS: Weighted Schedulability: Varying Stress Factor (RF)

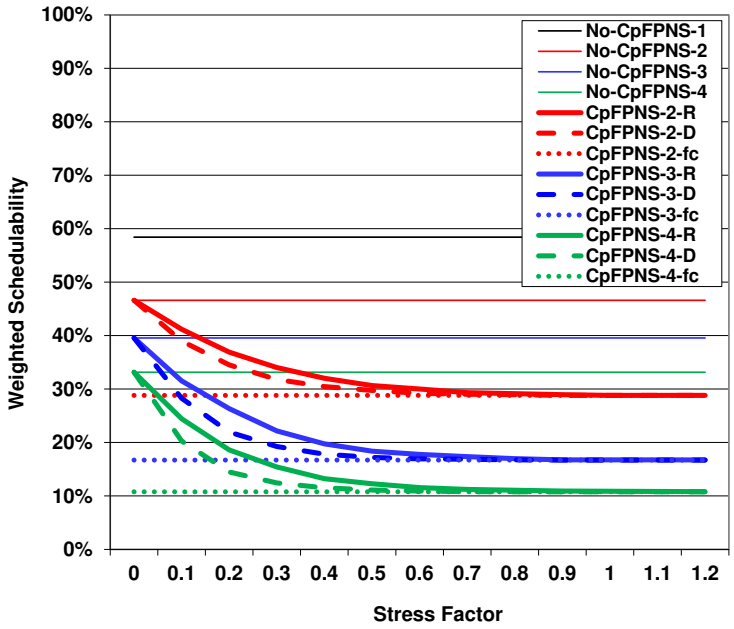


Fig. 6 pFPNS: Weighted Schedulability: Varying Stress Factor (RF)

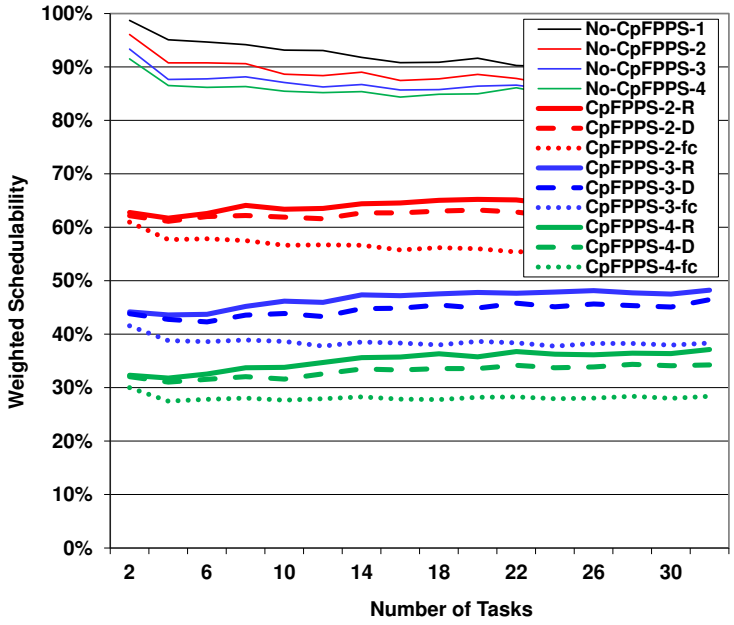


Fig. 7 pFPPS: Weighted Schedulability: Varying number of tasks in each task set

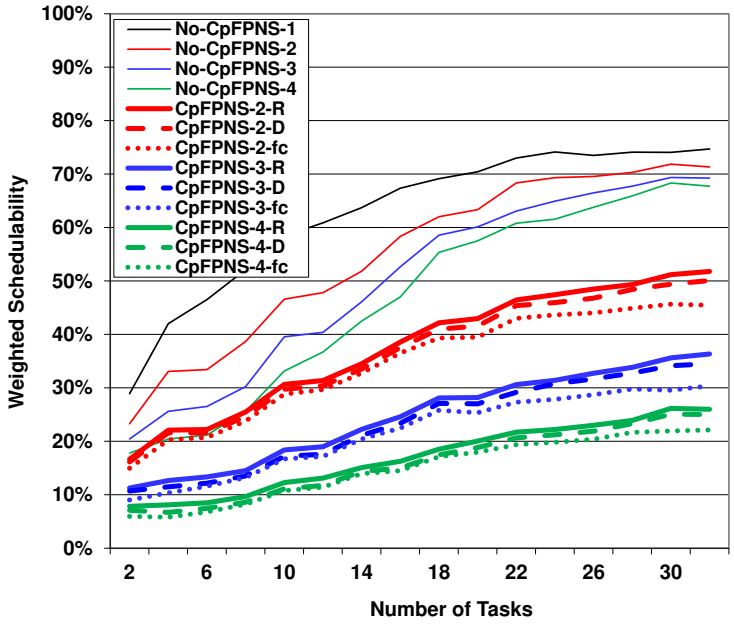


Fig. 8 pFPNS: Weighted Schedulability: Varying number of tasks in each task set

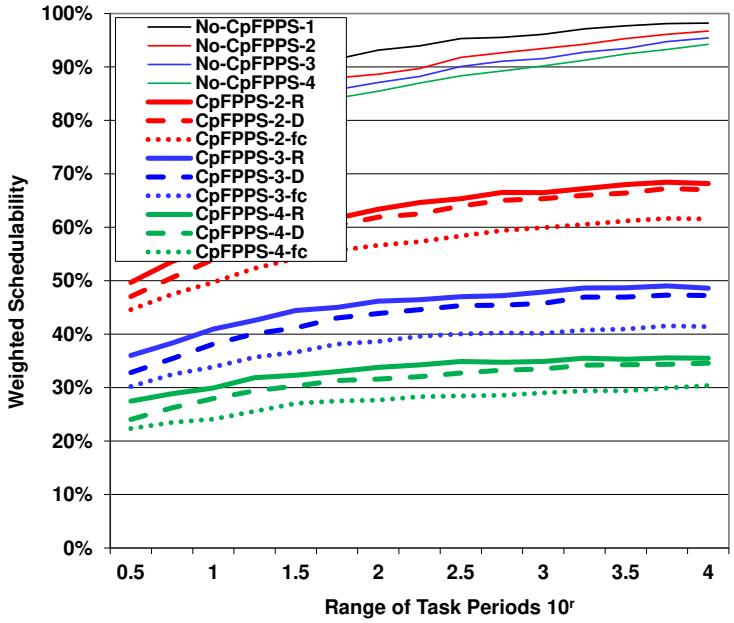


Fig. 9 pFPPS: Weighted Schedulability: Varying range of task periods

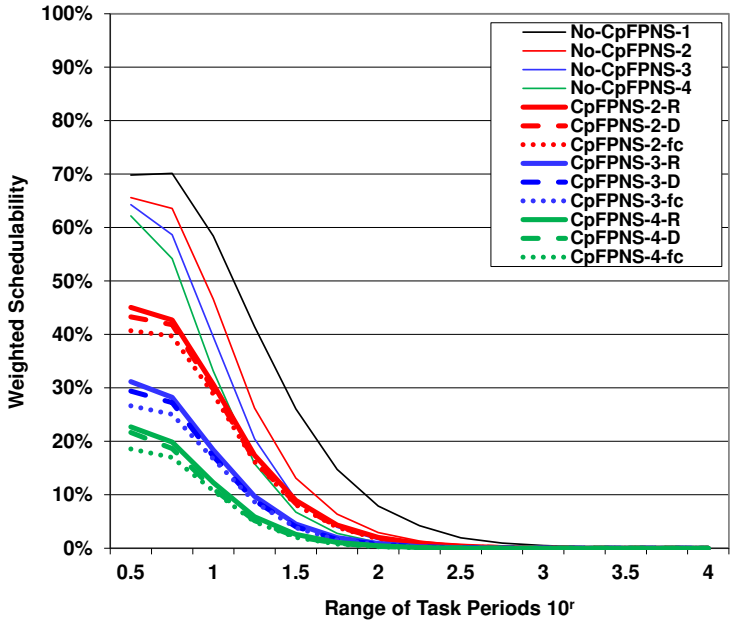


Fig. 10 pFPNS: Weighted Schedulability: Varying range of task periods

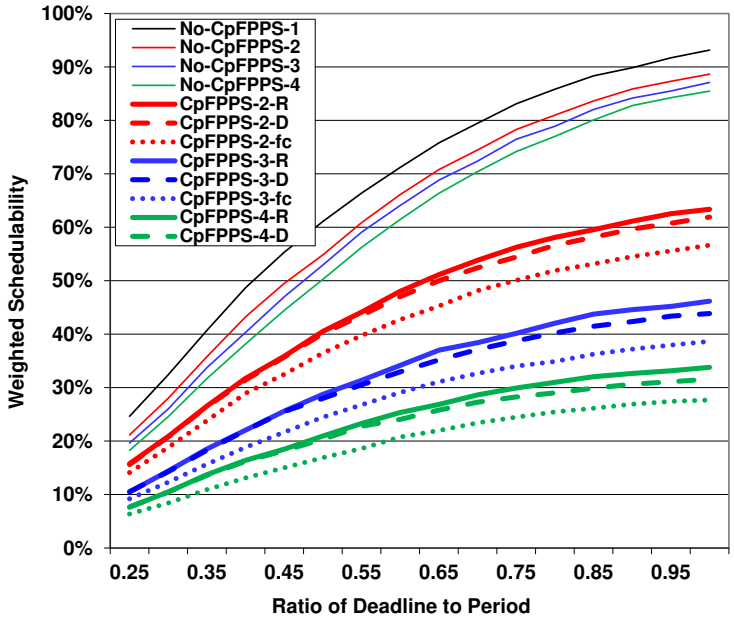


Fig. 11 pFPPS: Weighted Schedulability: Varying ratio of deadlines to periods

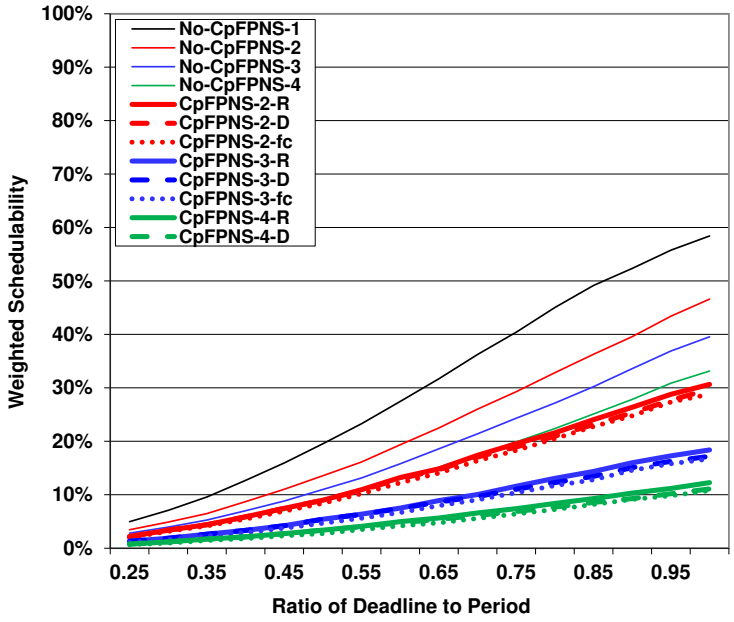


Fig. 12 pFPNS: Weighted Schedulability: Varying ratio of deadlines to periods

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 steps of 2 are shown in Figure 7 for pFPPS, and Figure 8 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 8), 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 7) 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 7 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 9 for pFPPS, and Figure 10 for 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 11 for pFPPS, and Figure 12 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 Case Study

In this section, 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 by Law and Bate (2016) and Lesage et al (2018). 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+ (Lesage et al, 2018), along with a framework that facilitates taking timing measurements (Bate et al, 2020).

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).

6.1 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

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

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 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.

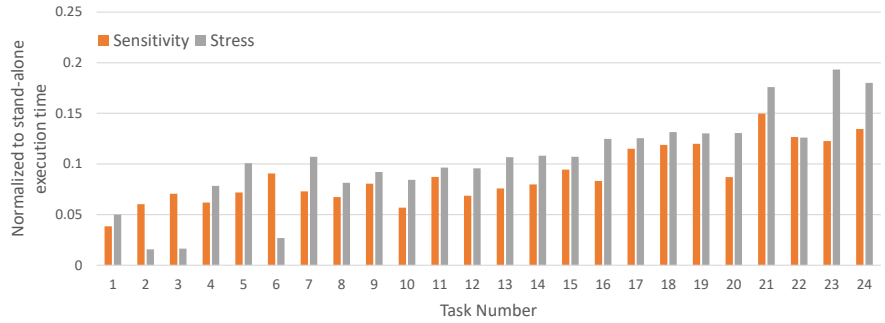


Fig. 13 Estimated resource stress and resource sensitivity values for 24 tasks from a Rolls-Royce aero-engine control systems normalized to the task's estimated stand-alone WCET

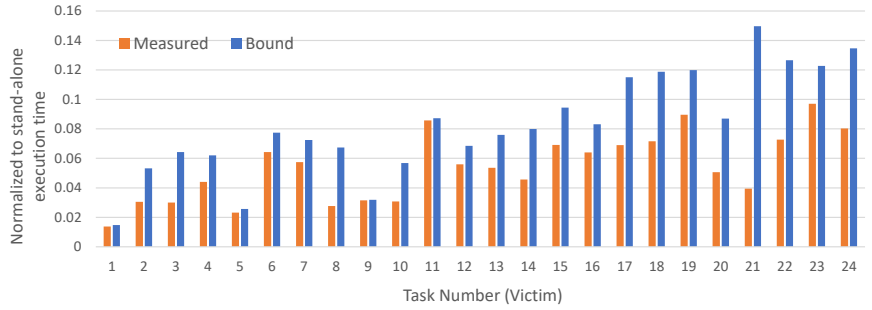


Fig. 14 Increase in execution time of a (victim) task co-running with its paired task. Maximum observed value and computed bound derived from resource sensitivity and resource stress values, normalized to the stand-alone execution time of the victim task.

Figure 13 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 14 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.

7 Task Allocation

Task allocation for partitioned scheduling on a multi-core processor is an NP-hard problem that corresponds to bin-packing (Garey and Johnson, 1979). Practical approaches to task allocation can therefore be divided into two main categories: (i) heuristic methods (Oh and Son, 1995; López et al, 2004; Fisher et al, 2006), and (ii) search-based techniques such as Simulated Annealing (Kirkpatrick et al, 1983; Tindell et al, 1992; Natale and Stankovic, 1995), Genetic Algorithms (Monnier et al, 1998; Oh and Wu, 2004), and Particle Swarm Optimization (Kennedy and Eberhart, 1995; Salman et al, 2002).

The heuristic methods operate according to a *greedy* algorithm. Each heuristic consists of two policies. The first policy determines the order of cores for trial allocation of a selected task to a core. For example, First-Fit selects cores in index order, Best-Fit selects cores in order of their remaining capacity[5], smallest first, and Worst-Fit selects cores in order of their remaining capacity, largest first. The second policy dictates how the tasks are ordered for allocation to cores, for example by Decreasing Utilization, Decreasing Density etc. The greedy algorithm iterates through the tasks once in the predetermined order (e.g. Decreasing Utilization). For each task, a trial allocation is checked for each core in turn in the order prescribed

[5] Capacity is usually taken to mean utilization, but other measures are also possible.

(e.g. First-Fit). If the trial allocation forms a schedulable system along with the previously allocated tasks, then the allocation for that task is confirmed and the process moves on to the next task, otherwise the trial allocation of the task moves on to the next core. If all cores have been tried for a given task and none of the trial allocations succeeded, then the overall allocation fails; there is no back-tracking. Note, there is an assumption implicit in the way in which the greedy algorithm directs the allocation. This assumption is that the schedulability test used to analyse the tasks allocated to one core does not depend on the sets of tasks allocated to other cores.

The context-independent schedulability tests (**CpFPPS-m-fc** and **CpFPNS-m-fc** derived in Section 3.3) for the MRSS task model do not depend on the sets of tasks on other cores. These tests reduce to the standard response time analysis for fixed priority preemptive and non-preemptive scheduling with the task execution times increased to take account of the maximum interference due to contention (i.e. C_k is increased by $\sum_{r \in H}(m-1)X_k^r$). Hence, when these tests are used, prior heuristic approaches to task allocation developed for partitioned fixed priority scheduling are directly applicable.

The context-dependent schedulability tests (i.e. the **CpFPPS-m-R** and **CpFPPS-m-D** tests derived in Section 3.1 for pFPPS, and the **CpFPNS-m-R** and **CpFPNS-m-D** tests derived in Section 3.2 for pFPNS) depend on the sets of tasks on the other cores. This challenges the use of simple heuristic approaches to task allocation, since the performance of the greedy algorithm breaks down when the assumption of no dependence is broken.

For example, consider a four core system where, according to the schedulability test used, allocating tasks to the second and subsequent cores impinges on the schedulability of the tasks previously allocated to the first core. In this case, the First-Fit and Best-Fit heuristics will allocate tasks to the first core, making use of nearly all of its capacity, then, as trial allocations proceed to the second core, the tasks on the first core will likely become unschedulable. This happens due to account being taken in the schedulability test of the additional cross-core contention and interference emanating from the second core, and hence the overall allocation will fail. The Worst-Fit policy may perform somewhat better in this respect; however, the allocation will still be misdirected, as an unchanging set of tasks on one core can become unschedulable due to the allocation of tasks to another core. In general, a partial allocation of tasks to one core can be obtained that is not in itself viable under any possible allocation of the remaining tasks to the other cores. Note, prior research has shown that Worst-Fit performs poorly for the task allocation problem, with López et al (2004) showing that Worst-Fit achieves the lowest overall utilization bound for any reasonable[6] greedy allocation algorithm.

[6] A *reasonable* allocation algorithm is one that only fails once there is no core on which a task can be successful allocated.

Further, the allocation difficulties caused by dependencies on the set of tasks on other cores cannot be solved by assuming that all tasks that remain unallocated will be allocated to some other core where they can cause contention and interference, since this would reduce performance to that obtained with the context-independent schedulability tests. For these reasons, a greedy approach to allocation is not viable with the context-dependent schedulability tests. More general search-based optimization techniques such as Simulated Annealing, Genetic Algorithms, Particle Swarm Optimization are required instead[7].

7.1 Guidelines for Allocation

Notwithstanding the difficulties in applying existing heuristics, it is possible to deduce some guidelines or rules-of-thumb for task allocation via thought experiments. Assume that the task set to be allocated is made up of two subsets of tasks V and W. Subset V comprises tasks that access shared hardware resources and hence potentially suffer cross-core contention and interference, while subset W comprises tasks that do not access shared hardware resources. Recall that the interference term $I_i^r(R_i)$ in the context-dependent schedulability tests 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 the task τ_i under analysis, within its response time R_i; and (ii) the total resource stresses 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 a time interval of length R_i. The interference term is given by (5), repeated below for convenience.

$$I_i^r(R_i) = \sum_{\forall y \neq x} \min(E_i^r(R_i, y), S_i^r(R_i, x)) \tag{12}$$

The $min(\ldots)$ function in (12) implies that the interference considered due to contention can be reduced by allocations that unbalance the values of $E_i^r(R_i, y)$ and $S_i^r(R_i, x)$. For example, allocating only tasks from subset W to some core y reduces $E_i^r(R_i, y)$ to zero, and hence the contribution to $I_i^r(R_i)$ from that core to zero. Further, allocating as many tasks as possible from subset V with high resource sensitivity and high resource stress to the same core x will increase the total resource sensitivity $S_i^r(R_i, x)$ for that core, and reduce the total resource stress for other cores, decreasing the value returned by the $min(\ldots)$ function. A potentially useful guideline for task allocation is therefore to aim for an allocation where tasks from the subset V are packed into a small number of cores. In particular, it is useful to place those tasks with high resource sensitivity and high resource stress (as compared to their execution times) together on the same core. Taken to extremes, allocating all of the tasks in

[7] Alternative techniques such as Mixed-Integer Linear Programming (MILP) may also be viable, but are not explored further here.

subset V to one core, and all of the tasks in subset W to other cores would eliminate all cross-core contention and interference.

We illustrate the possibilities and difficulties of task allocation using a practical example based on data for 6 tasks (listed in Table 1) from the industrial case study described in Section 6. Note all times are given in nanoseconds.

Table 1 Case study task parameters in nanoseconds.

Task ID	Stand-alone	Sensitivity	Stress
1	224844	8646	11250
2	211406	12760	3308
4	127709	7917	10000
5	126927	9114	12787
6	122448	11094	3281
7	116511	8489	12475

For the purposes of this example, we assume a system with two cores, and that all tasks are released at the same time, and must complete within a deadline of 0.5ms (i.e. 500000ns). The periods of all tasks are assumed to be much longer than this deadline, and thus the total resource sensitivity $S_i^r(R_i, x)$ and total resource stress $E_i^r(R_i, y)$ over the longest task response times can be found by simply summing the resource sensitivity and resource stress values for the tasks allocated to each of the cores. Table 2 gives these values, along with the sum of the stand-alone execution times for the tasks allocated to each core, for 7 possible allocations labelled A to G. These allocations are the only plausible ones, i.e the only ones that could meet the deadlines even if cross-core contention and interference were ignored.

Table 2 Basic parameters for various allocations.

Alloc.	Core 1				Core 2			
	Tasks	Stand -alone	Sensitivity	Stress	Tasks	Stand -alone	Sensitivity	Stress
A	1,2	436250	21406	14558	4,5,6,7	493595	36614	38543
B	1,4,5	479480	25677	34037	2,6,7	450365	32343	19064
C	1,4,6	475001	27657	24531	2,5,7	454844	30363	28570
D	1,4,7	469064	25052	33725	2,5,6	460781	32968	19376
E	1,5,6	474219	28854	27318	2,4,7	455626	29166	25783
F	1,5,7	468282	26249	36512	2,4,6	461563	31771	16589
G	1,6,7	463803	28229	27006	2,4,5	466042	29791	26095

Table 3 gives the results for the 7 different allocations, including the total interference that the tasks on each core suffer as a result of cross-core contention, the total execution time of the tasks allocated to each core including interference (i.e. the response time of the lowest priority task), and the sum of the interference on both cores. The results in Table 3 assume a context-dependent analysis, using both resource sensitivity and resource

Table 3 Computed results for various allocations (context-dependent analysis).

Alloc.	Core 1			Core 2			Both Cores
	Tasks	Interference	Total ET	Tasks	Interference	Total ET	Interference
A	1,2	21406	457656	4,5,6,7	14558	508153	35964
B	1,4,5	19064	498544	2,6,7	32343	482708	51407
C	1,4,6	27657	502658	2,5,7	24531	479375	52188
D	1,4,7	19376	488440	2,5,6	32968	493749	52344
E	1,5,6	25783	500002	2,4,7	27318	482944	53101
F	1,5,7	16589	484871	2,4,6	31771	493334	48360
G	1,6,7	26095	**489898**	2,4,5	27006	**493048**	53101

stress values. Allocation A is not schedulable despite minimizing the total interference over both cores by keeping the resource sensitivity and resource stress of the tasks on Core 1 low (see Table 2). This is because once interference is taken into account the load on Core 2 becomes too large (508153ns). Similarly, allocations C and E are also unschedulable, due to too high a load on Core 1 once interference is taken into account. The most promising allocations are F and G. Allocation F provides the lowest sum of interference over both cores (48360ns) of all the schedulable allocations, by minimizing the resource stress due to the tasks on Core 2 (see Table 2). Allocation G has a substantially higher sum of interference over both cores (53101ns), but balances the task load across the two cores better, maximizing the headroom for any overruns, and is therefore arguably the most robust task allocation. This simple example illustrates the difficulties of the task allocation problem and the requirement, typical of real systems, to consider multiple criteria such as both schedulability and robustness (Davis and Burns, 2007).

Table 4 provides directly comparable results to Table 3, but this time using context-independent analysis. Here, only resource sensitivity values are used, and contention from the other core is not bounded by the resource stress values. This means that the total interference for both cores is constant at 58020ns, almost 10000ns higher than allocation F in Table 3. Again, allocation G is the most robust, allowing the greatest headroom for overruns; however, this headroom is reduced by more than 2750ns, compared with the results of context-dependent analysis, shown in Table 3.

Table 4 Computed results for various allocations (context-independent analysis).

Alloc.	Core 1			Core 2			Both cores
	Tasks	Interference	Total ET	Tasks	Interference	Total ET	Interference
A	1,2	21406	457656	4,5,6,7	36614	530209	58020
B	1,4,5	25677	505157	2,6,7	32343	482708	58020
C	1,4,6	27657	502658	2,5,7	30363	485207	58020
D	1,4,7	25052	494116	2,5,6	32968	493749	58020
E	1,5,6	28854	503073	2,4,7	29166	484792	58020
F	1,5,7	26249	494531	2,4,6	31771	493334	58020
G	1,6,7	28229	**492032**	2,4,5	29791	**495833**	58020

The discussion and examples given above highlight the difficulties involved in task allocation for partitioned multi-core systems under the MRSS task model. In particular, when the schedulability of a task allocated to one core is dependent on the tasks allocated to other cores (i.e. context-dependent schedulability tests are used) then the use of a greedy algorithm and commonly applied heuristics are no longer viable. In this case, it is clear that highly effective solutions to the task allocation problem will only be possible via search-based techniques, such as Simulated Annealing or Genetic Algorithms.

7.2 Task Allocation using Simulated Annealing

Given the lack of effective heuristics for task allocation for partitioned multi-core systems when cross-core contention and interference is taken into account, we developed an implementation of Simulated Annealing aimed at solving the problem. Simulated Annealing (Kirkpatrick et al, 1983) is a general-purpose probabilistic search-based technique that can be used to solve optimization problems. Specifically, Simulated Annealing is a meta-heuristic search, which seeks to approximate global optimization within a large search space for a given optimization problem. It is typically used when the search space is discrete; as is the case of task allocation considered here.

Pseudo code for the Simulated Annealing algorithm is shown in Listing 1. Simulated Annealing relies on two key functions, a `Cost_Function` that determines the quality of each possible solution, and a `Modify_Function` that makes a randomly chosen, but valid modification to the current solution, in order to create a new solution that is close to it.

For Simulated Annealing to be effective, it is important that the `Cost_Function` provides a smooth and continuous metric, indicative of solution quality, that can drive the search towards an optimal solution. In the context of task allocation, we use the processor speed scaling factor F (Punnekkat et al, 1997). For a given allocation of tasks to cores, the `Cost_Function` determines the smallest value of F such that the execution times, resource sensitivities, and resource stresses of all tasks can be scaled by a factor of $1/F$ (alternatively, the periods and deadlines can be scaled by a factor of F) and the system just remains schedulable. This metric optimizes both schedulability and robustness, since F takes its smallest value for the task allocation that can tolerate the processor running at the lowest possible speed.

The processor speed scaling factor provides a continuous metric, that is at or below 1.0 for schedulable task allocations, and above that value for unschedulable allocations. The value of F is calculated via a binary search, in conjunction with an appropriate schedulability test. As a starting point, the binary search requires minimum and maximum bounds. These can be determined as follows: (i) the minimum bound is such that the scaled deadline for one of the tasks is reduced to its execution time, (ii) the maximum bound is such that the execution times of all tasks (inflated due to resource sensitivities)

fit within the smallest scaled deadline of any task. Any value of F smaller than the minimum bound is guaranteed to result in an unschedulable system, whereas a value of F equal to the maximum bound is guaranteed to result in a schedulable system, given that the deadlines are constrained ($D_i \leq T_i$).

It is essential that the `Modify_Function` is able to span the search space, otherwise the algorithm may be unable to ever find the optimal solution. In the case of the task allocation problem, it must be possible, via repeated application of the `Modify_Function` to move from any valid task allocation to any other one. Our implementation of the `Modify_Function` makes one of two possible changes to an existing allocation: (i) it selects a task at random and changes its allocated core to a randomly selected different core, (ii) it selects two different tasks at random that are allocated to different cores, and swaps their allocation around[8]. The single task modification is randomly selected 20% of the time, with swapping selected the remaining 80% of the time.

The Simulated Annealing algorithm (see Listing 1) operates via two nested loops. The outer loop (lines 8-24) represents a series of reducing temperatures, used in the choices that the algorithm makes. In the experiments, the initial `temperature` was set to 1.0, and the final `min_temperature` to 0.01. Further, the `cooling_factor` (line 23) was set to 0.95499, which results in 100 iterations of the outer loop. The inner loop (lines 9-22) iterates 50 times at each temperature. Thus the algorithm explores 5000 allocations in all, starting from an initial allocation of tasks to cores. In the experiments, the initial allocation was taken directly from the system generation, with an equal number of tasks, with equal total utilization, assigned to each core.

Simulated Annealing explores the search space by making modifications to an existing allocation via the `Modify_Function` (line 10), and then determining the quality of the new allocation formed via the `Cost_Function` (line 11). If the new allocation is an improvement on the best allocation seen so far then it is saved (lines 12-15). The signature behavior of the algorithm is embodied in lines 16-20. If the new allocation is an improvement on the current one (line 16), then it becomes the current allocation, which the algorithm will continue searching from (lines 18-19). If the new allocation does not represent an improvement (line 17), then there is still a chance that it will be accepted, and hence built upon. The probability of acceptance depends on how much worse the allocation has become, and the current temperature. Initially, when the temperature is high, new allocations can be accepted that are substantially worst than the current allocation. This helps to avoid the search becoming stuck in a local optimum. As the temperature decreases, only smaller negative steps are likely to be accepted, until at very low temperatures, the algorithm effectively behaves like a hill-climbing search, only accepting improved allocations.

[8] In the unlikely event that all tasks are allocated to the same core, then a null swap is performed that does not modify the allocation.

Listing 1 Simulated Annealing.

```
1  // input initial_allocation
2
3  current_allocation = initial_allocation;
4  best_allocation = initial_allocation;
5  last_cost = Cost_Function(current_allocation);
6  best_cost = last_cost;
7
8  do{
9      for(i = 0; i < max_iterations; i++){
10         new_allocation = Modify_Function(current_allocation);
11         new_cost = Cost_Function(new_allocation);
12         if (new_cost < best_cost){
13             best_cost = new_cost;
14             best_allocation = new_allocation;
15         }
16         if ((new_cost < last_cost) ||
17             (Rand() < exp((last_cost - new_cost) / temperature))){
18             last_cost = new_cost;
19             current_allocation = new_allocation;
20         }
21         // otherwise discard new_allocation
22     }
23     temperature *= cooling_factor;
24 } while(temperature) > min_temperature)
25
26 // output best_allocation
```

7.3 Simulated Annealing Experiments

We explored the improvements in task allocation that can be achieved using Simulated Annealing by revisiting the first experiment discussed in Section 5.3 (illustrated by Figures 1 and 2). Recall that in that experiment we compared the performance of the various schedulability tests for partitioned fixed priority preemptive and non-preemptive scheduling taking into account cross-core contention and interference via the *Success Ratio* metric, i.e. the percentage of systems generated that were deemed schedulable. We repeated the experiment, this time comparing the performance of the default initial assignment of tasks to cores, with that obtained via Simulated Annealing starting from the initial assignment. Since Simulated Annealing involves many trial allocations, we reduced the number of systems generated per utilization level from 1000 to 100. This was done to ensure that the overall runtime remained manageable[9]. Task sets were generated with parameters as described in Section 5.1. Each system comprised nm tasks, with a different set of n tasks, with total utilization U, initially allocated to each of the m

[9] The Simulated Annealing algorithm was configured to iterate 5000 times. On each iteration the schedulability test was run approximately 10 times to determine the processor speed scaling factor via binary search. Hence, to analyse 100 systems requires approximately 5000000 schedulability tests.

cores. By default $n = 10$, hence each 2 core system had 20 tasks in total, and each 4 core system 40 tasks in total. Deadline Monotonic priority ordering was used throughout, since this was shown, in Section 4.1, to be optimal for the simpler schedulability tests used in the preemptive case, and is also an effective heuristic in the non-preemptive case (Davis et al, 2016).

The Simulated Annealing algorithm started from the initial allocation and was able to re-allocate tasks to different cores in order to improve overall system schedulability. For a system to be schedulable, the task set on each of its cores had to be schedulable accounting for cross-core contention and interference. While the initial allocation comprised task sets of equal utilization on each core, this was not necessarily the case with the final allocation obtained via Simulated Annealing.

We compared the effectiveness of the task allocations generated by Simulated Annealing for three schedulability tests for each of pFPPS and pFPNS:

- **CpFPPS-m-R**: The context-dependent 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 context-dependent deadline based analysis of pFPPS with contention, introduced in Section 3.1, and given by (2), (3), (4), and (5).
- **CpFPPS-m-fc**: The context-independent fully composable analysis of pFPPS with contention, introduced in Section 3.3, and given by (2), (3), and (10).
- **CpFPNS-m-R**: context-dependent 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 context-dependent deadline based analysis of pFPNS with contention, introduced in Section 3.2, and given by (8), (9), (4), and (5).
- **CpFPNS-m-fc**: The context-independent fully composable analysis of pFPNS with contention, introduced in Section 3.3, and given by (8), (9), and (10).

Figure 15 for pFPPS and Figure 16 for pFPNS illustrate the effectiveness of the allocations produced by Simulated Annealing for 2 cores and for 4 cores, respectively. (The results for 3 cores were similar and were omitted to avoid cluttering the graphs). In the preemptive case, the results for Simulated Annealing are labelled **CpFPPS-m-SA-R**, **CpFPPS-m-SA-D**, and **CpFPPS-m-SA-fc** respectively, and are compared to the baseline allocation, labelled **CpFPPS-m-R**, **CpFPPS-m-D**, and **CpFPPS-m-fc**. Similarly, for the non-preemptive case, the results for Simulated Annealing are labelled **CpFPNS-m-SA-R**, **CpFPNS-m-SA-D**, and **CpFPNS-m-SA-fc** respectively, and are compared to the baseline allocation, labelled **CpFPNS-m-R**, **CpFPNS-m-D**, and **CpFPNS-m-fc**.

The different schedulability tests, number of cores, and baseline / Simulated Annealing are coded into the line types and colors as follows: the response time and deadline based context-dependent tests are shown as solid and dashed lines respectively, with the fully composable context-independent test shown as dotted lines. The results for 4 cores are in green for the baseline and in turquoise for Simulated Annealing. Similarly, the results for 2 cores are in red for the baseline and in orange for Simulated Annealing.

Figure 15 for pFPPS and Figure 16 for pFPNS show that Simulated Annealing is able to improve schedulability, compared to the baseline, for each of the schedulability tests and numbers of cores considered. These results are further highlighted in Figure 17 for pFPPS and Figure 18 for pFPNS. These figures show the number of systems that were not schedulable with the baseline allocation, but where Simulated Annealing was able to find a schedulable allocation, as a percentage of all systems considered. Observe that the improvement obtained is much larger for the context-dependent schedulability tests (solid and dashed lines), than it is for the context-independent schedulability tests (dotted lines). This is because the context-dependent schedulability tests take account of both the resource sensitivity of the tasks on the same core as the task under analysis, and the resource stress due to the tasks on other cores. By doing so, these tests provide an opportunity for the Simulated Annealing algorithm to allocate the tasks in a way that reduces the amount of cross-core contention and interference considered. This tends to results in allocations where the resource sensitivity and resource stress is high on some cores and low on others, subject of course to the system remaining schedulable.

Although there are dominance relations between the tests as described in Section 3.4, the random process enacted by Simulated Annealing, and the fact that it is not guaranteed to find the optimal solution means that dominance between the allocations generated is not guaranteed. Thus, in Figures 15 and 16, the results for **CpFPPS-m-SA-D** (e.g. dashed orange lines) can sometimes slightly exceed those for **CpFPPS-m-SA-R** (e.g. solid orange lines) due to statistical variation.

Table 5 Number of additional schedulable systems found using Simulated Annealing for task allocation.

	Test	pFPPS		pFPNS	
2 cores	-fc	145	3.6%	662	16.6%
	-D	422	10.6%	876	21.9%
	-R	380	9.5%	845	21.1%
4 cores	-fc	106	2.7%	624	15.6%
	-D	474	11.9%	946	23.7%
	-R	448	11.2%	912	22.8%

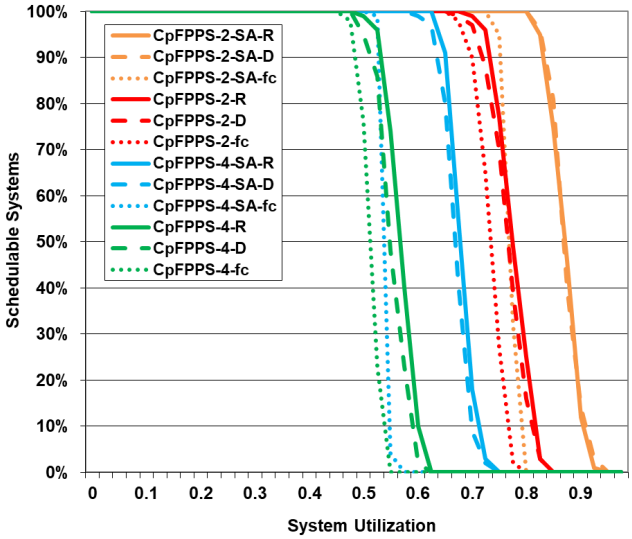


Fig. 15 pFPPS: Task allocation: Schedulable systems Simulated Annealing vs. baseline for 2 and 4 cores, context-independent and context-dependent schedulability tests.

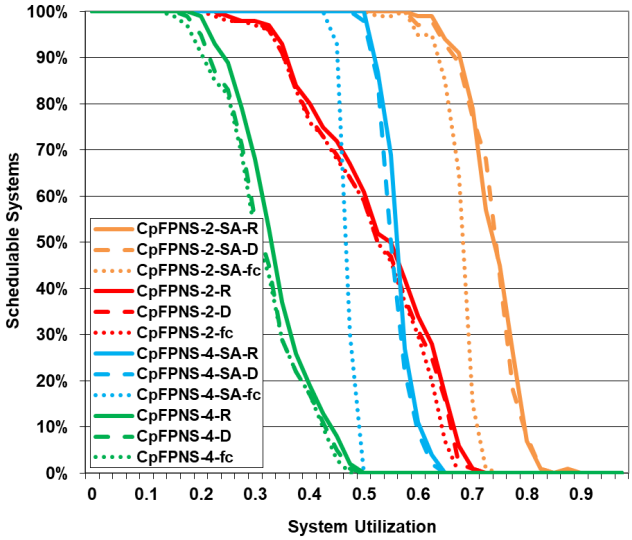


Fig. 16 pFPNS: Task allocation: Schedulable systems Simulated Annealing vs. baseline for 2 and 4 cores, context-independent and context-dependent schedulability tests.

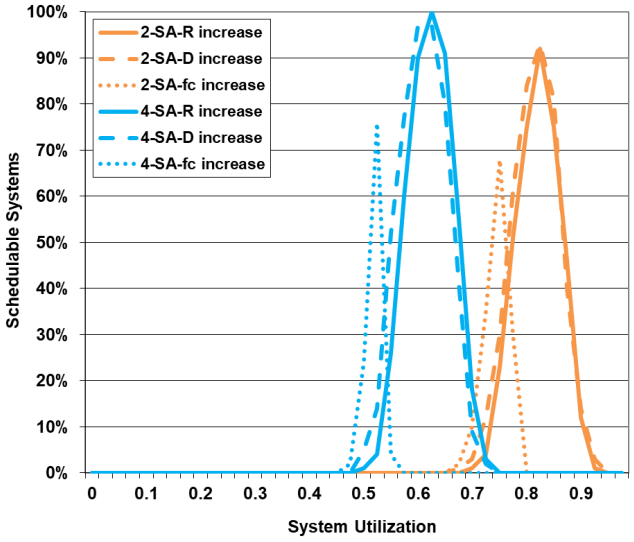


Fig. 17 pFPPS: Task allocation: Increase in schedulable systems Simulated Annealing vs. baseline for 2 and 4 cores, context-independent and context-dependent schedulability tests.

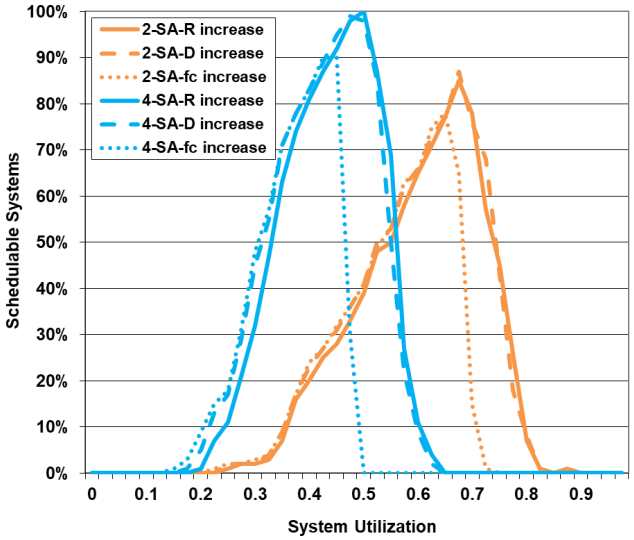


Fig. 18 pFPNS: Task allocation: Increase in schedulable systems Simulated Annealing vs. baseline for 2 and 4 cores, context-independent and context-dependent schedulability tests.

The number of additional systems that were found schedulable using the allocations determined by Simulated Annealing are listed in Table 5, both as a number out of 4000 systems in total, and as a percentage. Observe that the gains obtained by using Simulated Annealing are larger for the context-dependent tests for systems with 4 cores than for systems with 2 cores. This is because the larger systems present more opportunities for task allocations that reduce the interference between the cores. Further, the improvements are considerably higher for non-preemptive scheduling (pFPNS), than for preemptive scheduling (pFPPS). This is because with non-preemptive scheduling, schedulability is sensitive not only to cross-core interference, but also to the distribution of the periods and deadlines of the tasks allocated to each core. Allocating tasks that have similar deadlines to the same core can greatly improve schedulability, since with non-preemptive scheduling the execution time of each task must fit within the shortest deadline of the other tasks on the same core. Appropriate allocation of tasks to cores can thus be effective in addressing the blocking problem with non-preemptive scheduling. In determining task allocations for pFPNS, Simulated Annealing is effectively balancing two factors: (i) blocking effects, and (ii) cross-core interference, with the combination leading to substantially improved schedulability, as illustrated in Figures 16 and 18.

The effects of task allocation using Simulated Annealing can be observed by examining the total resource sensitivity utilization ($\sum_{\tau_i \in \Gamma_x} X_i^r / T_i$) for each core, in the case of two cores. Figures 19 and 20 show the total resource sensitivity utilization for the task sets on each of the two cores, as a frequency distribution for 1000 different systems. The task sets on each core had an initial execution time utilization of 0.5. Since the default Sensitivity Factor of $SF = 0.25$ was used, the baseline (red bar) indicates that both cores had an initial total resource sensitivity utilization of 0.125. Note, the red bar extends to 1.0, and is cut off in both Figures. Figure 19 shows the results for pFPPS. The green and orange frequency distributions are for Simulated Annealing combined with the context-dependent schedulability tests, i.e. **CpFPPS-2-SA-R** and **CpFPPS-2-SA-D** respectively. Observe that these distributions are bi-modal, with the vast majority of the best allocations a substantial distance away from being balanced in terms of the total resource sensitivity utilization on each core. Note, since the total resource sensitivity utilization across both cores remains constant, each distribution is symmetrical about the baseline. The blue frequency distribution is for Simulated Annealing combined with the context-independent schedulability test, i.e. **CpFPPS-2-SA-fc**. With this test, the cross-core contention and interference that a task is assumed to be subject to does not depend on the tasks allocated to other cores. As a consequence, there is no selection pressure to place tasks with high resource sensitivity and high resource stress on one core and others with low resource sensitivity and low resource stress on the other core. Hence, this distribution resembles a normal distribution, with the best allocations far more balanced in terms of the total resource sensitivity utilization on each core.

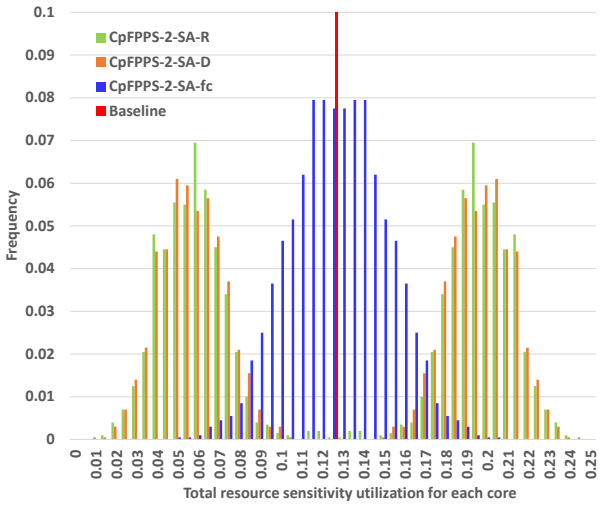


Fig. 19 pFPPS: Frequency distribution of the total resource sensitivity utilization on 2 cores: Simulated Annealing with context-independent and context-dependent tests vs. baseline.

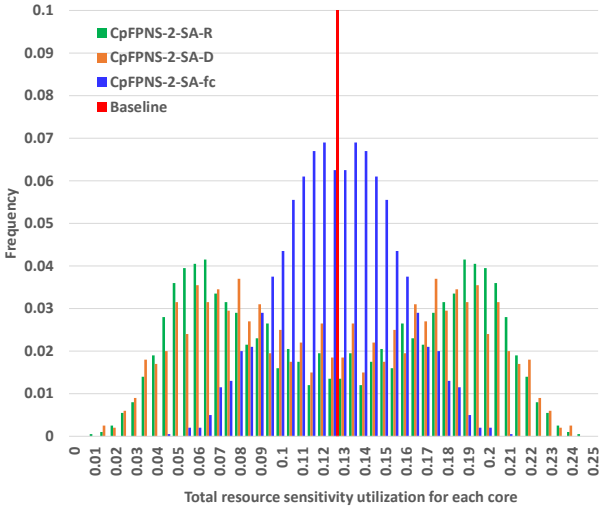


Fig. 20 pFPNS: Frequency distribution of the total resource sensitivity utilization on 2 cores: Simulated Annealing with context-independent and context-dependent tests vs. baseline.

Figure 20 shows similar results for pFPNS. The green and orange frequency distributions are for Simulated Annealing combined with the context-dependent schedulability tests, i.e. **CpFPNS-2-SA-R** and **CpFPNS-2-SA-D** respectively. These distributions are again bi-modal, with the majority of the best allocations a substantial distance away from being balanced in terms of the total resource sensitivity utilization on each core. Compared with the preemptive case, however, there is more variation and the two modes are less well defined. The blue frequency distribution is for Simulated Annealing combined with the context-independent schedulability test, i.e. **CpFNPS-2-SA-fc**. With this test, the cross-core contention and interference that a task is assumed to be subject to does not depend on the tasks allocated to other cores. Thus, similar to the preemptive case, this distribution resembles a normal distribution, with the best allocations far more balanced in terms of the total resource sensitivity utilization on each core.

7.4 Summary

In this section we argued that commonly applied heuristic methods of task allocation based on a greedy assignment algorithm, for example First-Fit, Decreasing Utilization, are not viable in the context of the MRSS task model, since the schedulability of a task allocated to one core is typically dependent on the tasks allocated to other cores. We showed that Simulated Annealing is highly effective at finding schedulable task allocations when used in conjunction with the context-independent and context-dependent schedulability tests introduced for the MRSS task model. Further, the improvement over a simple baseline allocation is substantially increased when context-dependent schedulability tests are employed. This is because such tests take into account both the resource sensitivity of the tasks on the same core as the task under analysis, and the resource stress due to tasks on other cores. By doing so, they provide an opportunity for the allocation algorithm to assign tasks to cores in a way that reduces cross-core contention and interference, and hence improves schedulability. This tends to result in allocations where the resource sensitivity and resource stress of tasks is high on some cores and low on others.

Our focus here has been on task allocation assuming a single shared hardware resource; however, in practice the problem is further complicated by different groups of tasks accessing different shared hardware resources, with different resource sensitivities and different resource stresses. It is clear that in general highly effective solutions to the task allocation problem will only be possible via search-based techniques, such as Simulated Annealing or Genetic Algorithms, with a carefully designed fitness function that optimizes both schedulability and other important criteria such as robustness and extensibility.

8 Conclusions

The main contributions of this paper are (i) the Multi-core Resource Stress and Sensitivity (MRSS) task model, underpinned by an industrial case-study; (ii) schedulability analyses for the MRSS task model, and their evaluation; and (iii) compatible task allocation strategies, based on Simulated Annealing. 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 test effectiveness, and complexity results showing the opposite trade-off between context independence and schedulability test efficiency.
- 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.

Further, a preliminary case study explores the resource stress and resource sensitivity of 24 tasks from a Rolls-Royce aero-engine control system. This industrial case study provides an underpinning proof-of-concept for the MRSS task model. Finally, a consideration of task allocation shows that commonly used heuristics and greedy assignment algorithms are no longer viable when task schedulability depends on cross-core contention, and hence the allocation of tasks to other cores. Instead, Simulated Annealing, with an appropriate cost-function, can provide an effective method of task allocation that optimizes both schedulability and robustness, under the MRSS task model.

Acknowledgements

This research was funded in part by Innovate UK HICLASS project (113213) and the ESPRC grant STRATA (EP/N023641/1) and MARCH (EP/V006029/1). EPSRC Research Data Management: No new primary data was created during this study.

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.

References

Akesson B, Nasri M, Nelissen G, Altmeyer S, Davis RI (2020) 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, IEEE, pp 3–11, DOI 10.1109/RTSS49844.2020.00012, URL https://doi.org/10.1109/RTSS49844.2020.00012

Akesson B, Nasri M, Nelissen G, Altmeyer S, Davis RI (2021) A comprehensive survey of industry practice in real-time systems. Real-Time Syst p 41, DOI 10.1007/s11241-021-09376-1, URL https://doi.org/10.1007/s11241-021-09376-1

Altmeyer S, Douma R, Lunniss W, Davis RI (2014) Evaluation of cache partitioning for hard real-time systems. In: 26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014, IEEE Computer Society, pp 15–26, DOI 10.1109/ECRTS.2014.11, URL https://doi.org/10.1109/ECRTS.2014.11

Altmeyer S, Davis RI, Indrusiak LS, Maiza C, Nélis V, Reineke J (2015) A generic and compositional framework for multicore response time analysis. In: Forget J (ed) Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015, ACM, pp 129–138, DOI 10.1145/2834848.2834862, URL https://doi.org/10.1145/2834848.2834862

Altmeyer S, Douma R, Lunniss W, Davis RI (2016) On the effectiveness of cache partitioning in hard real-time systems. Real Time Syst 52(5):598–643, DOI 10.1007/s11241-015-9246-8, URL https://doi.org/10.1007/s11241-015-9246-8

Andersson B, Kim H, de Niz D, Klein MH, Rajkumar R, Lehoczky JP (2018) Schedulability analysis of tasks with corunner-dependent execution times. ACM Trans Embed Comput Syst 17(3):71:1–71:29, DOI 10.1145/3203407, URL https://doi.org/10.1145/3203407

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

Audsley NC, Burns A, Richardson M, Tindell KW, Wellings AJ (1993) Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal 8:284–292(8), URL https://digital-library.theiet.org/content/journals/10.1049/sej.1993.0034

Bastoni A, Brandenburg BB, Anderson JH (2010) Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In: International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, pp 33–44

Bate I, Griffin D, Lesage B (2020) Establishing confidence and understanding uncertainty in real-time systems. In: Cucu-Grosjean L, Medina R, Altmeyer S, Scharbarg J (eds) 28th International Conference on Real Time Networks and Systems, RTNS 2020, Paris, France, June 10, 2020, ACM, pp

67–77, DOI 10.1145/3394810.3394816, URL https://doi.org/10.1145/3394810.3394816

Cheng S, Chen J, Reineke J, Kuo T (2017) 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, IEEE Computer Society, pp 209–219, DOI 10.1109/RTSS.2017.00027, URL https://doi.org/10.1109/RTSS.2017.00027

Dasari D, Andersson B, Nélis V, Petters SM, Easwaran A, Lee J (2011) 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, IEEE Computer Society, pp 1068–1075, DOI 10.1109/TrustCom.2011.146, URL https://doi.org/10.1109/TrustCom.2011.146

Davis RI, Burns A (2007) Robust priority assignment for fixed priority real-time systems. In: Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS 2007), 3-6 December 2007, Tucson, Arizona, USA, IEEE Computer Society, pp 3–14, DOI 10.1109/RTSS.2007.11, URL https://doi.org/10.1109/RTSS.2007.11

Davis RI, Burns A (2010) On optimal priority assignment for response time analysis of global fixed priority pre-emptive scheduling in multiprocessor hard real-time systems. Tech. Rep. YCS-2010-451, University of York, Computer Science Dept.

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

Davis RI, Burns A, Bril RJ, Lukkien JJ (2007) Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. Real Time Syst 35(3):239–272, DOI 10.1007/s11241-007-9012-7, URL https://doi.org/10.1007/s11241-007-9012-7

Davis RI, Zabos A, Burns A (2008) Efficient exact schedulability tests for fixed priority real-time systems. IEEE Trans Computers 57(9):1261–1276, DOI 10.1109/TC.2008.66, URL https://doi.org/10.1109/TC.2008.66

Davis RI, Cucu-Grosjean L, Bertogna M, Burns A (2016) A review of priority assignment in real-time systems. J Syst Archit 65:64–82, DOI 10.1016/j.sysarc.2016.04.002, URL https://doi.org/10.1016/j.sysarc.2016.04.002

Davis RI, Altmeyer S, Indrusiak LS, Maiza C, Nélis V, Reineke J (2018) An extensible framework for multicore response time analysis. Real Time Syst 54(3):607–661, DOI 10.1007/s11241-017-9285-4, URL https://doi.org/10.1007/s11241-017-9285-4

Davis RI, Griffin D, Bate I (2021) Schedulability analysis for multi-core systems accounting for resource stress and sensitivity. In: Brandenburg B (ed) 33rd Euromicro Conference on Real-Time Systems, ECRTS 2021, July 5-9, 2021, Virtual Conference, Schloss Dagstuhl - Leibniz-Zentrum für

Informatik, LIPIcs, vol 196, pp 7:1–7:26

Emberson P, Stafford R, Davis RI (2010) Techniques for the synthesis of multiprocessor tasksets. In: International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), pp 6–11, URL http://retis.sssup.it/waters2010/waters2010.pdf

Fernández M, Gioiosa R, Quiñones E, Fossati L, Zulianello M, Cazorla FJ (2012) Assessing the suitability of the NGMP multi-core processor in the space domain. In: Jerraya A, Carloni LP, Maraninchi F, Regehr J (eds) 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, ACM, pp 175–184, DOI 10.1145/2380356.2380389, URL https://doi.org/10.1145/2380356.2380389

Fisher N, Baruah SK, Baker TP (2006) The partitioned scheduling of sporadic tasks according to static-priorities. In: 18th Euromicro Conference on Real-Time Systems, ECRTS'06, 5-7 July 2006, Dresden, Germany, Proceedings, IEEE Computer Society, pp 118–127, DOI 10.1109/ECRTS.2006.30, URL https://doi.org/10.1109/ECRTS.2006.30

Fuchsen R (2010) How to address certification for multi-core based IMA platforms: Current status and potential solutions. In: 29th Digital Avionics Systems Conference, pp 5.E.3–1–5.E.3–11, DOI 10.1109/DASC.2010.5655461

Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman

George L, Rivierre N, Spuri M (1996) Preemptive and nonpreemptive real-time uniprocessor scheduling. Tech. rep., INRIA Research Report, No. 2966, URL https://hal.inria.fr/inria-00073732

Giannopoulou G, Lampka K, Stoimenov N, Thiele L (2012) Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In: Jerraya A, Carloni LP, Maraninchi F, Regehr J (eds) 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, ACM, pp 63–72, DOI 10.1145/2380356.2380372, URL https://doi.org/10.1145/2380356.2380372

Griffin D, Bate I, Davis RI (2020a) Dirichlet-Rescale (DRS) algorithm software: dgdguk/drs: v1.0.0 available at https://doi.org/10.5281/zenodo.4118059

Griffin D, Bate I, Davis RI (2020b) 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, IEEE, pp 76–88, DOI 10.1109/RTSS49844.2020.00018, URL https://doi.org/10.1109/RTSS49844.2020.00018

Hassan M (2018) 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, IEEE Computer Society, pp 495–505, DOI 10.1109/RTSS.2018.00062, URL

https://doi.org/10.1109/RTSS.2018.00062

Huang W, Chen J, Reineke J (2016) 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, ACM, pp 158:1–158:6, DOI 10.1145/2897937.2898046, URL https://doi.org/10.1145/2897937.2898046

Iorga D, Sorensen T, Wickerson J, Donaldson AF (2020) 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, IEEE, pp 200–212, DOI 10.1109/RTAS48715.2020.000-6, URL https://doi.org/10.1109/RTAS48715.2020.000-6

Joseph M, Pandya PK (1986) Finding response times in a real-time system. Comput J 29(5):390–395, DOI 10.1093/comjnl/29.5.390, URL https://doi.org/10.1093/comjnl/29.5.390

Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of International Conference on Neural Networks (ICNN'95), Perth, WA, Australia, November 27 - December 1, 1995, IEEE, pp 1942–1948, DOI 10.1109/ICNN.1995.488968, URL https://doi.org/10.1109/ICNN.1995.488968

Kim H, de Niz D, Andersson B, Klein MH, Mutlu O, Rajkumar R (2016) Bounding and reducing memory interference in cots-based multi-core systems. Real Time Syst 52(3):356–395, DOI 10.1007/s11241-016-9248-1, URL https://doi.org/10.1007/s11241-016-9248-1

Kim N, Ward BC, Chisholm M, Anderson JH, Smith FD (2017) Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. Real Time Syst 53(5):709–759, DOI 10.1007/s11241-017-9272-9, URL https://doi.org/10.1007/s11241-017-9272-9

Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680, DOI 10.1126/science.220.4598.671, URL https://science.sciencemag.org/content/220/4598/671, https://science.sciencemag.org/content/220/4598/671.full.pdf

Lampka K, Giannopoulou G, Pellizzoni R, Wu Z, Stoimenov N (2014) 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, DOI 10.1007/s11241-014-9211-y, URL https://doi.org/10.1007/s11241-014-9211-y

Law S, Bate I (2016) 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, IEEE Computer Society, pp 189–199, DOI 10.1109/ECRTS.2016.21, URL https://doi.org/10.1109/ECRTS.2016.21

Lesage B, Law S, Bate I (2018) TACO: an industrial case study of test automation for coverage. In: Ouhammou Y, Ridouard F, Grolleau E, Jan M, Behnam M (eds) Proceedings of the 26th International Conference on Real-Time Networks and Systems, RTNS 2018, Chasseneuil-du-Poitou, France, October 10-12, 2018, ACM, pp 114–124, DOI 10.1145/3273905.3273910,

URL https://doi.org/10.1145/3273905.3273910

Leung JY, Whitehead J (1982) On the complexity of fixed-priority scheduling of periodic, real-time tasks. Perform Evaluation 2(4):237–250, DOI 10.1016/0166-5316(82)90024-4, URL https://doi.org/10.1016/0166-5316(82)90024-4

López JM, Díaz JL, García DF (2004) Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. IEEE Trans Parallel Distributed Syst 15(7):642–653, DOI 10.1109/TPDS.2004.25, URL https://doi.org/10.1109/TPDS.2004.25

Maiza C, Rihani H, Rivas JM, Goossens J, Altmeyer S, Davis RI (2019) A survey of timing verification techniques for multi-core real-time systems. ACM Comput Surv 52(3):56:1–56:38, DOI 10.1145/3323212, URL https://doi.org/10.1145/3323212

Monnier Y, Beauvais J, Déplanche A (1998) A genetic algorithm for scheduling tasks in a real-time distributed system. In: 24th EUROMICRO '98 Conference, Engineering Systems and Software for the Next Decade, 25-27 August 1998, Vesteras, Sweden, IEEE Computer Society, pp 20,708–20,714, DOI 10.1109/EURMIC.1998.708092, URL https://doi.org/10.1109/EURMIC.1998.708092

Natale MD, Stankovic JA (1995) Applicability of simulated annealing methods to real-time scheduling and jitter control. In: 16th IEEE Real-Time Systems Symposium, Palazzo dei Congressi, Via Matteotti, 1, Pisa, Italy, December 4-7, 1995, Proceedings, IEEE Computer Society, pp 190–199, DOI 10.1109/REAL.1995.495209, URL https://doi.org/10.1109/REAL.1995.495209

Nowotsch J, Paulitsch M (2012) Leveraging multi-core computing architectures in avionics. In: Constantinescu C, Correia MP (eds) 2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012, IEEE Computer Society, pp 132–143, DOI 10.1109/EDCC.2012.27, URL https://doi.org/10.1109/EDCC.2012.27

Oh J, Wu C (2004) Genetic-algorithm-based real-time task scheduling with multiple goals. J Syst Softw 71(3):245–258, DOI 10.1016/S0164-1212(02)00147-4, URL https://doi.org/10.1016/S0164-1212(02)00147-4

Oh Y, Son SH (1995) Allocating fixed-priority periodic tasks on multiprocessor systems. Real Time Syst 9(3):207–239, DOI 10.1007/BF01088806, URL https://doi.org/10.1007/BF01088806

Paolieri M, Quiñones E, Cazorla FJ, Davis RI, Valero M (2011) Ia^3: 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, IEEE Computer Society, pp 280–290, DOI 10.1109/RTAS.2011.34, URL https://doi.org/10.1109/RTAS.2011.34

Pellizzoni R, Schranzhofer A, Chen J, Caccamo M, Thiele L (2010) Worst case delay analysis for memory interference in multicore systems. In: Micheli GD, Al-Hashimi BM, Müller W, Macii E (eds) Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010, IEEE Computer Society, pp 741–746, DOI 10.1109/DATE.2010.5456952, URL https://doi.

org/10.1109/DATE.2010.5456952

Punnekkat S, Davis RI, Burns A (1997) Sensitivity analysis of real-time task sets. In: Shyamasundar RK, Ueda K (eds) Advances in Computing Science - ASIAN '97, Third Asian Computing Science Conference, Kathmandu, Nepal, December 9-11, 1997, Proceedings, Springer, Lecture Notes in Computer Science, vol 1345, pp 72–82, DOI 10.1007/3-540-63875-X_44, URL https://doi.org/10.1007/3-540-63875-X_44

Radojkovic P, Girbal S, Grasset A, Quiñones E, Yehia S, Cazorla FJ (2012) 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, DOI 10.1145/2086696.2086713, URL https://doi.org/10.1145/2086696.2086713

Rapita Systems (2019) Multicore timing analysis for do-178c. https://www.rapitasystems.com/downloads/multicore-timing-analysis-do-178c

Rihani H, Moy M, Maiza C, Davis RI, Altmeyer S (2016) Response time analysis of synchronous data flow programs on a many-core processor. In: Plantec A, Singhoff F, Faucou S, Pinho LM (eds) Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016, ACM, pp 67–76, DOI 10.1145/2997465.2997472, URL https://doi.org/10.1145/2997465.2997472

Salman AA, Ahmad I, Al-Madani S (2002) Particle swarm optimization for task assignment problem. Microprocess Microsystems 26(8):363–371, DOI 10.1016/S0141-9331(02)00053-4, URL https://doi.org/10.1016/S0141-9331(02)00053-4

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

Schranzhofer A, Pellizzoni R, Chen J, Thiele L, Caccamo M (2010) Worst-case response time analysis of resource access models in multi-core systems. In: Sapatnekar SS (ed) Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010, ACM, pp 332–337, DOI 10.1145/1837274.1837359, URL https://doi.org/10.1145/1837274.1837359

Tindell K, Burns A, Wellings AJ (1992) Allocating hard real-time tasks: An np-hard problem made easy. Real Time Syst 4(2):145–165, DOI 10.1007/BF00365407, URL https://doi.org/10.1007/BF00365407

Valsan PK, Yun H, Farshchi F (2016) 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, IEEE Computer Society, pp 161–172, DOI 10.1109/RTAS.2016.7461361, URL https://doi.org/10.1109/RTAS.2016.7461361

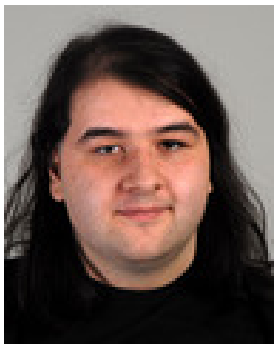
Yun H, Yao G, Pellizzoni R, Caccamo M, Sha L (2013) Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: 19th IEEE Real-Time and Embedded Technology and

Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013, IEEE Computer Society, pp 55–64, DOI 10.1109/RTAS.2013.6531079, URL https://doi.org/10.1109/RTAS.2013.6531079

Yun H, Pellizzoni R, Valsan PK (2015) 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, IEEE Computer Society, pp 184–195, DOI 10.1109/ECRTS.2015.24, URL https://doi.org/10.1109/ECRTS.2015.24

Author Biographies

Robert I. Davis is a Reader in the Real-Time Systems Research Group at the University of York, UK. Robert received his PhD in Computer Science from the University of York in 1995. Since then he has founded three start-up companies, all of which have succeeded in transferring real-time systems research into commercial products. Robert's research interests include real-time scheduling and schedulability analyses for single-core, multi-core, and networked systems.



David Griffin is a member of the Non-Standard Computation Research Group at the University of York, UK. His research has primarily been in the application of non-standard techniques to real-time problems, utilising techniques from various other fields such as lossy compression, statistics, and machine learning.

Iain Bate is Professor of Dependable Real-Time Systems at the University of York, UK. His research interests include scheduling and timing analysis for both safety-critical systems and high-performance systems, and the design and certification of Cyber Physical Systems. He has chaired four leading International Conferences and is a frequent member of Programme Committees. He was the Editor-in-Chief of the Microprocessors and Microsystems journal and then the Journal of Systems Architecture for 15 years. He was a Visiting Professor at the Malardalen University, Sweden for 5 years. He currently leads two industry-related projects related to multi-core design and assurance.