Shared-Resource-Centric Limited Preemptive Scheduling: A Comprehensive Study of Suspension-based Partitioning Approaches

Zheng Dong, Cong Liu, Soroush Bateni, Kuan-Hsun Chen, Jian-Jia Chen, Georg von der Brüggen, Junjie Shi
University of Texas at Dallas, USA
TU Dortmund, Deaprtment of Computer Science, Dortmund, Germany

Citation: 10.1109/RTAS.2018.00026

BIBTEX:

```
@inproceedings{DBLP:conf/rtas/Dong0BCCBS18,
  author    = {Zheng Dong and
               Cong Liu and
               Soroush Bateni and
               Kuan{-}Hsun Chen and
               Jian{-}Jia Chen and
               Georg von der Br{\"{u}}ggen and
               Junjie Shi},
  title     = {Shared-Resource-Centric Limited Preemptive Scheduling: {A} Comprehensive
               Study of Suspension-Based Partitioning Approaches},
  booktitle = {{RTAS}},
  pages     = {164--176},
  publisher = {{IEEE} Computer Society},
  year      = {2018}
}
```

Shared-Resource-Centric Limited Preemptive Scheduling: A Comprehensive Study of Suspension-based Partitioning Approaches

Zheng Dong, Cong Liu, Soroush Bateni
University of Texas at Dallas, USA

Kuan-Hsun Chen, Jian-Jia Chen, Georg von der Brüggen, Junjie Shi
Technical University of Dortmund, Germany

***Abstract*—This paper studies the problem of scheduling a set of hard real-time sporadic tasks that may access CPU cores and a shared resource. Motivated by the observation that the CPU resource is often abundant compared to the shared resources in multi-core and many-core systems, we propose to resolve this problem from a counter-intuitive shared-resource-centric perspective, focusing on judiciously prioritizing and scheduling tasks' requests in a limited preemptive manner on the shared resource while viewing the worst-case latency a task may experience on the CPU cores as suspension delays. We develop a rather comprehensive set of task partitioning algorithms that partition tasks onto the shared resource with the objective of guaranteeing schedulability while minimizing the required size of the shared resource, which plays a critical role in reducing the overall cost and complexity of building resource-constrained embedded systems in many application domains. A GPU-based prototype case study and extensive simulation-based experiments have been conducted, which validate both our shared-resource-centric scheduling philosophy and the efficiency of our suspension-based partitioning solutions in practice.**

1 Introduction

In recent years, the microprocessor industry has turned to multi-core processor designs for next generation embedded systems. By increasing the number of cores, it is possible to dramatically improve the performance as well as the energy efficiency. This trend has brought in the many-core architectures, which incorporate a large number of cores to provide unprecedented advantage in terms of performance-per-watt and inter-core communication latency over traditional microprocessor architectures. Examples of many-core platforms include Intel SCC [15], Godson-T [21], and STHorm [39], which have cores organized in multiple islands sharing various shared resources, such as DRAM modules, on-chip buses, or computing accelerators such as GPU and FPGA. Many-core platforms have been widely used in real-time embedded systems and proved to provide excellent performance for many applications such as computer vision and image analysis workloads [33], [41].

To guarantee real-time properties in such multi-core and many-core systems, a difficult problem is to analyze the execution behaviors of tasks that may access both CPU cores and shared resources. A common approach is to analyze schedulability from a traditional core-centric view [8], [18], [19], [36]. That is, the tasks are scheduled on CPU cores under certain scheduling algorithms, which incorporate the latency due to resource contention in the analysis. The idea is to focus on judiciously allocating CPU resources while viewing shared resources as I/O and bounding the worst-case latency a task may experience on such resources. Then, by treating such latency as suspension delays, we can transform the original multi-resource scheduling problem into single-resource (i.e., CPU) scheduling of suspending tasks.

For a more in-depth treatment of literature on the impact of resource sharing on performance and worst-case timing analysis, a survey can be found in [5]. One line of work in timing analysis for multicore systems is based on an assumption that the CPU execution and shared-resource access patterns are well structured, e.g., [9], [14], [22], [24], [26], [28], [29], [32], [35], [37], [40], [42]. For example, in the superblock execution model [35], the execution of a superblock has three phases: data acquisition, local execution, and data replication phases. A similar model can be found in the open international standard IEC 61131-3. Altmeyer et. al [6] present a framework to decouple response-time analysis in a compositional manner based on the context independent WCET values. However, resource partitioning and task partitioning were not discussed in the above results.

The CPU-centric perspective is sound for traditional embedded systems where computing resources may be insufficient while the contention on shared resources is often light. However, for advanced embedded systems that employ multi-core platforms to serve large-scale real-time workloads, shared resources may become the actual scheduling bottleneck, causing the worst-case latency bound on the shared resource (thus the resulting schedulability test) to be rather pessimistic or even impossible to derive. Motivated by this observation, it may be much more viable to resolve this multi-resource scheduling problem from the counter-intuitive shared-resource centric perspective since tasks may experience much lighter contention on CPU cores, as previously argued in [25] which considers a simplified single-unit shared resource scenario (equivalent to uniprocessor scheduling from a shared-resource-centric perspective).

In this paper, we resolve the multi-resource scheduling problem from the above-argued shared-resource-centric perspective. This means, our developed techniques focus on judiciously scheduling tasks' requests in a limited preemptive manner[1] on the shared resource (treating the shared resource as the first-class units) and bounding the worst-case latency a task may experience on the CPU cores (treating CPU cores as "I/O"). The benefit is obvious, particularly on many-core platforms, where tasks may receive rather trivial or even no interference on CPU cores as the CPU resource is often abundant compared to the shared resource. We seek to develop practical solutions that may benefit embedded systems designers. We develop a comprehensive set of techniques that are based on non-trivial schedulability tests with different runtime complexity and accuracy (w.r.t. both schedulability and the

This work is supported by NSF grants CNS 1527727 and CNS CAREER 1750263; and DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/), subproject B2.

[1]We focus on limited preemptive scheduling (general enough to cover non-preemptive scheduling as well) rather than preemptive scheduling due to the limited- or non-preemptive execution feature seen in several major shared resources in practice such as GPU or memory.

needed size of the shared resource) for scheduling a set of hard real-time (HRT) tasks that may access CPU cores and multiple units of a shared resource in an interleaving manner. If reasonably high runtime/analysis complexity is affordable, then our solutions may yield a high schedulability and a minimized required size of the shared resource, which is one of the most critical factors that may reduce the overall cost and complexity of the SWaP (size, weight, and power) constrained embedded systems.

Contributions. By tackling the multi-resource scheduling problem from a shared-resource-centric perspective, we carefully prioritize and schedule the shared resource requests of tasks and view the worst-case response times of a task on CPU cores as suspensions. We develop a set of task partitioning algorithms, combined with a set of uniprocessor suspension-aware schedulability tests for suspending tasks under limited-preemptive scheduling, that yield efficient schedulability tests and minimized required size of the shared resource. Technically speaking, two open problems are solved: (*i*) to the best of our knowledge, there does not exist any solution on dealing with *the limited-preemptive suspending task partitioning problem*, and (*ii*) how to design partitioning algorithms in this problem context such that the required size of the shared resource is minimized. For practical considerations, we present a set of methods that exhibit different runtime complexities and accuracy w.r.t both schedulability and the required shared resource size. Thus, embedded systems designers may consider adopting different methods according to their affordable runtime overhead.

We have conducted a prototype case study using GPU as the shared resource, which validates both our shared-resource-centric scheduling philosophy and the efficiency of our suspension-based partitioning solutions in practice. Extensive simulation-based experiments have also been conducted, showing that our proposed techniques may simultaneosly yield effective schedulability tests as well as minimized shared resource size in many cases.

2 System Model

We assume in this paper that we have a multicore platform comprised of n identical cores which share a multi-unit resource, e.g., memory or GPU.

Task Model: we consider the problem of scheduling a set of n HRT sporadic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ that may need to access both CPUs and a shared resource, e.g., memory or GPU. Each task is released repeatedly, and each such invocation is called a job. A hard real-time task τ_i is characterized by a 6-tuple $\tau_i(e_i, s_i, p_i, d_i, \sigma_i, z_i)$: p_i denotes the minimum inter-arrival time (also known as period) of τ_i; each job of τ_i has a relative deadline d_i; e_i denotes an upper bound on total execution time of each job of τ_i on a computing core (ignoring the latency due to shared resource accesses); σ_i denotes the maximum number of resource access segments, each of them may consist of several consecutive operations on the shared resource; and each task τ_i may access z_i units of the shared resource concurrently for at most s_i time units (across all of its phases accessing the shared resource). Note that if the platform allows multiple operations to be directly processed without involving CPU cores, a task can incur multiple requests to the shared resource within a resource access segment, e.g., fetching (or writing back in the other direction) a collection of data and instructions from the shared main memory to the local scratchpad memory.

We highlight the following properties on shared resource accesses:

- Each operation on the shared resource is atomic (non-split operation), with a processing time upper-bounded by B. However, a task's execution can be preempted at any point at which an atomic operation completes.[2] This is illustrated in Fig. 1 where task τ_1 has higher priority than τ_2 and requests the shared resource at t_4. Here τ_1 experiences blocking from task τ_2 as the resource is granted to τ_2 at time t_3, and τ_1 preempts τ_2 at time t_5 when one of τ_2's atomic operations completes.
- Each task τ_i may access z_i units of the shared resource concurrently. The shared resource can be partitioned into several regions.
- We consider a fixed-priority resource arbiter in this paper: the arbiter of the shared resource always grants the request that is assigned the highest priority.

Fact 1. *Since each operation on a shared resource is assumed to be atomic and thus non-preemptive, each shared resource segment of task τ_i can be blocked by a lower priority task for at most B time units. Therefore, we know that the maximum blocking time experienced by any job released by task τ_i due to non-preemptive blocking on the shared resource is at most $\sigma_i \times B$.*

The j^{th} job of task τ_i, denoted by $\tau_{i,j}$, is released at time $r_{i,j}$ and has an absolute deadline at time $d_{i,j}$, i.e., $d_{i,j} = r_{i,j} + d_i$. Successive jobs of the same task are required to execute in sequence. The *shared resource utilization* of a task τ_i is defined as $u_i^s = \frac{s_i}{p_i}$, and the total shared resource utilization of the task system τ is defined as $U_{sum}^s = \sum_{i=1}^{n} u_i^s$. Since we do not concern the CPU utilization of a task in our reasoning, for conciseness, let u_i and U_{sum} denote the shared resource utilization of task τ_i and the total shared resource utilization of τ, respectively. Due to space constraints, we focus our attention on the implicit-deadline case in this paper, i.e., $d_i = p_i$ for all tasks τ_i.

Since it is more practical to apply task-level fixed-priority scheduling [23] to many shared resources [16], we focus on RM scheduling on the shared resource, where tasks are prioritized by the shortest-period-first. We assume that ties are broken by task ID, i.e., lower IDs are favored. According to our task indexing rule, we know that task τ_i has a higher priority than any task τ_k when $i < k$.

Example illustrating limited preemptive scheduling on the shared resource. Consider the simplest case in which two tasks τ_1 and τ_2 are assigned on core 1 and core 2, respectively. We assume that τ_1 has higher priority than τ_2. Furthermore, τ_1 accesses 1 unit of shared resource and τ_2 accesses 2 units. Since τ_1 and τ_2 are partitioned in the same shared resource region, the minimum amount of shared resource allocated in this region is 2. We are now analyzing task τ_2 in this example. The schedule of accessing a shared resource by these two tasks is indicated in Fig. 1:

- At time t_1, tasks τ_1 and τ_2 start their computation.

[2]Note that if a task's segment only contains one operation, then the execution of the entire segment becomes non-preemptive.

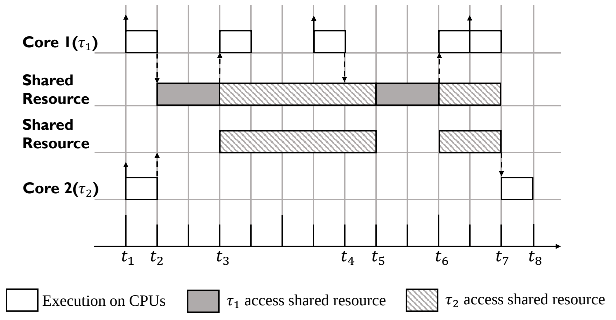

Core 1(τ_1)
Shared Resource
Shared Resource
Core 2(τ_2)
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8
Execution on CPUs
τ_1 access shared resource
τ_2 access shared resource

Fig. 1: An example of accessing a shared resource by two tasks. Each task is statically assigned onto one core.

- At time t_2, tasks τ_1 and τ_2 both attempt to access the shared resource. The request from task τ_1 is granted, while task τ_2 suspends itself on core 2.
- At time t_3, task τ_1 finishes its access to the shared resource and resumes its local computation. At the same time, task τ_2 starts to access the shared resource.
- At time t_4, task τ_1 again attempts to access the shared resource. *Due to the minimum non-preemptive region, this request from task τ_1 is blocked by task τ_2.*
- At time t_5, after leaving the non-preemptive region, task τ_2 on the shared resource is preempted by task τ_1.
- At time t_6, task τ_1 finishes its access to the shared resource and resumes its local computation. At the same time, task τ_2 continues to access the shared resource.
- At time t_7, task τ_2 finishes its access to the shared resource and resumes its local computation.
- At time t_8, task τ_2 finishes its execution.

From the point of view of the shared resource, task τ_2 suspends its resource accesses in time intervals $[t_1, t_2)$ and $[t_7, t_8)$ due to local computations. Moreover, task τ_2 executes on the shared resource in time intervals $[t_3, t_4)$ and $[t_6, t_7)$ and awaits (in the queue) to access the shared resource in time intervals $[t_2, t_3)$ and $[t_5, t_6)$.

Note again that we focus on limited preemptive scheduling (covering non-preemptive scenarios) on the shared resource. This is fundamentally motivated by the limited- or non-preemptive execution feature seen in several major shared resources in practice such as memory [1], [3], [17] and GPUs [2], [38].

3 Design Overview and Preliminary Results

Since the shared resource is the actual scheduling bottleneck, we propose to resolve this scheduling problem from a shared-resource-centric perspective. Specifically, we view the response time experienced by each phase of each task on the CPU core as the *suspension delay*, counter-intuitively viewing the time each task uses to access the shared resource as the actual execution time. Thus, we transform this shared resource scheduling and size minimization problem into a suspending task partitioning problem, i.e., each task τ_i accesses the shared resource for s_i time units (across all its shared resource phases) and suspends for e_i time units (across all its CPU phases). This means that we investigate how to partition a set of HRT sporadic tasks onto the multi-unit shared resource, assuming that those tasks may suspend due to executing on CPU cores. Our goal is two-fold: (*i*) developing HRT schedulability tests for timing predictability, and (*ii*) minimizing the number of units of the shared resource for accommodating all tasks in τ.

To reach this goal, there are two key challenges. First, to the best of our knowledge, there does not exist any work on real-time schedulability analysis that can deal with *the limited preemptive suspending task partition problem*. Thus, we need to first develop new schedulability analysis techniques for the limited preemptive scheduling problem of partitioning a set of suspending tasks (suspension due to executing on CPU cores in our context) onto multiple processors (multiple units of the shared resource in our context). Second, we need to design parallelism-aware partitioning algorithms in terms of minimizing the required size of the shared resource.

Definition 1. *A Resource j has a specified size 1 and can only accommodate sporadic tasks whose $z_i = 1$. At any point in time, only one task can be executed on a resource.*

Definition 2. *A Resource Partition j has a specified size Z_j and can only accommodate sporadic tasks whose $z_i \leq Z_j$. At any point in time, only one task can be executed on a resource partition.*

Our approach solves the above-mentioned two challenges in two separate steps. In step 1 (Sec. 4), we focus on developing schedulability tests that can guarantee real-time constraints. Specifically, we first develop a set of uniprocessor schedulability tests for suspending tasks scheduled under RM, that exhibit different runtime complexities. As the problem context is about partitioning tasks onto a multi-unit shared resource, the term uniprocessor herein refers to a resource partition defined above. In step 1, we assume that $z_i = 1$ holds for any τ_i in the system. Thus, each uniprocessor refers to a unique partition containing $z_i = 1$ unit of the shared resource.

After that, in the second step (Sec. 5), we remove the assumption of having $z_i = 1$ for all tasks, and design a set of partitioning algorithms, including both efficient heuristics and integer linear programming (ILP) based approaches, which seek to minimize the number of units of the shared resources needed to accommodate all tasks in the system. Note that various schedulability conditions developed in step 1 can be applied to the partitioning algorithms herein for checking schedulability. We believe that developing a comprehensive set of partitioning algorithms combined with various schedulability tests, both exhibiting varying complexities and accuracy (w.r.t. both schedulability and the required shared resource size), shall yield a wide range of practical solutions that can be adopted by embedded system designers under different scenarios.

Before presenting our developed partitioning algorithms and the corresponding schedulability tests, we first provide a review on existing schedulability tests for uniprocessor suspending task scheduling (including a set of new constant-time schedulability tests we derive in this paper based on existing tests as shown in Sec. 3.2), categorized by their runtime complexity.

3.1 Pseudo-Polynomial-Time Tests

We first summarize the existing results for testing the schedulability of a suspending task τ_k under fixed-priority preemptive uniprocessor scheduling with pseudo-polynomial-time complexity. The following tests assume that all the tasks with higher-priority than τ_k, denoted by a set $hp(k)$, can meet their deadlines for a constrained-deadline suspending task set. They all need pseudo-polynomial-time complexity.

Task τ_k is schedulable under the fixed-priority preemptive scheduling if one of the following conditions in the following four lemmas hold. The correctness of these four lemmas can be found in the unifying analysis by Chen et. al [11].

Lemma 1 (Suspension as Carry-In).

$$\exists t \mid 0 < t \le p_k, \quad e_k + s_k + \sum_{\tau_i \in hp(k)} \left(\left\lceil \frac{t}{p_i} \right\rceil + 1 \right) s_i \le t \quad (1)$$

Lemma 2 (Suspension as Jitter).

$$\exists t \mid 0 < t \le p_k, \quad e_k + s_k + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t + p_i - s_i}{p_i} \right\rceil s_i \le t \quad (2)$$

According to Fact 1, the maximum blocking time of task τ_k due to *non-preemptive blocking* from lower priority tasks on the shared resource is at most $\sigma_k \times B$, then Lemma 1 to Lemma 2 under limited preemptive task model can be rewritten as following:

Lemma 3 (Suspension and Blocking as Carry-In).

$$\exists t \mid 0 < t \le p_k, \quad s_k + e_k + \sigma_k \times B + \sum_{\tau_i \in hp(k)} \left(\left\lceil \frac{t}{p_i} \right\rceil + 1 \right) s_i \le t \quad (3)$$

Lemma 4 (Suspension and Blocking as Jitter).

$$\exists t \mid 0 < t \le p_k, \quad s_k + e_k + \sigma_k \times B + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t + p_i - s_i}{p_i} \right\rceil s_i \le t \quad (4)$$

The proofs of the above two lemmas are given in an appendix. Note that there are two other strategies for preemptive scheduling: one by Jane Liu [31, Pages 164-165] (Theorem 3 in [12]) and another by Chen et al. [11] (Theorem 5 in [12]). Since the proof provided in [11] is heavily based on fixed-priority preemptive scheduling, these tests are unfortunately difficult to be extended to limited preemptive scheduling.

3.2 Constant-Time Tests

We further derive the following RM schedulability tests exhibiting only constant-time complexity based on a recent schedulability analysis framework k^2U [10]. We will use the following time-demand function $W_i(t)$ for the simple sufficient schedulability analysis:

Lemma 5. *Task τ_k is schedulable under RM limited preemptive scheduling if one of the following conditions holds:*

$$\left(\frac{s_k + e_k + \sigma_k \times B}{p_k} + 2 \right) \prod_{\tau_i \in hp(k)} \left(\frac{s_i}{p_i} + 1 \right) \le 3 \quad (5a)$$

$$\sum_{\tau_i \in hp(k)} \frac{s_i}{p_i} \le \ln \left(\frac{3}{\frac{s_k + e_k + \sigma_k \times B}{p_k} + 2} \right) \quad (5b)$$

Proof: This comes from the k^2U framework where $\alpha_i \le 2$ and $\beta_i \le 1$, as explained in [10] and [30]. ■

Lemma 6. *Task τ_k is schedulable under RM limited preemptive scheduling if $\sum_{\tau_i \in hp(k)} \frac{s_i}{p_i} < 1$ and*

$$\frac{e_k + s_k + \sigma_k \times B}{p_k} + \frac{\sum_{\tau_i \in hp(k)} \left(2s_i - \frac{s_i^2}{p_i} \right)}{p_k} + \sum_{\tau_i \in hp(k)} \frac{s_i}{p_i} \le 1 \quad (6)$$

The proof of Lemma 6 is given in the appendix. The time complexity to evaluate the schedulability tests in Lemmas 5 and 6 is $O(1)$ if $\prod_{\tau_i \in hp(k)} (1 + \frac{s_i}{p_i})$, $\sum_{\tau_i \in hp(k)} s_i$ and $\sum_{\tau_i \in hp(k)} \frac{s_i^2}{p_i}$ have been calculated in the previous iterations.

3.3 Schedulability Test for A Task Set

The tests in Section 3.2 can be applied to RM scheduling by validating whether the conditions are satisfied for each task τ_k in the task set. We can also consider the schedulability of the whole task set by using the following lemma. The proof of Lemma 7 is given in the appendix.

Lemma 7. *All the tasks in a task set τ are schedulable under RM preemptive scheduling on a shared resource if*

$$\sum_{\tau_i \in \tau} \frac{s_i}{p_i} \le \ln \left(\frac{3}{2 + \max_{\tau_i \in \tau} \frac{s_i + e_i + \sigma_i \times B}{p_i}} \right) \quad (7)$$

4 Step 1: Developing Schedulability Tests for Suspension-aware Task Partitioning

As discussed earlier, in the first step, we consider a special case in which $z_i = 1$ holds for any task τ_i in the system. With the set of uniprocessor suspension-aware schedulability tests presented in Sec. 3, we now present an efficient partitioning algorithm, namely *STPartition*, and an ILP formulation for scheduling suspending tasks on a multi-unit shared resource and the corresponding schedulability conditions. *STPartition* will return the number of units of the shared resource that is needed to accommodate all tasks in the system. Note that for implicit-deadline sporadic suspending task systems, the feasibility-analysis for partitioning scheduling is NP-hard in the strong sense, by transformation from the bin-packing problem [27]. Thus, designing an optimal partitioning algorithm with acceptable time complexity is hard.

4.1 A Partitioning Heuristic and its Analysis

We develop a fast Algorithm *STPartition* that partitions a set of suspending tasks in τ onto multiple shared resources. Its pseudocode is shown in Algorithm 1. Tasks are indexed according to RM. Then for each task τ_i in order, we find a set of feasible used resources onto which τ_i can be assigned (lines 4-8). We then assign τ_i to one of them arbitrarily (lines 9-10). If a task cannot be assigned to any used shared resource, then it is assigned to a new resource on which no task has been assigned (line 12). The algorithm returns the units of shared resources (denoted by r) needed to schedule all tasks in τ.

Fitting algorithms with varying time complexity. In line 12 in Algorithm 1, there are several options to choose a resource to assign task τ_k, including

- *First-Fit* (FF): assign to the smallest index j that is feasible.

Algorithm 1 STPartition pseudocode.

```
Input: $\tau$;
Output: resource allocation and resource assignment for the tasks in $\tau$;
 1: index the tasks in the rate-monotonic order;
 2: $\pi_1 \leftarrow \{\tau_1\}$, $r \leftarrow 1$;
 3: for $k \leftarrow 2$ to n do
 4:     for $j \leftarrow 1$ to $r$ do
 5:         if $\tau_k$ can pass the schedulability of RM on the shared resource $j$
            under the interference from $\pi_j$ then
 6:             mark $j$ as a feasible resource for $\tau_k$;
 7:         end if
 8:     end for
 9:     if the set of the feasible resources for $\tau_k$ is not empty then
10:         assign $\tau_k$ to an arbitrarily feasible resource $j$ and set $\pi_j \leftarrow \pi_j \cup \{\tau_k\}$;
11:     else
12:         $r \leftarrow r + 1$ and $\pi_r \leftarrow \{\tau_k\}$;
13:     end if
14: end for
```

- *Best-Fit* (BF): assign to the index j that is feasible and has the maximum $\sum_{\tau_i \in \pi_j} \frac{s_i}{p_i}$.
- *Worst-Fit* (WF): assign to the smallest index j that is feasible and has the minimum $\sum_{\tau_i \in \pi_j} \frac{s_i}{p_i}$.

Except the sorting of the tasks in the RM order, *STPartition* has polynomial-time complexity of $O(n \cdot r)$ if the tests from Lemma 5 to Lemma 6 is used. Moreover, *STPartition* has pseudo-polynomial-time complexity if the pseudo-polynomial-time tests in Section 3.1 are used. According to the above discussions, by adopting the first-fit strategy and the STPartition algorithm, we have formulated different strategies as follows:

- **ST-FF-TDA(Baseline)**: adopt suspensions simply as computation (i.e., extending the execution time parameter of each task to accomodate its suspension length).
- **ST-FF-TDA(Carry)**: adopt Lemma 3.
- **ST-FF-TDA(Jitter)**: adopt Lemma 4.
- **ST-FF-TDA(Mixed)**: adopt all the above TDA-based tests. As long as a task can pass one of these tests, it can be assigned to that resource.

- **ST-FF-CT(Baseline)**: adopt suspension as computation and use $\ln(2)$ as the utilization bound.
- **ST-FF-CT(Carry)**: adopt Eq. (5a).
- **ST-FF-CT(Jitter)**: adopt Eq. (6).
- **ST-FF-CT(Mixed)**: adopt all the above CT tests. As long as a task can pass one of these tests, it can be assigned to that resource.

Similarly, all the methods can be combined with the other two fitting methods BF and WF.

We conclude the STPartition algorithm with the following worst-case analyses, which illustrate the upper bound of the system utilization loss yielded by STPartition algorithm using different schedulability tests. The proofs of the the following two theorems are in Appendix.

Theorem 1. *Algorithm ST-FF-CT(Carry) always successfully partitions any sporadic suspending task system τ on m shared resources for which*

$$U_{sum} \leq m \times \ln\left(\frac{3}{2 + \overline{u_{max}}}\right) \tag{8}$$

where $\overline{u_{max}} = \max_{i=m+1}^{n} \frac{e_i + s_i + \sigma_i \times B}{p_i}$

Theorem 2. *Algorithm ST-FF-CT(Jit) always successfully partitions any sporadic suspending task system τ on m shared resources for which*

$$U_{sum} \leq \frac{m}{3}(1 - \overline{u_{max}}) \tag{9}$$

where $\overline{u_{max}} = \max_{i=m+1}^{n} \frac{e_i + s_i + \sigma_i \times B}{p_i}$

4.2 An ILP-based Approach

In addition to Algorithm *STPartition*, we also develop an ILP-based partitioning approach. Among the schedulability tests in Section 3.2, the conditions in Eqs. (5b) and (6) are compatible with linear programming. Therefore, we will explain how to design ILPs based on these three schedulability tests. Suppose that $x_{i,j}$ is a binary variable in $\{0, 1\}$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$ and that y_j is a binary variable in $\{0, 1\}$ for $j = 1, 2, \ldots, n$. The variable $x_{i,j}$ is 1 if task τ_i is allocated to resource j; otherwise $x_{i,j}$ is 0. The variable y_j is 1 if shared resource j is allocated to be used; otherwise y_j is 0. We first explain the conditions that define the number of allocated shared resources and the task partitioning:

$$\text{min.} \quad \sum_{j=1}^{n} y_j \tag{10a}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} x_{i,j} = 1 \qquad \forall i = 1, 2, \ldots, n \tag{10b}$$

$$x_{i,j} \leq y_j \qquad \forall i, j = 1, 2, \ldots, n \tag{10c}$$

$$y_j \in \{0, 1\} \qquad \forall j = 1, 2, \ldots, n \tag{10d}$$

$$x_{i,j} \in \{0, 1\} \qquad \forall i, j = 1, 2, \ldots, n \tag{10e}$$

The condition in Eq. (10b) forces a task τ_i to be allocated to *exactly* one shared resource, and the condition in Eq. (10c) allows task τ_i to be allocated to a shared resource j only when y_j is 1.

We now explain how to specify a linear constraint that describes the schedulability test in Eq. (5b). If $x_{k,j}$ is 1, then task τ_k is assigned on shared resource j. If $x_{k,j}$ is 0, then task τ_k is not assigned on shared resource j. Therefore, we use $(1 - x_{k,j}) + x_{k,j} \ln\left(\frac{3}{\frac{s_k + e_k + \sigma_k \times B}{p_k} + 2}\right)$ to define the right-hand-side of the condition in Eq. (5b), which results in 1 if $x_{k,j} = 0$ and results in $\ln\left(\frac{3}{\frac{s_k + e_k + \sigma_k \times B}{p_k} + 2}\right)$ if $x_{k,j} = 1$. Therefore, $\forall j, k = 1, \ldots, n$

$$\sum_{i=1}^{k-1} \frac{s_i}{p_i} x_{i,j} \leq (1 - x_{k,j}) + x_{k,j} \ln\left(\frac{3}{\frac{s_k + e_k + \sigma_k \times B}{p_k} + 2}\right) \tag{11}$$

For notational brevity, let v_i be $\left(2s_i - \frac{s_i^2}{p_i}\right)$. We use $(1 - x_{k,j})n + x_{k,j}$, which results in n if $x_{k,j} = 0$ and results in 1 if $x_{k,j} = 1$. The condition in Eq. (6) can be written as the following linear constraint $\forall j, k = 1, \ldots, n$:

$$\left(\frac{s_k + e_k + \sigma_k \times B}{p_k}\right) x_{k,j} + \sum_{i=1}^{k-1} \left(\frac{s_i}{p_i} + \frac{v_i}{p_k}\right) x_{i,j} \leq (1 - x_{k,j})n + x_{k,j} \tag{12}$$

Since we assume that $s_k + e_k > 0$, the condition $\sum_{\tau_i \in hp(k)} \frac{s_i}{p_i} < 1$ holds automatically when the condition in Eq. (12) holds if $x_{k,j}$ is 1.

The condition in Eq. (7) can be written as follows:

$$\sum_{i=1}^{n} \frac{s_i}{p_i} x_{i,j} \leq \ln\left(\frac{3}{2 + \max_{\tau_i}\left\{\frac{s_i+e_i+\sigma_i \times B}{p_i}\right\}}\right), \forall j = 1, 2, \ldots, n \quad (13)$$

According to the above discussions, we have formulated different ILPs, depending on the adopted schedulability. They are as follows:

- **ILP-Baseline**: Eq. (10) with suspension as computation under the utilization bound $\ln(2)$.
- **ILP-Carry**: Eq. (10) together with Eq. (11).
- **ILP-Jitter**: Eq. (10) together with Eq. (12).
- **ILP-Inflation**: Eq. (10) together with Eq. (13).

We can further improve the schedulability tests used in the ILP by combining the tests from Eq. (11) and (12). If one of the tests is feasible, we can safely assign task τ_k to resource j. Here, we introduce binary variables $\eta_{k,j,\ell}$ for $k, j = 1, 2, \ldots, n$ and $\ell = 1, 2$. If $x_{k,j}$ is 1, then $\eta_{k,j,1}$ indicates the case whether the test in Eq. (5b) is successful and $\eta_{k,j,2}$ indicates the case whether the test in Eq. (6) is successful. Therefore, the condition in Eq. (11) is revised by replacing $x_{k,j}$ with $\eta_{k,j,1}$ and the condition in Eq. (12) is revised by replacing $x_{k,j}$ with $\eta_{k,j,2}$. Moreover, we need at least one of tests is successful to set $x_{k,j}$ to 1. If $x_{k,j}$ is 0, then the solution $\eta_{k,j,1} = \eta_{k,j,2} = 0$ indicates that there is no need to test the schedulability of task τ_k on resource j. Let Δ_k denote $\frac{s_k+e_k+\sigma_k \times B}{p_k}$. Therefore, the constraints are as follows:

$$\forall j, k = 1, \ldots, n$$

$$\sum_{\ell=1}^{2} \eta_{k,j,\ell} \geq x_{k,j} \quad (14a)$$

$$\sum_{i=1}^{k-1} \frac{s_i}{p_i} x_{i,j} \leq (1 - \eta_{k,j,1}) + \eta_{k,j,1} \ln\left(\frac{3}{\Delta_k + 2}\right) \quad (14b)$$

$$\Delta_k \eta_{k,j,2} + \sum_{i=1}^{k-1} \left(\frac{s_i}{p_i} + \frac{v_i}{p_k}\right) x_{i,j} \leq (1 - \eta_{k,j,2})n + \eta_{k,j,2} \quad (14c)$$

The above ILP by using Eq. (10) together with Eq. (14) is called **ILP-Combo**.

5 Step 2: Shared Resource Minimization

In Sec. 4, we assumed that $z_i = 1$ holds for all tasks in the system, i.e., each task τ_i requires one unit of the shared resource. We now remove this assumption and present two advanced partitioning methods that may minimize the total number of units of the shared resource required to accommodate and successfully schedule all tasks in τ.

5.1 An Efficient Heuristic Algorithm

We now introduce an efficient partitioning heuristic algorithm that can effectively reduce the required size of the shared resources, motivated by the following observation.

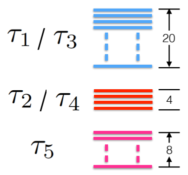
Example Note again that an implicit-deadline sporadic task that accesses both CPU cores and shared resources can be denoted as $\tau_i(e_i, s_i, p_i, d_i, \sigma_i, z_i)$. Consider a task system containing five such tasks (shown in Fig 2a), $\tau_1(1, 3, 10, 10, 5, 20)$, $\tau_2(1, 2, 16, 16, 5, 4)$, $\tau_3(2, 4, 16, 16, 5, 20)$, $\tau_4(1, 3, 16, 16, 5, 4)$, and $\tau_5(1, 2, 10, 10, 5, 8)$. Non-preemptive block size is 0.001. If the system grants each task its required number of shared resources, then all tasks can be accommodated which yields a total shared resource size of 56 units, which implies creating a new resource partition for each task, as illustrated in in Fig. 2b.

Algorithm 2 PSTPartition pseudocode.

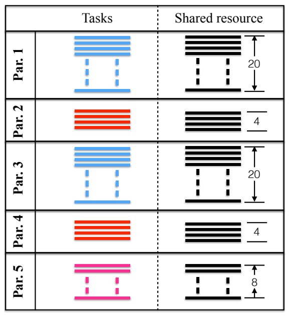
```
Input: $\tau$;
Output: resource allocation and resource assignment for the tasks in $\tau$;
 1: sort the tasks in $\tau$ according to $z_i$ values, in which $z_i \geq z_{i+1}$ and ties
    are broken by preferring smaller $p_i$ for a smaller index;
 2: $\pi_1 \leftarrow \{\tau_1\}$, $r \leftarrow 1$, and $Z_1 \leftarrow z_1$;
 3: for $k \leftarrow 2$ to n do
 4:     for $j \leftarrow 1$ to $r$ do
 5:         if $z_k \leq Z_j$ then
 6:             if all the tasks in $\pi_j \cup \{\tau_k\}$ can pass the schedulability of RM
                on the shared resource partition $j$ then
 7:                 mark $j$ as a feasible resource for $\tau_k$;
 8:             end if
 9:         end if
10:     end for
11:     if the set of the feasible resource partitions for $\tau_k$ is not empty then
12:         assign $\tau_k$ to an arbitrarily feasible resource partition $j$ and set $\pi_j \leftarrow$
            $\pi_j \cup \{\tau_k\}$;
13:     else
14:         $r \leftarrow r + 1$, $\pi_r \leftarrow \{\tau_k\}$ and $Z_r \leftarrow z_k$;
15:     end if
16: end for
17: return $\sum_{i=1}^{r} Z_i$ and $\pi_1, \pi_2, \ldots, \pi_r$
```

An improved partitioning strategy (as shown in Fig. 2c) would assign τ_1 and τ_2 onto a resource partition with 20 units, assign τ_3 and τ_4 onto a second resource partition with 20 units, and assign τ_5 onto a third resource partition with 8 units. The resulting resource partition would be still schedulable under RM according to Eq. (5a), but yields a reduced shared resource size of 48 units. This is because it is possible to schedule multiple tasks inside one resource partition, thus reducing the required shared resource size, i.e., the required shared resource size equals the maximum z_i value among all tasks assigned to the corresponding resource partition. Motivated by this, we know that a task's z_i value may decide the required shared resource size of the resource partition to which this task is assigned. Thus, our intuitive idea is to *assign tasks with similar z_i values onto the same resource partition*. For the above example, if we assign τ_1 and τ_3 together to a resource partition with 20 units of the shared resource, and assign τ_2, τ_4, and τ_5 onto another resource partition with 8 units of the shared resource, then tasks in both resource partitions are schedulable under RM according to Eq. (5a). The resulting total shared resource size is merely 28 (as shown in Fig. 2d). This is because if we can schedule the tasks with similar z_i values on one resource partition, the negative impact due to multiple tasks's parallelism can be "masked" by only one task that has the maximum parallelism, i.e., z_i value. Our parallelism-aware partitioning heuristic, namely *PSTPartition*, is inspired by the above-discussed idea, which always seeks to assign tasks with similar z_i values to the same resource partition.

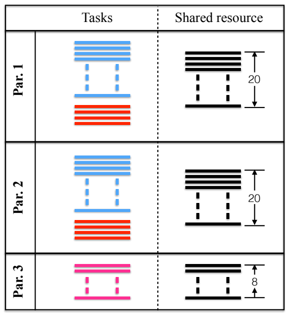
Algorithm description. The pseudocode for the *PSTPartition* algorithm is given in Algorithm 2. *Tasks are ordered and*



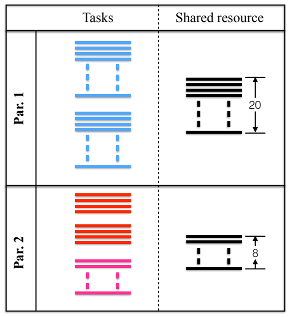
(a) 5 tasks.



(b) Naived partitioning.



(c) Improved partitioning.



(d) Parallelism-aware partitioning.

Fig. 2: Different partitioning strategies.

re-indexed according to non-increasing parallelism z_i. Then for each task τ_k in this order, we find a set of feasible allocated resource partitions (a used partition is one onto which at least one task has been assigned) onto which τ_k can be assigned (lines 4-10). We then assign τ_k to one such partition (line 12). If a task τ_k cannot be assigned to any existing partition, then the system creates a new partition with z_k units of the shared resource and assigns τ_k to it (line 14). We can similarly apply various fitting heuristics as described in Sec. 4.1.

5.2 An ILP-based Approach

To identify an optimal partition that yields a minimal shared resource size, we further formulate this problem as an ILP. The variable y_j is 1 if z_j units of the shared resources are allocated to be used; otherwise y_j is 0. The variable $x_{i,j}$ is 1 if task τ_i is allocated to these z_j shared resources; otherwise $x_{i,j}$ is 0. The following ILP formulation seeks to find the partition that yields a minimal shared resource size.

$$\text{min.} \quad \sum_{j=1}^{n} y_j z_j \tag{15a}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} x_{i,j} = 1 \qquad \forall i = 1, 2, \ldots, n \tag{15b}$$

$$x_{i,j} z_i \le y_j z_j \qquad \forall i, j = 1, 2, \ldots, n \tag{15c}$$

$$x_{j,j} = y_j \qquad \forall j = 1, 2, \ldots, n \tag{15d}$$

$$y_j \in \{0, 1\} \qquad \forall j = 1, 2, \ldots, n \tag{15e}$$

$$x_{i,j} \in \{0, 1\} \qquad \forall i, j = 1, 2, \ldots, n \tag{15f}$$

The condition in Eq. (15b) forces a task τ_i to be allocated to *exactly* one shared resource partition, and the condition in Eq. (15c) allows task τ_i to be allocated to the z_j shared resources associated to task τ_j if y_j is 1 and $z_i \le z_j$.

We can simply replace Eq. (10) with Eq. (15) in all the six different ILPs in Section 4 when $z_i > 1$ for a certain task τ_i.

6 Case Study using a GPU-based Prototype

We have fully implemented our shared-resource-centric partitioning framework on a GPU-enabled platform, where GPU is viewed as the shared resource that serves multiple tasks. The motivation behind choosing such a GPU-based case study is that GPU-based embedded systems are now pervasively used in several cyber-physical embedded systems such as autonomous driving. For example, Volvo has recently announced using the latest NVIDIA DRIVE PX2 computing engine to power a fleet of 100 Volvo XC90 SUVs starting to hit the road in 2017 year [7], [34]. In such systems, multiple real-time tasks may need to actively compete for the limited GPU resources to execute their computation-intensive components (e.g., object recognition with deep learning). GPU is suitable for our purpose because it has multiple computing units, namely stream multiprocessors (SM), while each GPU-accelerated task implemented under CUDA (a GPGPU programming model) exhibit parallelism in terms of the number of needed SMs.

Task	Para.	Period	GPU	CPU	Res. Time	Warps
τ_1	12	6.40	0.80	1.00	N/A	30
τ_2	12	25.58	1.60	3.00	20.5	
τ_3	2	12.79	1.00	2.00	N/A	
τ_4	2	12.79	0.90	2.50	N/A	
τ_5	4	19.19	1.80	2.00	17.70	
τ_6	6	7.99	1.00	1.00	N/A	
τ_7	6	8.99	0.80	2.00	N/A	
τ_8	1	11.19	0.50	2.50	N/A	

TABLE I: CPU-centric scheduling.

Through conducting this case study, we would like to answer the following research questions: ***(i)*** is it practically feasible to resolve the problem of real-time scheduling with shared resources from our counter-intuitive shared-resource-centric angle, ***(ii)*** whether solving this scheduling issue from the shared-resource-centric view is more effective than the traditional CPU-centric view, ***(iii)*** whether our proposed schedulability tests can provide reliable predictability for given task systems according to the tasks' runtime response time, and ***(iiii)*** whether our proposed partitioning approach can efficiently reduce the required size of the shared resource compared to the best practice approach.

Prototype implementation and experimental setup. We implement our prototype system in an Ubuntu Liunx system with a 4-core Intel i7-4770 CPU and an NVIDIA "Fermi" GTX 480 GPU. The GPU has 15 SMs, each containing 32 shader cores. Under the Fermi architecture, each 16 shader cores share the same pipeline and memory bandwidth and are collectively called a warp. Thus, we set our smallest computation unit to be a single warp containing 16 threads. In our case study, we target at a multi-tasking environment where we have in total eight real-time periodic tasks performing matrix-based computations. Each task has an individual thread on CPU. The GPU kernel code is only written for one thread. For each kernel, a configuration is created that indicates how many

Task	Part.	Para.	Period	GPU	CPU	Res. Time	Carry	Jitter	Mixed	Warps
τ_1	1	12	6.40	0.80	1.00	5.8	Pass	Pass	Pass	12
τ_6		6	7.99	1.00	1.00	6.7	Pass	Pass	Pass	
τ_7		6	8.99	0.80	2.00	8.0	Pass	Pass	Pass	
τ_8	2	1	11.19	0.50	2.50	10.1	Pass	Pass	Pass	6
τ_4		2	12.79	0.90	2.50	11.2	Fail	Pass	Pass	
τ_3		2	12.79	1.00	2.00	11.9	Pass	Pass	Pass	
τ_5	3	4	19.19	1.80	2.00	14	Pass	Pass	Pass	12
τ_2		12	25.58	1.60	3.00	18.3	Pass	Fail	Pass	

TABLE II: Shared-resource-centric scheduling with first-fit partitioning using constant-time (CT) schedulability tests.

Task	Part.	Para.	Period	GPU	CPU	Res. Time	Carry	Jitter	Mixed	Warps
τ_1		12	6.40	0.80	1.00	6.13	Pass	Pass	Pass	
τ_6	1	6	7.99	1.00	1.00	7.5	Pass	Pass	Pass	12
τ_7		6	8.99	0.80	2.00	8.64	Pass	Fail	Pass	
τ_8		1	11.19	0.50	2.50	10.5	Pass	Pass	Pass	
τ_4	2	2	12.79	0.90	2.50	12.2	Pass	Pass	Pass	2
τ_3		2	12.79	1.00	2.00	12.6	Fail	Pass	Pass	
τ_5	3	4	19.19	1.80	2.00	10.1	Pass	Pass	Pass	12
τ_2		12	25.58	1.60	3.00	11.8	Pass	Pass	Pass	

TABLE III: Shared-resource-centric scheduling with first-fit partitioning using TDA-based schedulability tests.

Task	Part.	Para.	Period	GPU	CPU	Res. Time	Carry	Jitter	Mixed	Warps
τ_1		12	6.40	0.80	1.00	6.03	Pass	Pass	Pass	
τ_2	1	12	25.58	1.60	3.00	17.71	Fail	Pass	Pass	12
τ_7		6	8.99	0.80	2.00	8.32	Pass	Pass	Pass	
τ_6	2	6	7.99	1.00	1.00	7.45	Pass	Pass	Pass	6
τ_5		4	19.19	1.80	2.00	16.6	Pass	Pass	Pass	
τ_3		2	12.79	2.00	2.00	12.6	Pass	Pass	Pass	
τ_4	3	2	12.79	0.90	2.50	12.2	Fail	Pass	Pass	2
τ_8		1	11.19	0.50	2.50	10.5	Pass	Pass	Pass	

TABLE IV: Shared-resource-centric scheduling with Algorithm 2 using CT schedulability tests.

Task	Part.	Para.	Period	GPU	CPU	Res. Time	Carry	Jitter	Mixed	Warps
τ_1		12	6.40	0.80	1.00	6.03	Pass	Pass	Pass	
τ_2	1	12	25.58	1.60	3.00	17.71	Pass	Pass	Pass	12
τ_7		6	8.99	0.80	2.00	8.32	Pass	Pass	Pass	
τ_6	2	6	7.99	1.00	1.00	7.45	Pass	Pass	Pass	6
τ_5		4	19.19	1.80	2.00	16.6	Pass	Pass	Pass	
τ_3		2	12.79	2.00	2.00	12.6	Pass	Pass	Pass	
τ_4	3	2	12.79	0.90	2.50	12.2	Fail	Pass	Pass	2
τ_8		1	11.19	0.50	2.50	10.5	Pass	Pass	Pass	

TABLE V: Shared-resource-centric scheduling with Algorithm 2 using TDA-based schedulability tests.

threads should be grouped together in a so-called thread block. This configuration also specifies the total number of thread blocks for the kernel. One thread block is assigned to one SM if there is enough space on that SM. In this case study, we set the thread block size to be 16, so that only one warp per thread block is used. The scheduler is written from the ground up according to the RM method. Since threads run independently, each thread contains it's own scheduler. The scheduler operates on ticks, which are approximately $0.713ns$ each. Although the latest GPU architectures such as Pascal enables limited preemptive execution on GPU, the GPU used in our case study is of the Fermi architecture with a non-preemptive GPU execution model. To enable limited preemptive scheduling on GPU, we leverage our previously developed GPGPU runtime module GPES [43], which indirectly enables limited preemptive GPGPU execution through breaking both computation and data into fine-grained sub-chunks. Thus, a higher-priority task can almost immediately preempt a lower-priority task on GPU through waiting for a rather small non-preemptive period.

Scheduling algorithms. Given the relatively high complexity of the ILP-based solutions, we choose to evaluate the practical efficacy of the fastest partitioning algorithm we develop, i.e., first-fit partitioning and Algorithm 2, combined with the constant-time efficient schedulability tests and TDA-based tests. We evaluate these two partitioning solutions also because the main goal of conducting this case study is to validate the practicality and efficiency of our shared-resource-centric scheduling methodology, but not to compare the performance of every single individual partitioning solutions (e.g., best-fit and worst-fit) we develop that exhibit different runtime complexity (instead we achieve this evaluation goal in Sec. 7 through conducting extensive sets of simulations). We use RM to prioritize and schedule tasks on GPU. Moreover, since the Intel i7 CPU has four physical cores, each supporting two threads, each of the eight tasks does not interfere with each other (significantly) while executing on logical CPU in our settings. This can be seen in the experimental results shown in

Tables I- V, where the execution time of each task on CPU is identical. The time unit in the tables is Millisecond (ms). For the traditional CPU-centric scheduling approach, since there is no contention on CPU for each task, we directly run the eight tasks on CPU without implementing any prioritization or partitioning strategy to control their execution, and simply let these tasks compete for GPU resources at runtime. For the shared-resource-centric scheduling approach, we evaluate our proposed partitioning algorithm (Algorithm 2) that seeks to minimize the units of shared resource required to schedule all eight tasks, and the first-fit partitioning algorithm (tasks indexed according to RM). Under these two shared-resource-centric partitioning approaches, the number of warps granted to execute tasks assigned to each partition equals the largest number of warps requested by any task in that partition.

Schedulability tests. We seek to validate the predictability of the schedulability tests provided in Sec. 4.1. Under first-fit partitioning (results are shown in Table II and Table III) and *PSTPartition* (results are shown in Table IV and Table V), we perform both constant-time schedulability tests and TDA-based test in each partition to validate the schedulability for each task. Constant-time schedulability tests under first-fit partitioning include **ST-FF-CT(Carry)**, **ST-FF-CT(Jitter)**, and **ST-FF-CT(Mixed)**; while TDA-based tests include **ST-FF-TDA(Carry)**, **ST-FF-TDA(Jitter)**, and **ST-FF-TDA(Mixed)**. For *PSTPartition*, we adopt Lemma 3 (**Carry**), Lemma 4 (**Jitter**), and both (**Mixed**) as constant-time schedulability tests, and adopt Eq. 5a (**Carry**), Eq. 6 (**Jitter**) and both (**Mixed**) as TDA-based schedulability tests.

Results. The results are collected and shown in Tables I- V. For the five tables, the "part." column denotes task partitions given by the corresponding partition algorithm, the "Para." column denotes the number of warps required by each task; the "Period" column denotes the task period; the "GPU" column denotes the execution time on GPU of each task; the "CPU" column denotes the execution time on CPU of each task; the "Res. Time" column denotes the longest runtime response time of each task; the "Carry", "Jitter", "Mixed" columns denote the corresponding schedulability tests as described above; the "Warps" column denotes the number of warps needed by each task under the corresponding approach. Note that the GPU used in this case study has 30 warps in total.

As seen in Table I, under the traditional CPU-centric scheduling approach, even if we use all the 30 warps, there are still six tasks that cannot meet their deadlines, implying that the GPU resource is far from sufficient to feasibly schedule this task set. On the other hand, under shared-resource-centric first-fit partitioning, as seen in Table II and Table III, we performed the constant-time schedulability tests and TDA-based tests to validate whether each task is schedulable or not in a partition. As seen in Table II, three partitions are created (shown by the divider line) under constant-time schedulability tests, each requiring a total number of wraps of 12, 6, and 12, respectively. Thus, this approach requires the entire GPU resource and can successfully schedule all eight tasks with their deadlines being met at runtime (i.e., tasks' response time is smaller than their periods). Note that τ_4 is not schedulable in partition 2 when we test τ_4 using **Carry**. However, if τ_4 is tested by **Jitter**, τ_4 becomes schedulable. A similar observation is shown in partition 3. τ_2 passes the schedulability test under **Carry** but fails under **Jitter**. This implies that there does not exist a domination relationship between the **Carry** and the **Jitter** tests. As seen in Table III, three partitions are created under TDA-based schedulability tests, each requiring a total number of wraps of 12, 2, and 12, respectively. This approach requires 26 warps and successfully schedule all eight tasks with their deadlines being met at runtime, which requires 4 warps less than the previous approach. This implies that, under the same partitioning method, TDA-based schedulability tests may admit more tasks than constant-time schedulability tests, yet at the cost of exhibiting higher time complexity.

Table IV and Table V show the results under our developed partitioning algorithm, which create three partitions but with different task assignments. Note that in our case study the partitions under constant-time schedulability tests and TDA-based schedulability tests are the same using *PSTPartition*. Our partitioning algorithm successfully schedules all eight tasks with their deadlines being met. As seen, the three partitions require a total number of warps of 12, 6 and 2, respectively, thus resulting in a significantly reduced number of total required units of the shared resource, i.e., merely 20 warps. As discussed in Sec. 5.1, this is because *PSTPartition* seeks to minimize the required units of the shared resource through considering tasks' parallelism parameters. As seen in Table IV and V, *PSTPartition* seeks to assign tasks with the same parallelism (starting from the task with the maximum parallelism) into the same partition as long as the schedulability of tasks in that partition can be maintained. Doing so masks the negative impact due to accommodating other tasks in that partition which exhibit same or smaller parallelism.

Our GPU-based case study answers the four early-raised research questions: the shared-resource-centric scheduling methodology can be practically and efficiently implemented in practice, which is superior to the CPU-centric scheduling view. Through incoporating carefully designed schedulability tests and parallelism-aware partitioning algorithms, we achieve much better performance in terms of both schedulability and minimum required size of the shared resource.

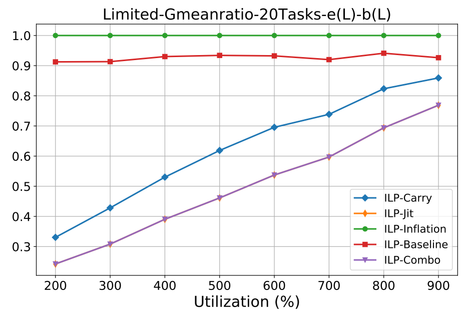
7 Simulations

We have conducted extensive simulations using synthesized task sets to evaluate our proposed approaches. We analyze the gathered results which are classified into three groups: 1) ILP-based approaches (ILP-Carry, ILP-Jitter, ILP-Inflation, ILP-Combo) 2) PSTPartition with TDA-based Tests, and 3) PSTPartition with Constant Time Tests. We consider first-fit strategies here due to the space limitation. Extensive results for other fitting strategies can be found in [13]. A comparison on different fitting strategies is given in the appendix.

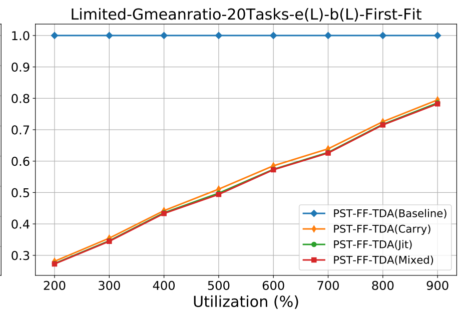
The compared results are based on the normalization on the geometric mean of the required number of shared resource size derived by the aforementioned 13 approaches, respectively, with respect to a given goal of task set utilization. The normalized geometric mean is calculated as the geometric mean on the required number of shared resource size of the resulting partition divided by the maximum required number of shared resource size, i.e., $\sum_{i=1}^{n} z_i$.

7.1 Simulation Setup

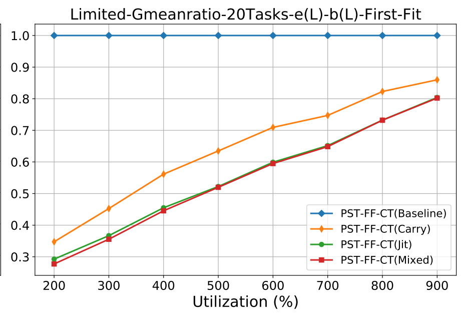
We generated synthesized implicit-deadline sporadic task sets, analyzing task sets of 4 different cardinalities, i.e.,



(a) ILP



(b) TDA



(c) Constant Time

Fig. 3: Different approaches for 20 tasks with etype=L and btype=L.

$10, 20, 30$, and 40 tasks. The UUnifast-Discard method proposed by Emberson et al. [20] was applied to generate a set of utilization values with the given utilization, where u_i was defined as s_i/p_i. Let $\mathbb{B}_i = \sigma_i \times B$ and . For each task τ_i, z_i was randomly drawn from $\{1, 2, 4, 6, 8, 10\}$ and e_i was set to one of three ranges depending upon the given type (etype): $[0.01(p_i - s_i - \mathbb{B}_i), 0.1(p_i - s_i - \mathbb{B}_i)]$ (etype = S), $[0.1(p_i - s_i - \mathbb{B}_i), 0.3(p_i - s_i - \mathbb{B}_i)]$ (etype = M), and $[0.3(p_i - s_i - \mathbb{B}_i), 0.45(p_i - s_i - \mathbb{B}_i)]$ (etype = L). The task periods were randomly distributed over two orders of magnitude. For each of these in total $3 \cdot 3 \cdot 4 \cdot 13 = 468$ settings, e.g., etypes, btypes, cardinalities, and approaches, we recorded 100 synthesized task sets for each utilization $U^s_{sum} \in [10 \cdot \ n\%, 45 \cdot \ n\%]$ (step size $5 \cdot \ n\%$). We used the Gurobi library [4] to implement the proposed ILPs. We adopted six machines in our local cluster which has eight 64-bit Intel processors running at 2.0 GHz with 16GB DDR3 RAM. For the 468 settings, it required seven hours to derive the results per machine.

7.2 Results and Discussions

Due to the high complexity of the ILP approaches, we can only get very few results with ILP-based approaches when the cardinalities of the task set were 30 and 40 within the limited amount of time. For the sake of fairness, we uniformly present the case that the cardinality of task sets is 20 with etype = L, btype = L, and $z_i \geq 1$ in the following comparisons. Due to the page limitation, all the other cases are presented in [13].

Comparison among different ILP-based approaches. Fig. 3a shows the normalized geometric means of five different ILP-based approaches , i.e, ILP-Carry, ILP-Jit, ILP-Inflation, ILP-Baseline, and ILP-Combo. When the total utilization U^s_{sum} becomes larger, the differences among all the ILP-based approaches become smaller. We can see that ILP-Combo outperforms the other ILP approaches and the normalized geometric mean of ILP-Jit is exactly the same as ILP-Combo in this case. Interestingly, we can see that ILP-Inflation is even worse than ILP-Baseline which directly considers e_i as a part of s_i. In this group, we choose the two extreme cases, i.e., ILP-Inflation and ILP-Combo, as the representative approaches.

Comparison with PSTPartition using TDA-based Tests. Fig. 3b shows the normalized geometric means of PSTPartition with four different TDA-based schedulability tests, i.e, TDA-Baseline, TDA-Carry, TDA-Jit, and TDA-Mix. The results are very similar for all TDA-based schedulability tests. Since ST+TDA(Mix) is the mixture approach which checks Lemmas 1, and 2, it always outperforms the others. In this group, we pick ST+TDA(Baseline) and ST+TDA(mix) as the representative approaches.

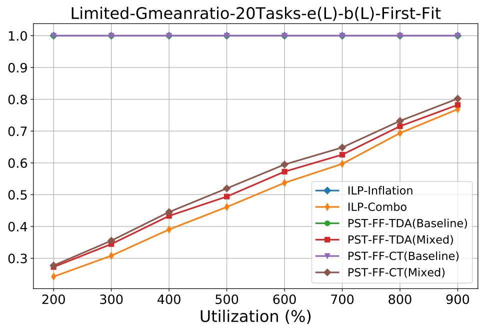


Fig. 4: Representative approaches for 20 tasks with etype=L and btype=L.

Comparison with PSTPartition with Constant Time Tests. Fig. 3c shows the normalized geometric means of PSTPartition with four different constant time schedulability tests, i.e, CT-Baseline, CT-Carry, CT-Jit, and CT-Mix. The trends are in general similar to the previous comparisons. PSTPartition using CT(Jit) has the results closest to PSTPartition using CT(mix). Again, since ST+CT(Mix) is the mixture approach with the advantage of three constant time tests, it can always outperform the others. In this group, we pick ST+CT(Baseline) and ST+CT(mix) as the representative approaches.

Representative Approaches Comparison. In Fig. 4, six different approaches are compared, but three of them are identical, including ILP-Inflation and PSTPartition with TDA(Baseline) and CT(Baseline) approaches, which are the maximum required number of shared resource size. Although ILP-Combo outperforms the others, it suffers from its computationally expensive overhead. When the number of tasks becomes 30, it is not available to use ILP to obtain the resource partition. We notice that the trade-off between the time complexity of schedulability tests and the minimization of required shared resource sizes are clear, while the derived results of TDA(mix) and CT(mix) are really close. We can conclude that the proposed heuristic PSTPartition with CT(mix) can provide the significant results in an efficient manner.

According to the above-discussed experimental results, we believe that a representative set of our developed partitioning algorithms and the corresponding schedulability tests can be efficiently applied in various practical scenarios with varying affordable runtime overhead. With different btypes or different etypes, we observe that the results are just slightly different to the presented results in the paper with the similar trends.

8 Conclusion

In this paper, we study the problem of scheduling HRT sporadic tasks that may access CPU cores and a shared resource. We resolve this problem from a counter-intuitive shared-resource-centric perspective, and develop a rather comprehensive set of suspending task partitioning algorithms that partition tasks onto the shared resource with the objective of guaranteeing schedulability while minimizing the required size of the shared resource. A GPU-based prototype case study and extensive simulation-based experiments have been conducted, which validate both our shared-resource-centric scheduling philosophy and the efficiency of our suspension-based partitioning solutions in practice.

Appendix

Proof of Lemma 3.

Proof: Under limited preemptive task scheduling, the execution of τ_k may be delayed by two sets of tasks:(*i*) higher priority tasks that may preempt τ_k; (*ii*) lower priority tasks that may block τ_k. Similar to the proof of Lemma 1 given in [11], the carry-in workload of a higher-priority task τ_i can be safely upper bounded by $\left(\left\lceil \frac{t}{p_i} \right\rceil + 1\right) s_i$. And according to Fact 1, the blocking time due to lower priority tasks is at most $\sigma_k \times B$. Thus, a task τ_k with a implicit deadline can be feasibly scheduled under fixed-priority if the following equation holds:

$$\exists t \mid 0 < t \leq p_k, \quad s_k + e_k + \sigma_k \times B + \sum_{\tau_i \in hp(k)} \left(\left\lceil \frac{t}{p_i} \right\rceil + 1\right) s_i \leq t.$$

Inuitively, we simply add the extra blocking time $\sigma_k \times B$ due to the limited preemptive model into the original schedulability test designed for the fully preemptive fixed-priority schedulability test [11]. ■

Proof of Lemma 4.

Proof: Under limited preemptive task scheduling, the execution of τ_k may be delayed by two sets of tasks:(*i*) higher priority tasks that may preempt τ_k; (*ii*) lower priority tasks that may block τ_k. Similar to the proof of Lemma 2 in the document [11], the carry-in workload of a higher-priority task τ_i can be safely upper bounded by $\left\lceil \frac{t+p_i-s_i}{p_i} \right\rceil s_i$. And according to Fact 1, the blocking time from lower priority tasks is at most $\sigma_k \times B$. Thus, it results that a task τ_k with a implicit deadline can be feasibly scheduled under fixed-priority if the following equation holds:

$$\exists t \mid 0 < t \leq p_k, \quad s_k + e_k + \sigma_k \times B + \sum_{\tau_i \in hp(k)} \left\lceil \frac{t + p_i - s_i}{p_i} \right\rceil s_i \leq t.$$

Inuitively, we simply add the extra blocking time $\sigma_k \times B$ due to the limited preemptive model into the original schedulability test designed for the fully preemptive fixed-priority schedulability test [11]. ■

Proof of Lemma 6.

Proof: This is based on a safe linear approximation of the schedulability test in Lemma 4 as follows:

$$\left\lceil \frac{t + p_i - s_i}{p_i} \right\rceil s_i \leq \left(\frac{t + p_i - s_i}{p_i} + 1 \right) s_i$$

By solving $e_k + s_k + \sigma_k \times B + \sum_{\tau_i \in hp(k)} \left(\frac{t+p_i-s_i}{p_i} + 1 \right) s_i = t$ under the condition $\sum_{\tau_i \in hp(k)} \frac{s_i}{p_i} < 1$, we know that task τ_k can meet its deadline if

$$t = \frac{e_k + s_k + \sigma_k \times B + \sum_{\tau_i \in hp(k)} \left(2s_i - \frac{s_i^2}{p_i} \right)}{1 - \sum_{\tau_i \in hp(k)} \frac{s_i}{p_i}} \leq p_k \quad (16)$$

The condition in Eq. (16) can be rewritten as in Eq. (6). ■

Proof of Lemma 7.

Proof: This comes from the test in Eq. (5b). The left-hand side of Eq. (5b) is maximized when we consider the whole task set, and the right-hand side of Eq. (5b) is minimized when we take $\ln \left(\frac{3}{2 + \max_{\tau_i \in \tau} \frac{s_i + e_i + \sigma_i \times B}{p_i}} \right)$. ■

Proof of Theorem 1.

Proof: We prove this theorem by contradiction. Suppose that Algorithm *STPartition* fails to assign task τ_k to the first m resources for contradiction. By the assumption that a single task is always schedulable on one shared resource, we know that $k > m$. Since the test in Eq. (5a) is adopted, we know that $\left(\frac{s_k+e_k+\sigma_k \times B}{p_k} + 2\right) \prod_{\tau_i \in \pi_j} (1 + u_i^s) > 3$ for $j = 1, 2, \ldots, m$ before task τ_k is allocated. Since $\cup_{j=1}^{m} \pi_j$ is equal to $hp(k)$, we have

$$\left(\frac{s_k + e_k + \sigma_k \times B}{p_k} + 2 \right)^m \prod_{\tau_i \in hp(k)} (1 + u_i^s) > 3^m$$

$$\Rightarrow \prod_{\tau_i \in hp(k)} (1 + u_i^s) > \left(\frac{3}{\frac{s_k + e_k + \sigma_k \times B}{p_k} + 2} \right)^m$$

$$\Rightarrow e^{\sum_{\tau_i \in hp(k)} u_i^s} > \left(\frac{3}{\frac{s_k + e_k + \sigma_k \times B}{p_k} + 2} \right)^m$$

$$\Rightarrow U_{sum} \geq \sum_{i=1}^{k-1} u_i^s > m \ln \frac{3}{2 + \frac{s_k + e_k + \sigma_k \times B}{p_k}}$$

Hence, we reach the contradiction. ■

Proof of Theorem 2.

Proof: We prove this theorem by contradiction. Suppose that Algorithm *STPartition* fails to assign task τ_k to the first m resources for contradiction. By the assumption that a single task is always schedulable on one shared resource, we know that $k > m$. Since the test in Eq. (6) is adopted, we know that $\frac{e_k+s_k+\sigma_k \times B}{p_k} + \frac{\sum_{\tau_i \in \pi_j} \left(2s_i - \frac{s_i^2}{p_i}\right)}{p_k} + \sum_{\tau_i \in \pi_j} \frac{s_i}{p_i} > 1$ for $j = 1, 2, \ldots, m$ before task τ_k is allocated. Since $\cup_{j=1}^{m} \pi_j$ is equal to $hp(k)$, we have

$$m \left(\frac{e_k + s_k + \sigma_k \times B}{p_k} \right) + \frac{\sum_{\tau_i \in hp(k)} \left(2s_i - \frac{s_i^2}{p_i} \right)}{p_k} + \sum_{\tau_i \in hp(k)} \frac{s_i}{p_i} > m$$

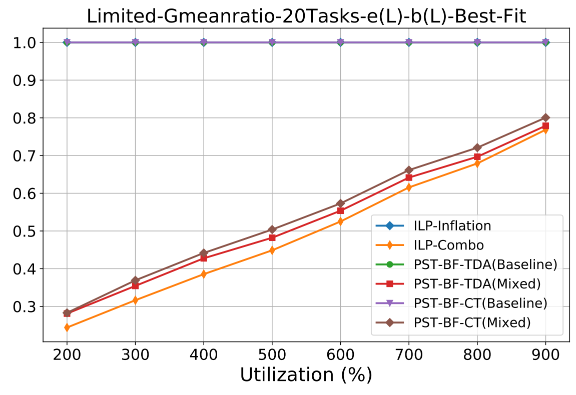
Due to the RM ordering of the tasks, we then have

$$m \left(\frac{e_k + s_k + \sigma_k \times B}{p_k} \right) + \sum_{\tau_i \in hp(k)} \frac{2s_i}{p_i} + \sum_{\tau_i \in hp(k)} \frac{s_i}{p_i} > m$$

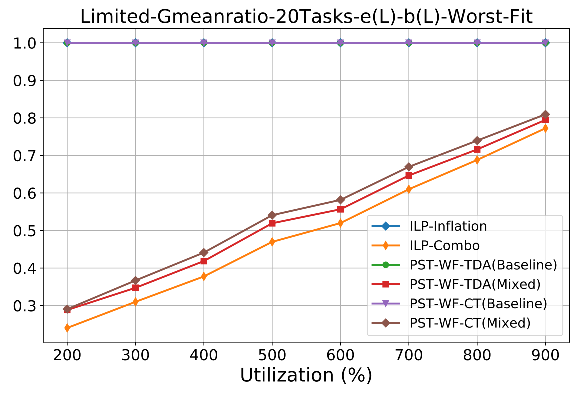
Hence, we reach the contradiction. ■

Comparisons of Fitting Strategies on ST/PSTPartition. When adopting Algorithm 1 and 2, the fitting strategies may affect the performance of the partition algorithms. As presented in Section 4.1, there are several fitting strategies, i.e., First-Fit (FF), Best-Fit (BF), and Worst-Fit (WF). For First-Fit, the feasible resource j with the smallest index will be chosen. For Best-Fit, the feasible resource j with the maximum total utilization of tasks in π_j will be chosen, whereas the feasible resource j with the minimum total utilization of tasks in π_j will be chosen for Worst-Fit. Since the results derived by both algorithms are similar, we only present the impact of different fitting strategies on PSTPartition.

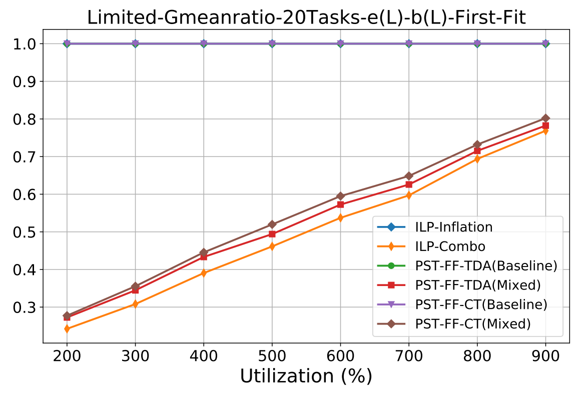
As shown in Fig 5, the derived results by using Best-Fit outperforms using Worst-Fit. Comparing to First-Fit, the derived normalized geometric mean by using Worst-Fit strategy is also worse. However, the trends among three different strategies are similar and the results are still close. Overall, we can conclude that Best-Fit is the most suitable strategy to use when we adopt Algorithms 1 and 2.



(a) Best-Fit



(b) Worst-Fit



(c) First-Fit

Fig. 5: Different fitting strategies for 20 tasks with etype=L and btype=L.

References

[1] Memory preemption. https://www.ibm.com/support/knowledgecenter/SSETD4_9.1.2/lsf_admin/resource_preemption_memory_example.html.

[2] Nvidia tesla p100 whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[3] Preemptable and nonpreemptable resource - OS. http://maulik245.blogspot.com/2010/12/preemptable-and-nonpreemptable-resource.html.

[4] Gurobi 7.0. https://www.gurobi.com/, 2017.

[5] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In P. R. D'Argenio and H. Melgratti, editors, *CONCUR 2013 – Concurrency Theory*, volume 8052 of *Lecture Notes in Computer Science*, pages 25–43. 2013.

[6] S. Altmeyer, R. I. Davis, L. S. Indrusiak, C. Maiza, V. Nélis, and J. Reineke. A generic and compositional framework for multicore response time analysis. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 129–138, 2015.

[7] Autonews. Carmakers tap nvidia's supercomputer to self-driving, 2016. http://europe.autonews.com/article/20160607/ANE/160609921/carmakers-tap-nvidias-supercomputer-to-make-leap-toward-self-driving.

[8] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable fpgas. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 1–12. IEEE, 2016.

[9] J.-J. Chen, W.-H. Huang, Z. Dong, and C. Liu. Fixed-priority scheduling of mixed soft and hare real-time tasks on multiprocessors. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2017 IEEE 23rd International Conference on*, pages 1–10. IEEE, 2017.

[10] J.-J. Chen, W.-H. Huang, and C. Liu. k2u: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium, 2015 IEEE*, pages 107–118. IEEE, 2015.

[11] J.-J. Chen, G. Nelissen, and W.-H. Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[12] J.-J. Chen, G. von der Bruggen, W.-H. Huang, and C. Liu. State of the art for scheduling and analyzing self-suspending sporadic real-time tasks. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 1–10, 2017.

[13] K.-H. Chen, J. Shi, and J.-J. Chen. Comparision with different approaches, cardinalities, btype, and etype . http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/results.tar.gz, 2017.

[14] S.-W. Cheng, J.-J. Chen, J. Reineke, and T.-W. Kuo. Memory bank partitioning for fixed-priority tasks in a multi-core system. In *RTSS*, December 2017.

[15] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the intel scc many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525–532. IEEE, 2011.

[16] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 257–270. IEEE, 2006.

[17] D. M. Dhamdhere. *Systems Programming and Operating Systems*. Tata McGraw-Hill, 1999.

[18] Z. Dong and C. Liu. Closing the loop for the selective conversion approach: A utilization-based test for hard real-time suspending task systems. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 339–350. IEEE, 2016.

[19] Z. Dong, C. Liu, A. Gatherer, L. McFearin, P. Yan, and J. H. Anderson. Optimal dataflow scheduling on a heterogeneous multiprocessor with reduced response time bounds. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[20] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[21] D.-R. Fan, N. Yuan, J.-C. Zhang, Y.-B. Zhou, W. Lin, F.-L. Song, X.-C. Ye, H. Huang, L. Yu, G.-P. Long, et al. Godson-t: An efficient many-core architecture for parallel program executions. *Journal of Computer Science and Technology*, 24(6):1061, 2009.

[22] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Resource sharing protocols for real-time task graph systems. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 272–281. IEEE, 2011.

[23] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 387–397. IEEE, 2009.

[24] J.-J. Han, D. Zhu, X. Wu, L. T. Yang, and H. Jin. Multiprocessor real-time systems with shared resources: Utilization bound and mapping. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):2981–2991, 2014.

[25] W.-H. Huang, J.-J. Chen, and J. Reineke. MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 158:1–158:6, 2016.

[26] W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu. Pass: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Proceedings of the 52nd Annual Design Automation Conference*, page 154. ACM, 2015.

[27] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[28] H. Kim and R. Rajkumar. Real-time cache management for multi-core virtualization. In *Embedded Software (EMSOFT), 2016 International Conference on*, pages 1–10. IEEE, 2016.

[29] K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, and N. Stoimenov. A formal approach to the wcrt analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Systems*, 50(5-6):736–773, Nov. 2014.

[30] C. Liu and J.-J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 173–183. IEEE, 2014.

[31] J. W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, 1st edition, 2000.

[32] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 87–96, 2015.

[33] NASA. The altair lunar lander. In *Fact sheet FS-2008-09-007-JSC, National Aeronautics and Space Administration*, 2008.

[34] NVIDIA. Nvidia accelerates race to autonomous driving at ces, 2016. https://blogs.nvidia.com/blog/2016/01/04/drive-px-ces-recap/.

[35] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE*, pages 741–746, March 2010.

[36] R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–12. IEEE, 2016.

[37] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, pages 213–222, April 2011.

[38] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs, 2014.

[39] F. Thabet, Y. Lhuillier, C. Andriamisaina, J.-M. Philippe, and R. David. An efficient and flexible hardware support for accelerating synchronization operations on the sthorm many-core architecture. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 531–534. EDA Consortium, 2013.

[40] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–12. IEEE, 2016.

[41] C. Y. Villalpando, A. E. Johnson, R. Some, J. Oberlin, and S. Goldberg. Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander. In *Aerospace Conference, 2010 IEEE*, pages 1–9. IEEE, 2010.

[42] H. Zhou and C. Liu. Task mapping in heterogeneous embedded systems for fast completion time. In *Proceedings of the 14th International Conference on Embedded Software*, page 22. ACM, 2014.

[43] H. Zhou, G. Tong, and C. Liu. GPES: a preemptive execution system for gpgpu computing. In *RTAS*, pages 87–97, 2015.