Semi-Partitioned Scheduling for Resource-Sharing Hard-Real-Time Tasks*

Mayank Shekhar
Southern Illinois University Carbondale
mayank@siu.edu

Harini Ramaprasad
Southern Illinois University Carbondale
harinir@siu.edu

Frank Mueller
North Carolina State University
mueller@cs.ncsu.edu

ABSTRACT

As real-time embedded systems integrate more and more functionality, they are demanding increasing amounts of computational power that can only be met by deploying them on powerful multi-core architectures. Efficient task allocation and scheduling on such architectures is paramount. Multi-core scheduling algorithms for independent real-time tasks has been the focus of much research over the years. However, in practice, tasks typically share software resources among each other. One of the foremost bottlenecks in successfully scheduling resource sharing tasks on multi-core architectures is the blocking times, especially remote blocking, that tasks may suffer from. In this paper, we present a novel semi-partitioned scheduling algorithm that significantly reduces blocking times of tasks by splitting a task into subtasks based on resource usage and executing resource independent and resource sharing subtasks on mutually exclusive cores. We demonstrate the effectiveness of our algorithm and evaluate it alongside the classic Distributed Priority Ceiling Protocol (DPCP) that uses a similar approach with "synchronization" cores and alongside a recent partitioned scheduling approach called Greedy Slacker. Results demonstrate that our algorithm achieves a higher scheduled utilization in a majority of task sets, with average improvements in the range of 10% to 15% over DPCP and Greedy Slacker.

1. INTRODUCTION

The need for increasing performance requirements under lower power/energy budgets or greener computing constraints has led to the advent of multi-core architectures for use in general purpose systems. As modern real-time and embedded systems continue to integrate more and more functionality, they are also demanding increasing amounts of computation that can only be satisfied by the adoption of multi-core architectures. However, due to their strict predictability requirements, several challenges need to be addressed before real-time systems can be safely deployed on multi-core architectures. This has been the focus of much research over the last several years.

In order to effectively use the processing power of a multi-core platform, efficient scheduling of tasks is paramount. Researchers have proposed numerous approaches for scheduling of real-time tasks on multi-core architectures, ranging from partitioned to global scheduling approaches and hybrid semi-partitioned scheduling approaches [6]. In practical real-time systems, tasks may share software resources (e.g., shared variables). The use of such shared resources must be appropriately arbitrated and the arbitration rules have a significant impact on task scheduling and task set schedulability. In recent years, researchers have proposed resource arbitration policies and task scheduling and schedulability techniques for multi-core architectures [1, 8, 13, 12, 17].

In the context of resource-sharing tasks executing on multi-core architectures, the main bottleneck or limiting factor for task set schedulability is the *blocking* that tasks may experience due to shared resources. Even though there may be available cores on the platform (for example, in many-core architectures with dozens of cores), algorithms are unable to utilize them to improve schedulability due to high blocking times, especially remote blocking times. Recently, researchers have proposed task allocation and scheduling approaches specifically targeted towards reducing blocking times of tasks, thus improving task set schedulability [12, 17].

Contributions: We propose a semi-partitioned scheduling scheme for resource-sharing tasks that strives to significantly reduce blocking times and improve task set schedulability. The fundamental idea is to dedicate separate cores for the execution of resource-sharing regions belonging to each shared resource and to employ a partially non-work-conserving scheduling approach to obtain tighter schedulability bounds. While our approach uses a larger number of cores and introduces data migration, we demonstrate that it has a considerable potential to improve schedulability due to reduction in blocking times. In contrast to most existing work on scheduling for resource sharing real-time tasks, we also explicitly consider architectural overheads introduced due to data migration. In the majority of experiments conducted, our algorithm outperforms existing work by a significant factor.

The rest of this paper is organized as follows. Section 2 discusses background information and related work. Section 3 presents our assumptions and system model. In Section 4, we present the details of our semi-partitioned scheduling

*This work was supported in part by NSF grants CNS-0905212, CNS-0720496 and CNS-0905181.

algorithm. Section 5 discusses architectural considerations and Section 6 presents a discussion on the limitations and the complexity of our algorithm. In Section 7, we discuss the details of a closely related work that we compare our algorithm with. Section 8 presents our experimental setup and results. We present our conclusions in Section 9.

2. BACKGROUND AND RELATED WORK

In this section, we present relevant background information and discuss related work in real-time scheduling on multi-core architectures. Real-time scheduling techniques on multi-core architectures may be broadly classified into three categories, namely partitioned, global and semi-partitioned approaches. In partitioned scheduling, tasks are statically allocated to cores and scheduled using single-core scheduling algorithms on each core. In global scheduling, tasks are placed in a common queue and dynamically scheduled on cores, leading to task migrations. Semi-partitioned scheduling is a hybrid approach where some tasks are partitioned and others are allowed to migrate, typically in a predetermined manner.

Schemes for Independent Tasks. Several real-time scheduling techniques have been proposed for tasks that are independent of each other, including partitioned, global and semi-partitioned approaches. Davis and Burns survey several such algorithms [6].

Schemes for Resource Sharing Tasks. Rajkumar et al. presented two extensions of the PCP for multiprocessor real-time systems under partitioned static priority scheduling, namely the DPCP or distributed priority ceiling protocol, used for distributed memory multi-core architectures, and the MPCP or multiprocessor priority ceiling protocol, used for shared memory and global semaphores [13]. DPCP is similar to ours in its use of separate cores for execution with and without shared resources. Hence, we compare our algorithm with it in our evaluation.

There have been multiprocessor synchronization protocols proposed for partitioned EDF scheduling [7]. In recent work, Block et al. proposed a Flexible Multi-processor Locking Protocol (FMLP) [1] that may be used with both static-priority and dynamic-priority scheduling schemes. Lozi et al.[11] proposed a mechanism by which critical sections may be locked on remote cores based on task profiling.

Lakshmanan et al. proposed a synchronization-aware partitioning algorithm (SPA) for resource sharing tasks [8]. Nemati et al. proposed a blocking-aware partitioning algorithm (BPA) [12] that achieved improvements over SPA. Wieder and Brandenburg proposed a heuristic based resource-aware partitioning algorithm termed the Greedy Slacker [17] that demonstrated significant improvements over BPA. In this algorithm, Audsley's optimal priority assignment scheme [3] is used to assign a static priority to the tasks on each core and the algorithm uses response time analysis to determine schedulability of a task. We explain the Greedy Slacker algorithm in more detail in Section 7 and compare our algorithm with it in Section 8.

Brandenburg [5] also proposed a linear programming based method to optimize total blocking incurred on a task, when it is scheduled using partitioned, fixed-priority scheduling and any resource sharing protocol. The authors demonstrated improvement in blocking time under both DPCP and MPCP. Another recently proposed work by Weider and Brandenburg [18] performed extensive blocking time analysis caused due to spin locks. Once again, this paper uses linear programming to optimize blocking time calculation. Unlike our current paper, neither of these papers consider architectural factors such as cache effects and migration overheads. In order to include such factors, the objective functions in these papers require extension. Such extension and subsequent comparison is out of the scope of our current paper and will be performed as part of future work.

In recent work, Afshar et al. propose a synchronization mechanism and associated schedulability analysis for resource sharing tasks scheduled using a semi-partitioned approach [2]. This work assumes that at most one subtask of a given task can be scheduled on a given core. As will be explained in Section 4, our scheme requires support for scheduling multiple subtasks of a given task on the same core. Hence, the work proposed by Afshar et al. is inapplicable in our context.

3. ASSUMPTIONS AND SYSTEM MODEL

3.1 Task Model

In this work, we assume a sporadic hard-real-time task model where the relative deadline of every task is less than or equal to its minimum inter-arrival time (hereafter called its period). We assume that tasks may share *software* resources among each other. Regions of a task where shared resources are used are referred to as resource sharing regions or critical sections. We assume that critical sections do not have nested resource usage.

We assume that tasks are split on the basis of their critical sections, i.e., a task is split into resource independent and resource sharing subtasks, and that the two types of subtasks are executed on mutually exclusive cores. Resource independent subtasks are executed in a preemptive manner under a static priority assignment. Resource sharing subtasks are assumed to be scheduled using a non-preemptive, static-priority scheduling policy.

3.2 Architectural Model

We assume a homogeneous multi-core architecture with a Network-on-Chip (NoC) for communication. Each core is assumed to have a lockable, set-associative private cache and it is assumed that there are no shared caches. We assume that separate channels are available in the NoC for communication between cores and main memory and for communication among cores, as is found in Tilera's TilePro64 architecture [16]. Main memory traffic is assumed to be arbitrated using a weighted TDMA scheme proposed in prior work [15] that makes memory access times independent of the physical location of cores. A TDMA approach is also used for core to core traffic in order to be able to bound migration overhead.

3.3 Cache Locking and Migration Model

We assume that every subtask may *statically* lock a subset of its own (non-shared) memory footprint in the cache on its core and that every subtask is allowed to lock at most one way in each cache set. Lines that are not locked in the cache are assumed to be fetched from the main memory. Every shared resource is assumed to be locked in an exclusive cache way on the core to which subtasks using that resource are allocated.

A given task may use some of its private data in both its resource independent and its resource sharing subtasks.

Symbol	Description
τ_i	Sporadic task with index i
T_i, C_i, D_i	Period, WCET and Relative Deadline of task τ_i
ρ_p	Shared resource with index p
$C_i(\rho_m)$	WCET of critical section of task τ_i using resource ρ_m
$\tau_{i,j}$	j^{th} subtask of task τ_i
$\phi_{i,j}$	Phase of subtask $\tau_{i,j}$ relative to phase of task τ_i
$C_{i,j}$	WCET of subtask $\tau_{i,j}$
$\rho_{i,j}$	Shared resource used by subtask $\tau_{i,j}$
$\mathbf{T}_i^c$	Set of subtasks of task τ_i on core c
C_i^c	Total WCET of task τ_i on core c (i.e.,$\sum$WCETs in $\mathbf{T}_i^c$)
v_i^c	Representative virtual task for subtasks in $\mathbf{T}_i^c$
Tv_i^c, Cv_i^c	Period and WCET of virtual task of v_i^c

Table 1: Symbols and Terminology

Since resource independent and resource sharing subtasks are executed on different cores, respectively, any cached private data needs to be migrated among these cores. In this work, we employ a proactive, push-based cache migration scheme developed in prior work [14] to migrate data among parent and critical cores.

Note that 1) the choice of memory lines to be locked for a given subtask is an orthogonal problem that is out of the scope of the current work; and 2) dynamic loading and locking of different memory lines belonging to a single subtask whose memory footprint is larger than one cache way may be allowed as long as every subtask uses a predetermined, exclusive subset of cache lines and accounts for loading/reloading costs in its own WCET. However, in practice, the use of dynamic loading and locking may result in prohibitive increases in blocking times. In this context, reserving more than one cache way for a shared resource may be a better option.

4. METHODOLOGY

In this section, we present the details of our algorithm to allocate and schedule resource sharing tasks on a multi-core architecture. The symbols and terminology that are used in the remainder of this paper are briefly described in Table 1. The need and usage of symbols will be explained where required in the rest of this section.

The fundamental goal of our algorithm is to reduce blocking times due to resource sharing and improve task set schedulability. To this end, we first divide tasks into groups such that no resource is shared by tasks in different groups. For instance, if τ_i uses shared resources ρ_p and ρ_q, τ_j uses shared resources ρ_q, ρ_r and τ_k uses shared resources ρ_r and ρ_s, then τ_i, τ_j and τ_k are all members of the same group even though there is no direct sharing between τ_i and τ_k. Groups are scheduled onto cores, starting with the group with highest utilization. Each group of tasks is assumed to be scheduled on an exclusive set of cores in order to prevent interference across groups. *In the rest of this discussion, we will describe the algorithm to allocate and schedule a given group of tasks onto a given set of cores.*

As mention in Section 3, tasks are split into resource independent and resource sharing subtasks and the two types of subtasks are executed on mutually exclusive cores. Resource sharing subtasks corresponding to a given resource are assumed to be allocated to a common dedicated core, termed a *critical* core for that resource. Subtasks on critical cores are scheduled *non-preemptively*[1] using a static priority assignment among them. Resource independent subtasks are partitioned onto an exclusive set of cores, with all subtasks of a given task being allocated to a common core, termed the task's *parent* core. Subtasks on parent cores are scheduled using a *preemptive*, static-priority scheduling scheme. In our current implementation, we employ the Rate Monotonic (RM) priority assignment policy [9].

Phases or offsets of each subtask with respect to the release time of its parent task are statically determined. Specifically, we calculate the worst-case response time of a given subtask and use this to offset the release of the next subtask of the same task. This gives rise to a potentially non-work-conserving schedule among subtasks of a given task.

We will now describe the steps and theory involved in our algorithm. To ease understanding, we will employ a running example throughout the description. Table 2 represents the characteristics of the task set that is used as a running example. The first, second and third columns show the index (or ID), period and locked WCET, respectively, of tasks. The locked WCET of a task is calculated assuming all its lines are locked in the cache. The fourth column shows the shared resource usage for tasks. The format is [ρ_p; C] (from t), where ρ_p is the shared resource used, C is the length of the critical section and t is the time at which the resource is requested, relative to the start of the task. In this example, all tasks belong to a single group due to the nature of shared resource usage. Tasks within a group are chosen for allocation in non-increasing order of their locked utilization (i.e., utilization calculated with locked WCET). Please note that, for the sake of clarity, we do not complicate the running example by adding migration overheads although we do include these overheads in our experiments.

i	T_i	C_i	*Shared_Resources*
1	90	30	[ρ_2; 10] (from 10)
2	900	300	[ρ_2; 30] (from 100)
3	100	30	[ρ_2; 5] (from 10), [ρ_2; 10] (from 15)
4	1000	300	[ρ_2; 20] (from 100), [ρ_3; 100] (from 120)
5	80	20	[ρ_1; 5] (from 5), [ρ_2; 5] (from 10)
6	800	200	[ρ_3; 50] (from 50), [ρ_2; 20] (from 100)

Table 2: Running Example - Task Set Characteristics

4.1 Division of tasks into subtasks

We split every task into subtasks on the basis of its critical sections. Since we assume non-nested resource sharing, a task using n resources has between n and $2n + 1$ subtasks. Figure 1 shows the division of tasks into subtasks for our running example. Here, tasks τ_1 and τ_2 are divided into three subtasks each and tasks τ_3 to τ_6 are divided into four subtasks each. Resource sharing regions using different resources are depicted using different colors and resource independent regions of all tasks have a common color.

4.2 Allocation of subtasks

The next step is to allocate the resource independent subtasks of a chosen task to a suitable parent core and its resource sharing subtasks to a set of critical cores such that the task is schedulable, or to declare the task unschedulable. First, a *potential* parent core for the task is chosen from among cores that are already designated as parent cores for

[1]Note that several state-of-the-art techniques [13, 8, 1, 12, 17] employ non-preemptive scheduling for resource sharing regions in order to reduce task blocking times.

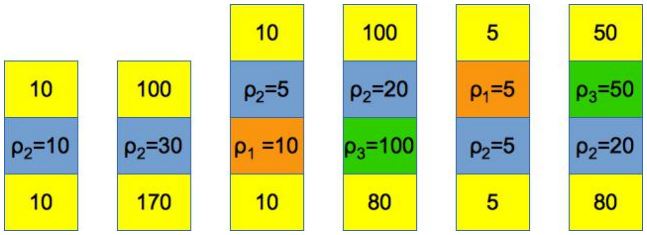


Figure 1: Running Example - Division of tasks into subtasks

previously allocated tasks, if any, or from the set of empty cores, otherwise. If no potential parent core is found, the task is deemed unschedulable. If multiple potential parent cores are found, one of them is chosen and the resource independent subtasks of the task are allocated to the chosen parent core. Resource sharing subtasks of a task are allocated to the critical cores already designated to their respective resources. If no core has yet been designated for a given shared resource, an empty core is chosen and designated as its critical core. If no empty core exists, the task is deemed unschedulable.

Once a potential parent core and a set of critical cores have been identified, an upper bound on the response time of each subtask, in execution sequence, is calculated. The phase or offset of the first subtask of a task is zero with respect to the release time of the task. The phase of every subsequent subtask is equal to the sum of its predecessor's phase and worst-case response time. In other words, the worst-case completion time of a subtask is assumed to be the release time of the next subtask. If the sum of all subtask response times is less than or equal to the relative deadline of the task, the task is declared schedulable and the potential parent core being considered is marked as a *candidate* parent core. In a similar way, other candidate parent cores are identified. Among all candidate parent cores, the core with the lowest current utilization (i.e., before the allocation of the current task) is actually designated as the task's parent core. If no candidate parent core is found, the task is deemed unschedulable.

4.3 Response time calculation

When preemption delays are not considered and tasks do not self-suspend, the worst-case response time of a task occurs when it is released at the critical instant, i.e., when it is released in phase with all higher-priority tasks [10]. In the current context, although each *task* may be shifted to have zero phase, subtasks within a task cannot be similarly shifted since their phases establish precedence constraints that are required for correctness of execution.

In order to guarantee safety of schedulability analysis in this context, it would be necessary to calculate response times at the job level instead of the task level, i.e., the response time of each job of a given task in the hyperperiod of a task set would need to be calculated. Since tasks are split into subtasks, each higher-priority subjob needs to be explicitly enumerated in the calculation. In practice, the complexity of such analysis becomes prohibitive, especially for a large hyperperiod. In order to perform a calculation at the task level while still being safe, we propose the use of a *virtual task* corresponding to a given task on a given core.

A virtual task of a task τ_i, denoted as v_i, is represented by the tuple (Cv_i, Tv_i, Av_i), where Cv_i is its WCET, Tv_i, its minimum inter-arrival time and Av_i, the maximum number of activations it may have within the period of task τ_i. Note that this virtual task is used purely for the purposes of analysis, as a means to capture the worst-case interference that the set of subtasks of task τ_i on a given core c, denoted by $\mathbf{T}_i^c$, can impose on lower-priority tasks on that core.

The characteristics of the virtual task, namely v_i^c, of τ_i on core c, are derived using Equations 1, 2 and 3, respectively.

$$Cv_i^c = \max C_{i,j}, \quad \forall \tau_{i,j} \in \mathbf{T}_i^c \tag{1}$$

$$Tv_i^c = \min\{\Delta_i^c\} \tag{2}$$

$$Av_i^c = |\mathbf{T}_i^c| \tag{3}$$

Here, Δ_i^c represents the set of differences between the phases of consecutive subtasks in $\mathbf{T}_i^c$. By taking the minimum among the inter-arrival times of subtasks, we account for the worst-case frequency of subtask activations for task τ_i on core c. By assigning the maximum among subtask WCETs as the WCET of the virtual task, we capture the longest interference that each activation can impose on lower-priority subtasks. When calculating the worst-case response time for a particular subtask on its core, we assume that all higher-priority virtual tasks on that core release in phase with the subtask under consideration. In this context, the modified response time calculation for subtask $\tau_{k+1,j}$, scheduled on core c, is shown in Equation 4.

$$h(t) = \sum_{i=1}^{i=k} \lfloor \frac{t}{T_i} \rfloor \times C_i^c + \min(\lceil \frac{t\%T_i}{Tv_i^c} \rceil, Av_i^c) \times Cv_i^c] + C_{k+1,j} + \beta\_L_{k+1,j} \tag{4}$$

Here, $h(t)$ is the demand on core c in an interval of time t. The first term in the equation captures interference by higher-priority task instances that are released and have deadlines within the interval t. Here, C_i^c denotes the total WCET of task τ_i on core c, i.e., this term only counts the WCETs of the subtasks of task τ_i that execute on core c and not all the subtasks of τ_i. C_i^c is calculated using Equation 7. The min term in Equation 4 captures the interference from the subtasks of the last instance of task τ_i that is released within the interval t but that does not complete fully within the interval by using the representative virtual task Tv_i^c. Here, the use of the min function ensures that the maximum number of virtual task instances considered is limited to the maximum number of subtasks of τ_i that exist on core c.

We now need to prove that the response time $h(t)$, calculated using Equation 4, is maximal. For this, we simply need to prove that the contribution of every higher priority task in the response time of the task under consideration (τ_{k+1}) within the interval t is maximal.

Let the contribution of a higher priority task τ_i in time t be $R_i(t)$. In other words, $R_i(t)$ is calculated using Equation 5.

$$R_i(t) = \lfloor \frac{t}{T_i} \rfloor \times C_i^c + \min(\lceil \frac{t\%T_i}{Tv_i^c} \rceil, Av_i^c) \times Cv_i^c \tag{5}$$

Let us assume that $R_i(t)$ is not maximal. This would imply that the contribution of task τ_i to the response time of τ_{k+1} in time t is $R_i(t)+\delta$. Here, $\delta \in \{C_{i,j}^c\}$ must hold, with $\{C_{i,j}^c\}$ representing the set of WCETs of subtasks of τ_i allocated to core c.

The first term in Equation 5 is part of the standard response time analysis. So, we only need to consider the contribution of the last job of τ_i release in the interval t, which is represented by the second term in Equation 5. Let the number of subtasks of task τ_i executed in the remaining time in t, namely $t\%T_i$, be ns. According to our assumption, since $\min(\lceil \frac{t\%T_i}{Tv_i^c} \rceil, Av_i^c)$ represents the number of subtasks in time $t\%T_i$ in Equation 5, ns must be greater than that. This is represented by Inequality 6.

$$ns > \min(\lceil \frac{t\%T_i}{Tv_i^c} \rceil, Av_i^c) \quad (6)$$

We now have two cases to consider.

Case 1: $\min(\lceil \frac{t\%T_i}{Tv_i^c} \rceil, Av_i^c) = Av_i^c$. Recall that, by definition, Av_i^c is the total number of subtasks of τ_i in time T_i. Since $t\%T_i < T_i$, only subtasks of one job of τ_i are being considered. Hence, $ns > Av_i^c$ is a contradiction.

Case 2: $\min(\lceil \frac{t\%T_i}{Tv_i^c} \rceil, Av_i^c) = \lceil \frac{t\%T_i}{Tv_i^c} \rceil$. Once again though, by definition, Tv_i^c is $\min\{T_{i,j}^c\}$. Hence, $\frac{t\%T_i}{Tv_i^c}$ is the maximum number of segments in interval $t\%T_i$. Thus, $ns > \lceil \frac{t\%T_i}{Tv_i^c} \rceil$ is a contradiction.

Thus, our assumption that $R_i(t)$ is not the maximum contribution of a higher priority task τ_i must be false. This proves that the contribution from higher priority tasks considered in Equation 4 is maximal and, hence, safe.

In our algorithm, all resource sharing subtasks pertaining to a given shared resource execute non-preemptively on a core dedicated to the resource, namely the *critical* core for that resource. Hence, the only blocking times that resource sharing subtasks may experience are caused by lower-priority resource sharing subtasks using the same resource. Specifically, the blocking time experienced by a resource sharing subtask $\tau_{k+1,j}$ using a shared resource ρ_p is the maximum among the lengths of *all* resource sharing subtasks using resource ρ_p. The blocking time of a subtask $\tau_{k+1,j}$ is denoted as $\beta\_L_{k+1,j}$.

$$C_i^c = \sum C_{i,j}, \quad \forall \tau_{i,j} \in \mathbf{T}_i^c \quad (7)$$

4.4 Task Allocation : Running Example

We now illustrate our approach for task allocation and schedulability check using the running example. Recall that the subtasks in this task set are shown in Figure 1. Let us assume that the system has 9 cores (3x3 mesh), labeled $c0$, $c1$,... $c8$. In accordance with a non-increasing utilization based ordering scheme, tasks are chosen for allocation in the order τ_1, τ_2, ... τ_6.

Allocation of task τ_1: At this stage, since all cores are empty, task τ_1's resource independent subtasks $\tau_{1,1}$ and $\tau_{1,3}$ are tested and successfully allocated on core $c0$ — designated as its parent core — and its single resource sharing subtask $\tau_{1,2}$, on core $c1$ — designated as the *critical* core for resource ρ_2. Figure 2 depicts the response times of the subtasks of τ_2 on cores $c0$ and $c1$, respectively. Since the task τ_1 is the only allocated task on both these cores, the worst-case response times for subtasks are equal to their respective WCETs. After this allocation, the total WCET of τ_1 on core $c0$, denoted by C_1^0, is 20 and that on core $c1$, namely C_1^1, is 10.

Calculation of virtual tasks for τ_1: From Figure 2, we observe that there are two subtasks of τ_1 that execute on $c0$. The inter-arrival time between these subtasks is 20, which

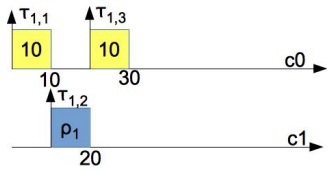


Figure 2: System state after allocation of task τ_1

makes the period of the virtual task v_1^0, namely Tv_1^0, equal to 20 in accordance with Equation 2. The maximum WCET among the subtasks in $\mathbf{T}_1^0$ is 10 and so, the WCET of the virtual task v_1^0, namely Cv_1^0, is 10 in accordance with Equation 1. On core $c1$, since there is only one subtask of task τ_1, the period of the virtual task v_1^1 is equal to the period of the *task* τ_1, namely 90. The WCET of v_1^1 is equal to the WCET of the subtask of τ_1 on core $c1$, namely 10. Hence, at this stage of the algorithm, the characteristics of the virtual tasks for task τ_1 are $v_1^0(10, 20, 2)$ and $v_1^1(10, 90, 1)$.

Allocation of task τ_2: We now use the steps involved in the allocation of task τ_2 to demonstrate the calculation of worst-case response times for **resource independent subtasks**. At this stage of the algorithm, $c0$ is an existing parent core, so we begin by considering $c0$ as a *potential* parent core for τ_2. Since τ_2 also uses resource ρ_2, its resource sharing subtask is tested on core $c1$, which has already been designated as the critical core for ρ_2.

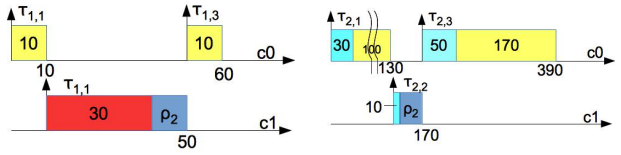
Temporary allocation of τ_2*'s subtasks*: To determine the schedulability of the system if core $c0$ were to be designated as the parent core for τ_2 and core $c1$, its critical core, we temporarily perform these allocations, i.e., $\tau_{2,1}$ and $\tau_{2,3}$ are assigned to core $c0$ and $\tau_{2,2}$, to core $c1$.

Temporary recalculation of virtual task for task τ_1: Since τ_1 and τ_2 share a resource, the resource sharing subtask of the higher-priority task τ_1, namely $\tau_{1,2}$, experiences blocking. In Figure 3, the rectangle shaded in *red* denotes the blocking time experienced by this resource sharing subtask. Since this blocking time changes the response time of subtask $\tau_{1,2}$, it also changes the phase of $\tau_{1,3}$ on core $c0$. This requires recalculation of the period of the virtual task of τ_1 on core $c0$. Applying Equation 2 once again, we get $v_1^0(10, 50, 2)$. Characteristics of v_1^1 remain unaffected.

Calculation of response time of task τ_2: Equation 8, derived from the modified response time calculation shown in Equation 4, is used to calculate the worst-case response time of subtask $\tau_{2,1}$.

$$h(t) = \sum_{i=1}^{i=1} \lfloor \lfloor \frac{t}{100} \rfloor \times 20 + \min(\lceil \frac{t\%100}{50} \rceil, 2) \times 10 \rfloor + 100 \quad (8)$$

Here, the initial value of t is the WCET of the subtask $\tau_{2,1}$, namely 100. Upon performing the iterative calculation, a response time of 130 is obtained for subtask $\tau_{2,1}$. Performing similar calculations for other subtasks, we obtain the response times depicted in Figure 3. Here, the rectangles shaded in *blue* represent delays suffered by lower-priority subtasks due to interruptions by higher-priority virtual tasks. Since the worst-case response times of both tasks are less than their respective relative deadlines, the tasks are schedulable. Thus, core $c0$ is a *candidate* parent core for task τ_2. Since core $c0$ is also the only (initially non-empty) candidate parent core at this stage, it is designated as the parent core of task τ_2, i.e., the allocation of τ_2 is made permanent.



(a) Task τ_1 (b) Task τ_2

Figure 3: System state after allocation of τ_1 and τ_2

Calculation of virtual tasks for τ_2: Based on the WCETs and response times of its subtasks, the WCET and period characteristics of the virtual tasks of task τ_2 on cores $c0$ and $c1$, respectively, are calculated as $v_2^0(170, 170, 2)$ and $v_2^1(30, 900, 1)$. To elaborate, there are two subtasks, namely $\tau_{2,1}$ and $\tau_{2,3}$, of τ_2 on core $c0$. The WCET of the virtual task v_2^0, namely Cv_2^0, is $\max(C_{2,1}, C_{2,3})$, which is 170. The inter-arrival times of these subtasks is also equal to 170, thus making Tv_2^0 equal to 170.

Allocation of task τ_3: Using the same process as detailed above, core $c0$ is also designated as the parent core for task τ_3 (C_3^0 is equal to 15). Since τ_3 also uses resource ρ_2, one of its resource sharing subtasks, namely $\tau_{3,2}$, is allocated to critical core $c1$ of resource ρ_2. In addition, task τ_3 uses resource ρ_1 in subtask $\tau_{3,3}$. An empty core, namely $c2$, is designated as the critical core for ρ_1.The resulting response times of tasks after the allocation of τ_3 are depicted in Figure 4.

In Figure 4, note the changes in response times for the subtasks of task τ_2 as compared to the one shown in Figure 3. This change is due to the fact that task τ_3 has higher priority than task τ_2 (Rate Monotonic priority assignment is assumed). The response times of resource independent subtasks of task τ_1 do not change because it is the highest-priority task in the system. Response times of the resource sharing subtask of τ_1 also remains the same because the critical section length for task τ_3 is less than that for τ_2, thus resulting in no increase in blocking time. Since task τ_3 has a new shared resource, namely ρ_1, a new empty core $c2$ is designated as the *critical* core for resource ρ_1.

Virtual task characteristics: At the end of the allocation of τ_3, virtual task characteristics are updated as follows: virtual tasks for τ_1 are $v_1^0(10, 50, 2)$, $v_1^1(10, 90, 1)$; those for τ_2 are $v_2^0(170, 210, 2)$, $v_2^1(30, 900, 1)$; and those for τ_3 are $v_3^0(10, 65, 2)$, $v_3^1(5, 100, 1)$, $v_3^2(10, 100, 1)$. Note that, after allocation of every new task, our algorithm recalculates virtual task characteristics for all previously allocated tasks.

Allocation of task τ_4: Upon following the same procedure as that for earlier tasks, it is found that task τ_4 cannot be accommodated on parent core $c0$. Hence, at this stage, an empty core $c3$ is chosen as a potential parent core for task τ_4 and a schedulability check is performed. The check is successful and the state of the system after the allocation of task τ_4 is depicted in Figure 5. Note that, since all resource sharing subtasks of a given resource execute on a common core, the fact that τ_4 belongs to a different *parent* core than tasks τ_1, τ_2 and τ_3 does not make a difference to the blocking times. In other words, there is no concept of *remote* blocking in our context.

5. ARCHITECTURAL CONSIDERATIONS

In this work, we assume the use of a multi-core architecture with lockable private set-associative caches, no shared caches and a bidirectional, mesh-based Network-on-Chip (NoC) interconnect such as the one found on Tilera's TilePro64 [16]. We now briefly discuss the effects of these architectural features on our task allocation and scheduling approach.

As discussed thus far, our algorithm splits a task into resource independent and resource sharing subtasks and the two types of subtasks are executed on the task's parent core and a set of critical cores, respectively. As mentioned in Section 3, we assume that every subtask may lock a chosen subset of its private data in the cache of the core to which it is allocated. We also assume that every shared resource (i.e., shared across multiple tasks) is locked in one exclusive cache way on its critical core.

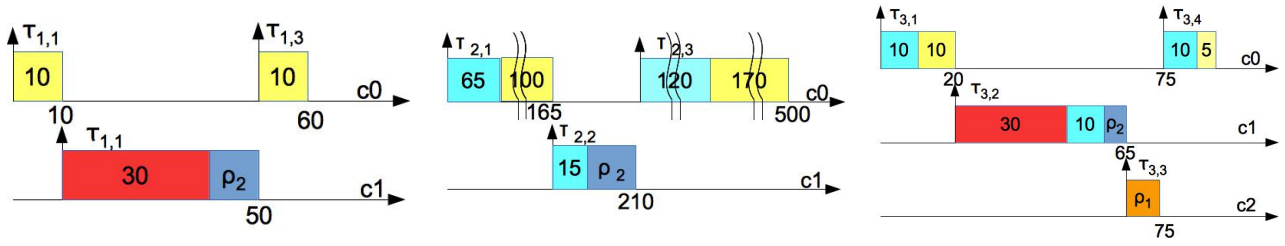
However, a task may have some private data that is used in both its resource independent subtasks (i.e., on its parent core) and its resource sharing subtasks (i.e., on one of its critical cores). We assume that any such data of a task is initially locked in the cache of its parent core. This data is proactively migrated and re-locked on a given critical core of the task when the task acquires the shared resource corresponding to that critical core. The data is assumed to be migrated back to its parent core when the shared resource usage is complete, but before actually releasing the resource. We employ a push-based migration mechanism developed in prior work [14] for this purpose. As a consequence of this migration model, the overhead of migration contributes to the worst-case blocking times of other tasks using the same shared resource and must be tightly bounded. This is accounted for by explicitly adding the migration overhead for a critical section of a task added to the WCET of the critical section.

Since they correspond to a task's parent core and a specific critical core, the source and destination cores for a given migration are statically known. In general, this information could be used to statically determine routes for each data migration in an effort to obtain tighter bounds on the migration overhead. However, such routing policies and analyses thereof are out of the scope of the current paper. Instead, we make some simplifying assumptions, namely that 1) traffic among cores assigned to a given group of resource sharing tasks does not interfere with traffic from other groups of tasks; 2) cores assigned to a given group are arranged in a mesh-based pattern on the NoC; 3) every migration among a given group could travel over the largest number of hops, under an X-Y routing scheme, within its group of cores; and 4) NoC contention is handled through the use of a simple time-division multiplexing among cores and, within each core, among different migration streams.

Under these assumptions, the worst-case one-way migration overhead for a given chunk of data is calculated using Equation 9.

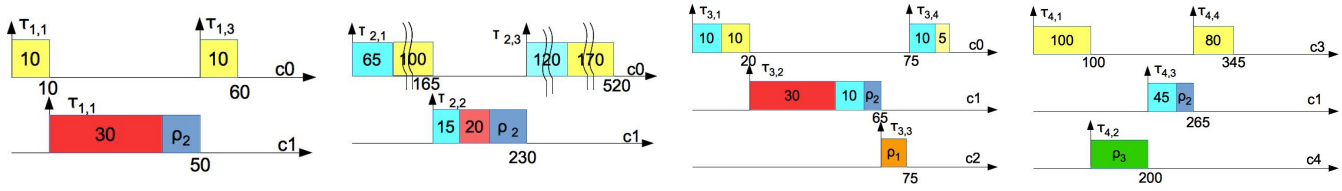
$$M_l^c = h_{max} * \frac{L}{B} * m_p * \min(|\mathbf{T}^c|, |\Re^c|) * l \qquad (9)$$

Here, M_l^c is the overhead for migrating l cache lines from/to a parent core c. $h_{max} = \lceil \frac{m^g}{2} \rceil$, where m^g is the number of cores allocated to the group of tasks under consideration, is the largest number of hops that need to be traversed assuming the cores in the group are organized as a mesh. L is the size of a cache line, B is the bandwidth along each hop of the NoC, m_p is the number of parent cores in the given group, $\mathbf{T}^c$ is the set of tasks whose parent core is c and $\Re^c$ is the set of unique shared resources accessed by tasks whose parent



(a) Task τ_1 (b) Task τ_2 (c) Task τ_3

Figure 4: System state before allocation of task τ_4



(a) Task τ_1 (b) Task τ_2 (c) Task τ_3 (d) Task τ_4

Figure 5: System state after allocation of task τ_4

core is c. The basic idea behind Equation 9 is that, since migration data starts from (or returns to) a parent core, the total number of parent cores is an upper bound on the number of cores sharing a given channel of the NoC for migration traffic in a time-division multiplexed manner. Within each parent core, at most $\min(|\mathbf{T}^c|, |\Re^c|)$ number of migrations may be contending for core c's share of the bandwidth.

We choose to employ a time-division multiplexed approach as opposed to prioritizing migration streams because using a priority-based scheme could result in prohibitively large migration overheads for lower-priority tasks, which would, in turn, result in prohibitively large blocking times.

6. DISCUSSION

6.1 Limitations

We now discuss limitations of our algorithm and present our justification for some of the limiting choices. Our algorithm 1) results in an increase in the total number of cores used, 2) results in a potential reduction of actual core utilization due to the use of a partially non-work-conserving schedule, 3) results in a potential increase in migration traffic and 4) currently does not support nested resources.

Since we allocate a dedicated core for each shared resource in the system, our algorithm admittedly uses an increased number of cores. However, in today's modern multi-core architectures, the number of cores on a single platform is continuing to increase. In this context, the fundamental bottleneck for schedulability of resource sharing real-time tasks is not the number of cores, but rather prohibitively high remote blocking times. In fact, increase in remote blocking makes the availability of cores useless.

Having said this, as part of future work, we propose to extend our algorithm to allow multiple shared resources to reside on a single critical core. This extension would require us to re-introduce some blocking terms that we currently do not require. We propose to conduct a sensitivity study to determine the trade-off between decreased schedulability and improved core usage. Furthermore, we propose to exploit this to allow a limited form of nested resource usage through the use of group locks on a set of resources that are allocated to the same critical core.

Although the use of a non-work-conserving schedule reduces the online core utilization, it enables us to calculate a much tighter estimate of subtask response times. Since real-time systems design requires *a-priori* guarantees of schedulability, we believe that the improved schedulability analysis makes the loss of processor utilization an acceptable trade-off.

Since our algorithm insists that resource independent data that may be needed on both parent and critical cores is migrated from and returned to the parent core of a task, it results in an increase in migration traffic and hence, migration overhead and task blocking times. In order to provide a realistic evaluation of the performance of our algorithm, as mentioned in Section 5, we explicitly account for migration costs in our experimental results.

6.2 Complexity

Every time a new task is checked for allocation, all previously allocated tasks require a recalculation of their response times, leading to a complexity of the $O(n^2)$ for a task set of size n. Our algorithm checks for the possibility of allocating a given task on multiple potential cores. Hence, the complexity of the overall algorithm is of the order $O(n^2 * m)$, where m is the number of cores. However, since this is an offline calculation, we believe it is an acceptable complexity. The Greedy Slacker algorithm that we compare our algorithm with [17] has a very similar complexity.

7. GREEDY SLACKER ALGORITHM

In this section, we briefly describe the outcome of the running example that was used in Section 4 when using the Greedy Slacker algorithm proposed by Wieder and Bran-

denburg [17]. The Greedy Slacker algorithm assumes that task critical sections are executed non-preemptively and that blocked tasks busy wait on their allotted core for access to shared resources. Response time analysis (Equation 10) is used to determine the schedulability of a set of tasks allocated to a core.

$$R_i = C_i + B_i^R + \max(B_i^{NP}, B_i^L) + \sum_{j=1}^{i-1}(\lceil \frac{R_i}{T_j} \rceil \times (C_j + B_j^R)) \quad (10)$$

Here, R_i and C_i are the response time and WCET of task τ_i, respectively. B_i^R is the remote blocking for task τ_i, i.e., the blocking time experienced by it when a resource it requests is being used by a task on another core. B_i^{NP} is the non-preemptive blocking for task τ_i, i.e., the blocking time experienced by it when a lower-priority task on the local core is busy waiting, followed by (non-preemptive) critical section execution. B_i^L is the local blocking for task τ_i, i.e., the blocking time experienced by it due to a lower-priority task on the local core using a local resource.

Tasks τ_1 and τ_2 are allocated, in their entirety, to core $c0$. After this allocation, the worst-case response time for task τ_2 is calculated using Equation 11.

$$h(300) = 300 + \lceil \frac{100}{90} \rceil \times 30 \quad (11)$$

Solving Equation 11, we obtain a worst-case response time of 450 for task τ_2. Figure 3 shows that the worst case response time of task τ_2 at this stage using our algorithm was 390. Subsequently, the Greedy Slacker allocates τ_3 on core $c0$. The algorithm then checks whether τ_4 is schedulable on core $c1$. As part of this check, the worst-case response time for task τ_3 is calculated using Equation 12.

$$h(30) = 30 + 20 + \max(20 + 30, 30) + \lceil \frac{30}{90} \rceil \times (30 + 20) \quad (12)$$

The first two terms in this equation are the WCET and total blocking time, respectively, for task τ_3. The third term denotes the interference from higher-priority task τ_1 in the response time of τ_3. It should be noted that, since Greedy Slacker busy waits for a global resource, in the third term, the remote blocking of task τ_1 is added along with its WCET. This extra addition of remote blocking in the third term makes task τ_3 unschedulable after the allocation of task τ_4 in the system. Hence, τ_4 is deemed unschedulable on the system and the algorithm quits, leaving tasks τ_4, τ_5 and τ_6 unscheduled in this case. On the other hand, from Figure 5, we see that, using our algorithm, the worst case response time of task τ_3 remains unchanged (equal to 90) after the allocation of task τ_4. Furthermore, our algorithm manages to schedule all six tasks on the given set of cores.

The primary reasons why our algorithm outperforms Greedy Slacker here are that 1) our algorithm employs a non-work-conserving schedule for a given task, enabling it to calculate tighter response time bounds; and 2) since Greedy Slacker assumes that tasks busy wait for a shared resource, it requires the addition of a remote blocking to be added to the WCET of higher-priority tasks in the response time calculation.

7.1 Migration Overheads

In Section 8, we quantitatively compare the performance of our proposed scheme with the greedy slacker algorithm. In order to perform a fair comparison among the two schemes, we explicitly account for the overheads introduced due to *one-way* migration of a shared resource to the core on which the task that acquires it is allocated. As with our algorithm, we use the time-division multiplexed approach described in Section 5 to account for traffic contention in the NoC. The calculation of migration overhead in the context of the greedy slacker is shown in Equation 13.

$$M_l = h_{max} * \frac{L}{B} * \min(m^g, |\Re^G|) * l \quad (13)$$

Here, M_l is the overhead for migrating l cache lines, m^g is the total number of cores allocated to the group of tasks under consideration and $\Re^G$ is the set of unique global resources. In the greedy slacker, since migration is required only for *global* shared resources, the number of simultaneous migrations is limited by the number of unique global shared resources. Furthermore, since tasks busy-wait in a non-preemptive manner for resource access, on each core, at most one migration can be performed at a time. Hence, the number of simultaneous migrations is limited by the minimum of the number of cores and number of global resources. Finally, in the greedy slacker, since a global shared resource may have to migrate from any core within a group to any other core, using the largest possible number of hops, $h_{max} = \lceil \frac{m^g}{2} \rceil$, is *required* for safety.

8. EVALUATION

The architectural configuration we assume is shown in Table 3. The external memory latency shown is calculated using the weighted TDM approach proposed in our previous work [15]. We assume that all four ways of the L1 data cache on cores are lockable. On *critical* cores, i.e., cores that are dedicated for the execution of resource sharing subtasks using a given shared resource, one cache way is reserved for locking the footprints of the shared resource.

Parameter	**Configuration**
Processor Model	in-order
Cache Line Size	32Bytes
L1 D-Cache Size/Associativity	256KB/4-way
L1 hit latency	1 cycle
Replacement Policy	Least Recently Used
Number of Cores	16
Cache to cache Transfer latency	13 cycles
External Memory Latency	90 cycles

Table 3: System Configuration

Our goal is to evaluate the effectiveness of our algorithm, alongside the Greedy Slacker algorithm proposed by Wieder et al. [17] and DPCP proposed by Rajkumar [13] under varying task set characteristics. To achieve this, we employ synthetic task sets that are randomly generated using an unbiased task set generator based on an algorithm proposed by Bini et al. [4][2].

We generate task sets that have a total utilization of up to 4 for a 16 core system as mentioned in Table 3. Note that, due to increases in actual utilization due to blocking times, a base utilization of 4 is reasonable. Each task set consists of 30 tasks. Table 4 represents the varying parameters used while generating task sets for our experiments. The resource sharing factor is used in order to ensure that, in every task set, at least a certain number of tasks use each shared resource. As defined by Wieder et al. [17], if the resource sharing factor is r and the total number of tasks in a task

[2]This generator has been appropriately modified to suit a multi-core environment.

Parameter	Range
Number of Resources	3-9, in steps of 1
Resource Sharing Factor	5% to 50%, in steps of 5%
Critical Section Length	2% to 8%, in steps of 1%

Table 4: Task Set Configuration Parameters

set is n, then the minimum number of tasks sharing each resource in the task set is equal to $r * n$. The critical section length for a task is defined as the percentage of its WCET that the task spends in critical sections. We use short critical sections in our evaluation since it is typical that critical sections are kept as short as possible in practice. Varying the resource sharing factor and critical section lengths for a given number of shared resources allows us to evaluate the performance of the three algorithms.

For each configuration, i.e., combination of the number of resources, the resource sharing factors and the critical section lengths, we generate 100 task sets. Each generated task set is provided as input to our semi-partitioned algorithm (hereafter denoted as **SP**), the Greedy Slacker (denoted as **GS**) and **DPCP**. All algorithms choose tasks for allocation in the same order and quit as soon as any one task becomes unschedulable. As such, the success ratio of each algorithm is reflected by the total utilization of tasks scheduled by it. Hence, in our evaluation, we use a metric based on the scheduled utilization for comparison. Specifically, for a given configuration, we show the *percentage of task sets for which each of the three algorithms had the greatest scheduled utilization.*

Figures 6 and 7 show two graphs comparing the performance of the three algorithms, namely SP, GS and DPCP. Each graph depicts the variation in performance of the three algorithms for a given critical section length, but with varying number of resources and varying resource sharing factors. Due to space constraints, we only present results for the two extreme critical section lengths among our configurations, namely 2% and 8%. The common legend for both graphs is shown as part of Figure 6. The x-axis of each graph shows the combination of resource sharing factor and the number of resources for a given experiment, denoted as res-sharing-factor&num-resources. The y-axis shows the percentage of task sets where a given algorithm manages to schedule a higher (locked) utilization than the other two algorithms[3]. In other words, each stacked bar shows, for a specific configuration of resource sharing factor and number of resources, the above-mentioned percentages for all three algorithms. For example, the first stacked bar in Figure 6 indicates that, for 3 resources and a resource sharing factor of 5%, SP performs better in approximately 65% of the task sets, DPCP performs better in approximately 25% of the task sets, and GS, in approximately 10% of the task sets.

From these two graphs, we may make several observations. (1) SP performs better (i.e., achieves a higher scheduled utilization in a larger number of task sets) than the other two algorithms in most of the configurations presented. The primary reason that SP mostly outperforms DPCP is that DPCP uses a pessimistic blocking time calculation. The comparison of SP and DPCP would be much more interesting when we enhance DPCP with the optimal blocking time calculation technique proposed by Brandenburg [5]. However, this technique uses a linear programming based approach. In order for us to use it in the context of our paper, we require a change to the objective function to incorporate migration overhead. So, we defer such a comparison to future work. SP outperforms GS because blocking time is significantly reduced in SP since execution with shared resources occurs on exclusive cores. (2) In most cases (for configurations with number of resources 5 and higher in Figure 6 and for most configurations in Figure 7), as the resource sharing factor increases, for a fixed number of resources (i.e., within sets of 10 consecutive bars), SP shows a trend of improved performance. This is because, as the resource sharing factor increases, since a larger number of tasks share a given resource, there is more benefit in the reduced blocking times obtained in SP due to execution of resource sharing regions on exclusive cores. (3) For lower resource sharing factors, as the number of resources increases, the performance of both GS and DPCP improves more than in the case of larger resource sharing factors. This is because, for smaller resource sharing factors, it is likely that a smaller number of tasks share a given resource. Hence, there is an increased likelihood that some resources end up being local resources instead of global resources. In both GS and DPCP, local resources require no data migration. In contrast, SP always requires migration since all shared resources are global.

Next, we study the trend in the performance of SP and GS as critical section lengths are varied. In GS, resources are migrated to the core on which a task using the resource executes. In contrast, SP migrates tasks to the core on which a shared resource resides. Hence, this study demonstrates the effects of varying migration overheads for the two algorithms. Figures 8 and 9 depict the results of this study. The x-axis in both graphs shows critical section lengths and the y-axis shows the average percentage for which a given algorithm schedules higher utilizations over configurations with varying resource sharing factors for a given critical section length. Both graphs show results for a fixed number of resources as indicated. Since we use randomly generated task sets for our experiments, we employ simple linear regression to identify performance trends.

In Figure 8, each scatter point denotes the average of the percentage of task sets for which GS performed the best over configurations with resource sharing factors between 25% and 50% (i.e., higher resource sharing factors), a given critical section length and for 9 resources. For this setup, the trend line shows that GS performs better at lower critical section lengths and its performance deteriorates as critical section lengths increase. This is because, as the resource sharing factor increases, the likelihood of global resources increases. In GS, resources are global resources and are migrated to the cores on which tasks using them reside. Smaller critical sections typically indicate smaller sizes of shared resources, hence requiring less data migration. Larger critical sections typically require more data migration, thus justifying the trend.

In Figure 9, each scatter point denotes the average of the percentage of task sets for which SP performed the best over configurations with resource sharing factors between 5% and 50% (i.e., a wide range of resource sharing factors), a given critical section length and for 3 resources. In this graph, the trend line shows that, for a given number of resources, the performance of SP improves slightly with increasing criti-

[3]Recall that locked utilization of a task is defined as the utilization of a task when its complete memory footprint is locked in the cache.

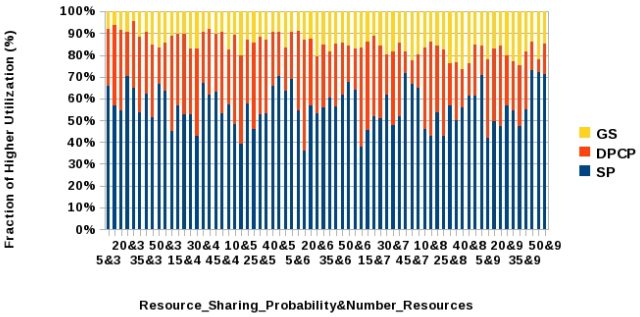


Figure 6: Comparison w/ crit. sec.=2% of WCET

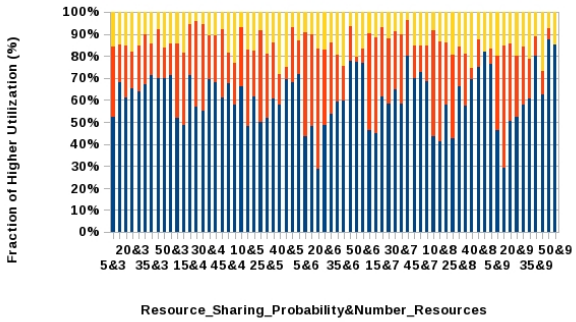


Figure 7: Comparison w/ crit. sec.=8% of WCET

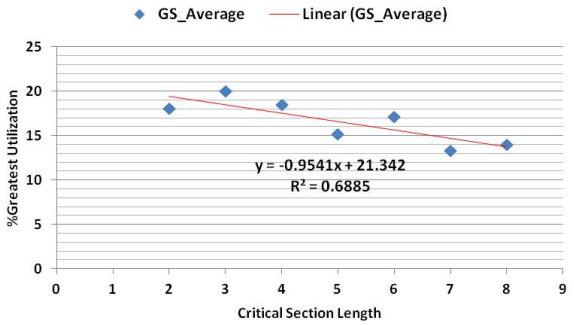


Figure 8: GS trend w/ varying crit. sec. and 9 res.

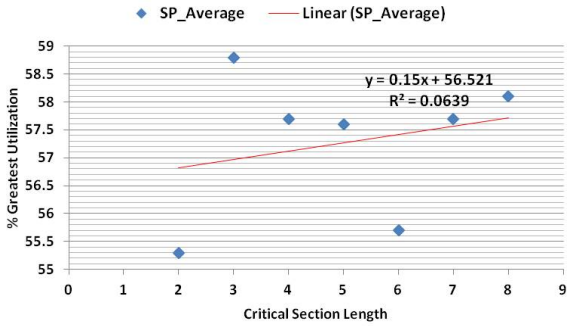


Figure 9: SP trend w/ varying crit. sec. and 3 res.

cal section length. This is because SP migrates tasks to cores on which shared resources reside. Hence, it avoids potentially higher resource migrations that would be required for larger critical sections. To summarize, SP demonstrates better performance in most cases for the small critical section lengths considered in this evaluation. Furthermore, the trend suggests that its performance could improve further for longer critical sections.

9. CONCLUSION AND FUTURE WORK

A major bottleneck in successfully scheduling resource sharing hard-real-time task sets on multi-core architectures is prohibitively high task blocking times. In this paper, we present a novel semi-partitioned scheduling scheme for resource sharing tasks. By allocating and executing resource independent and resource sharing portions of tasks on mutually exclusive cores and employing a partially non-work-conserving schedule, our approach effectively reduces blocking times of tasks, thus improving task set schedulability. We compare the performance of our algorithm against that of a recent partitioned scheduling scheme called Greedy Slacker and the classic Distributed Priority Ceiling Protocol (DPCP). Results demonstrate that our algorithm achieves a higher scheduled utilization in a majority of task sets.

In an effort to decrease blocking times, our algorithm assumes that every shared resource is allocated a dedicated core on which all critical sections using that resource execute. This assumption leads to the use of an increased number of cores. Furthermore, our algorithm does not allow nested resource usage. As part of future work, we propose to extend our algorithm to allow multiple shared resources on a given core and study the trade-off between better core utilization and increased task blocking times. We also propose to support a subset of nested resource usage through the use of group locks, with resources in a given group sharing a core.

10. REFERENCES

[1] B. B. A. Block, H. Leontyev and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Embedded and Real-Time Computing Systems and Applications*, pages 47–56, 2007.

[2] S. Afshar, F. Nemati, and T. Nolte. Resource sharing under multiprocessor semi-partitioned scheduling. In *RTCSA, 2012*, pages 290–299, Aug 2012.

[3] N. Audsley and Y. Dd. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991.

[4] E. Bini and G.C.Buttazzo. Measuring the performance of schedulability tests. *RTS*, 30(2):129–154, 2005.

[5] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for p-fp scheduling. In *RTAS*, 2013.

[6] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 2011.

[7] J. D. J.M. Lopez and D. Garcia. Utilization bounds for edf scheduling on real time multiprocessor systems. *Journal of Real Time Systems*, 1:39–68, 2004.

[8] K. Lakshmanan, D. de Niz, and R. Rajkumar.

Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, pages 469–478, 2009.

[9] J. Lehoczky, L. Sha, , and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium*, Santa Monica, California, Dec. 1989.

[10] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[11] J. Lozi, F. David, G. Thomas, J.Lawall, and G. Muller. Remote core locking: Migrating critical section execution to improve performance of multi-threaded applications. In *UNEXIS*, 2012.

[12] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *OPODIS*, pages 253–269, 2010.

[13] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, 1988.

[14] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 80–89, June 2009.

[15] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller. Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems. In *Euromicro Conference on Real-Time Systems*, July 2012.

[16] Tilera processor family. http://www.tilera.com/.

[17] A. Wieder and B. Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *SIES 2013*, 2013.

[18] A. Wieder and B. Brandenburg. On spin locks in autosar : Blocking analysis of fifo, unordered and priority ordered spin locks. In *RTSS*, 2013.