Parallel-Task Scheduling on Multiple Resources

Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Department of Mathematics and Computer Science, Technische Universiteit Eindhoven
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

***Abstract*—This paper addresses the problem of scheduling periodic parallel tasks on a multi-resource platform, where tasks have real-time constraints. The goal is to exploit the inherent parallelism of a platform comprised of multiple heterogeneous resources. A resource model is proposed, which abstracts the key properties of any heterogeneous resource from a scheduling perspective. A new scheduling algorithm called PSRP is presented, which refines MSRP. The schedulability analysis for PSRP is presented. The benefits of PSRP are demonstrated by means of an example application showing that PSRP indeed exploits the available concurrency in heterogeneous real-time systems.**

I. Introduction

Modern computers consist of several processing units, connected via one or more interconnects to a memory hierarchy, auxiliary processors and other devices. A simple approach to sharing such platforms between several applications treats the machine as a single resource: the task having access to the processor has also implicitly access to all other resources, such as a bus, memory, or network. Consequently, only a single task is allowed to execute at a time. On the one hand, this approach avoids the complexity of fine-grained scheduling of multiple resources. On the other hand, it prevents tasks with independent resource requirements to execute concurrently and thus use the available resources more efficiently. For example, a video processing task requiring the processor and operating on the processor's local memory can execute concurrently with a DMA transfer task moving data between the global memory and the network interface. In this paper we assume that a task represents workload which does not necessarily require a processor.

With the advent of multiprocessor platforms new scheduling algorithms have been devised aiming at exploiting some of the available concurrency. However, they are again limited to tasks which execute on one processor at a time. In this paper we address the problem of scheduling tasks on a multiprocessor platform, where a task can execute on several processors at the same time. Moreover, a task may also specify requirements for other heterogeneous resources, such as a bus, digital signal processor, shared memory variable, etc. In this respect our problem is related to parallel task scheduling on multiple resources, where a task may execute on several processors at the same time.

Parallel-task scheduling was originally investigated in the context of large mainframe computers without real-time constraints [1]. When threads belonging to the same task execute on multiple processors and communicate via shared memory, then it is often desirable to schedule these threads at the same time (called *gang scheduling* [1]), in order to avoid invalidating the shared memory (e.g. L2 cache) by switching in threads of other tasks. Also, simultaneous scheduling of threads which interact frequently will prevent the otherwise sequential execution due to synchronization points and the large context switching overheads [1]. Parallel-task scheduling is especially desired in data intensive applications (e.g. multimedia processing), where multithreaded tasks operate on the same data (e.g. a video frame), performing different functions at the same rate [2]. To the best of our knowledge, existing literature on preemptive parallel task scheduling with real-time constraints has only considered independent tasks. In this paper we present a fixed-priority preemptive multi-resource scheduling algorithm for parallel tasks with real-time constraints.

Problem description Current multiprocessor synchronization protocols only consider tasks which execute on a single processor at a time. They are not suitable for synchronizing parallel tasks, which may execute on several processors at a time and share resources. A simple approach to scheduling such tasks on a platform comprised of multiple heterogeneous resources is to "collapse" all the processors into one virtual processor and use uniprocessor scheduling. However, this will result in at most one task executing at a time. Our goal in this paper is to provide a scheduling algorithm for parallel tasks with real-time constraints which can exploit the inherent parallelism of a platform comprised of multiple heterogeneous resources.

Contributions In this paper we propose a new resource model, which abstracts the key properties of different kinds of resources from a scheduling perspective. We then present a new partitioned parallel-task scheduling algorithm called Parallel-SRP (PSRP), which generalizes MSRP [3] for multiprocessors, and the corresponding schedulability analysis for the problem of multi-resource scheduling of parallel tasks with real-time constraints. We show that the algorithm is deadlock-free, derive a maximum bound on blocking, and use this bound as a basis for a schedulability test. We present an example which demonstrates the benefits of PSRP.

Outline The remainder of this paper is structured as follows. Section II discusses the related work. The system model is introduced in Section III, followed by a recap of MSRP in Section IV. PSRP is presented in Section V, followed by its analysis in Section VI. An example evaluating PSRP is presented in Section VII and a discussion of PSRP in Section VIII. Section IX concludes this paper.

II. Related work

To the best of our knowledge our work is the first to consider parallel task scheduling on multiple resources and real-time constraints. In this section we discuss the related literature from the domains of multiprocessor scheduling with shared resources and multiprocessor scheduling of parallel tasks.

A. Multiprocessor scheduling with shared resources

When tasks share non-preemptive resources, they may block when the resource they are trying to access is already locked by another task. Such conflicts may be resolved offline by means of a table-driven schedule, or during runtime by means of synchronization protocols.

Dijkstra [4] presents one of the earlier synchronization protocols for multiprocessors, called the Banker's algorithm. The algorithm focuses on avoiding deadlock when several concurrently executing tasks share a common multi-unit non-preemptive resource, but it does not provide any real-time guarantees. Tasks do not have priorities nor timing constraints (besides terminating in finite amount of time), and each task is assumed to execute on its own processor. They may acquire and release the units of the shared resource in any order, as long as the total number of claimed units does not exceed a specified maximum, and as long as they release all claimed units upon completion. [5] presents a generalization of the Banker's algorithm to *several* non-preemptive multi-unit resources.

More recently, existing real-time synchronization protocols for uniprocessors have been extended to multiprocessors. They focus on synchronizing access to global resources, which are resources accessed from tasks executing on different processors. There are two main approaches for handling global resources: when a task wants to access a global resource which is already locked by another task executing on a different processor, the task may be (i) suspended, leaving the processor available for other tasks to execute, or (ii) spin-lock, holding the processor in reserve. [6] takes the suspension based approach and presents the Multiprocessor Priority Ceiling Protocol (MPCP) and Distributed Priority Ceiling Protocol (DPCP) (for distributed memory systems). [3] takes the spin-lock approach and presents the Multiprocessor Stack Resource Policy (MSRP). All three protocols assume partitioned EDF scheduling. The Flexible Multiprocessor Locking Protocol (FMLP) by [7] can be regarded as a combination of MPCP and MSRP and can be applied to both partitioned and global EDF multiprocessor scheduling.

The authors in [8], [9], [10] investigate the performance penalties between various spin-lock and suspension based protocols (MPCP, DPCP, MSRP and FMLP) and conclude that spin-lock based approaches incur smaller scheduling penalty than suspension based, especially for short critical sections. The authors in [11] extend the original suspension-based MPCP description with spin locking, compare the two implementations and show the opposite, i.e. that for low preemption overheads and long critical sections the suspension-based approaches perform better, while in other settings they perform similar. The authors in [7] claim that FMLP outperforms MSRP, by showing that FMLP can schedule more task sets than MSRP. They assume freedom in partitioning the task set, i.e. that tasks may be assigned to arbitrary processors, and exploit this assumption to schedule task sets which are not schedulable under MSRP. Arbitrary partitioning, however, may not necessarily hold for heterogeneous systems, where different processors may provide different functionality. Our PSRP algorithm is spin-lock based. The advantage of choosing this approach is simpler design and analysis, compared to a suspension based approach. We based our algorithm on MSRP, as it suits our model better, in the sense that given the particular resource requirements of tasks we cannot exploit the advantages of FMLP.

Nested critical sections can lead to deadlock. MSRP and MPCP explicitly forbid nested global critical sections. FMLP supports nested critical sections by means of *resource groups*. Two resources belong to the same resource group iff there exists a task which requests both resources at the same time. Before a task can lock a resource r it must first acquire the lock to the corresponding resource group $G(r)$. This ensures that only a single task can access the resources in a group at any given time. On the other hand, the resource groups in effect introduce large super-resources, which can be accessed by at most one task at a time, thus limiting concurrency in the system. In this paper we model tasks as sequences of parallel segments which require concurrent access to a set of resources. Nested global critical sections are addressed by a task segment requiring simultaneous access to all of its required resources (similar to the approach proposed in [12] for the general problem of deadlock avoidance in multitasking), without the need for locking entire resource groups.

The description of MSRP in [3], [8] does not address multi-unit resources, which were supported by the original SRP description for a uniprocessor [13]. Our PSRP algorithm supports multi-unit non-preemptive resources.

Notice that the schedulability analysis for PSRP resembles the holistic scheduling analysis presented by [14]. They describe the end-to-end delay for a pipeline of tasks in a distributed system, where each task is bound to a processor and can trigger a task on another processor by sending a message via a shared network. Their tasks correspond to our segments, and their pipelines of tasks correspond to our tasks. However, in their model each task executes on a single processor and may require only local non-preemptive resources. Their model was extended in [15] to include tasks which can synchronize on and generate multiple events. They allow tasks to execute concurrently on different nodes, but do not enforce parallel execution, while in this paper we assume parallel provision of all resources required by a task segment.

B. Multiprocessor scheduling of parallel tasks

While under multiprocessor scheduling of sequential tasks each task executes on exactly one processor at a time, under parallel task scheduling a task needs to execute on several preemptive resources (e.g. processors) or non-preemptive resources (e.g. graphical processing units) simultaneously.

A well-known method for addressing the parallel task scheduling problem is called *gang scheduling*. It was first introduced in [1], and later discussed among others in [16], [17]. In its original formulation it was intended for scheduling concurrent jobs on large multiprocessor systems.

The work on parallel task scheduling with real-time constraints dates back to [18], where the authors extend Amdahl's law [19] to include communication delay. They estimate the lower and upper bounds on speedup in a multiprocessor system and propose a method for estimating the response time of parallel tasks which incorporates the communication delays. They assume a uniform distribution of workload between the processors. In contrast, in this paper we target systems with arbitrary workload distribution.

More recently, the authors in [2] present a homogenous multi-processor scheduling algorithm for independent tasks which encourages individual threads of a multi-threaded task to be scheduled together. They observe that when such threads are cooperative and share a common working set, this method enables more effective use of on-chip shared caches resulting from fewer cache misses. They consider a multi-core architecture with symmetric single-threaded cores and a shared L2 cache. They employ the global PD^2 and EDF schedulers. Notice that their algorithm only "encourages" individual threads of a multithreaded task to be scheduled together, unlike gang scheduling, which guarantees that these threads will be scheduled together. Also, the threads belonging to the same task may have different execution times, but a common period.

The authors of [20] present a schedulability analysis for preemptive EDF gang scheduling on multiprocessors. They assume that a task τ_i requires a subset of m_i homogenous processors.

The authors of [21] adopt the *basic fork-join* model. A task starts executing in a single master thread until it encounters a *fork* construct. At that moment it spawns multiple threads which execute in parallel. A *join* construct synchronizes the parallel threads. Only the master thread can fork and join. A task can therefore be modeled as an alternating sequence of single- and multi-threaded subtasks. Both [20] and [21] assume fully preemptive and independent tasks.

The authors of [22] address the problem of scheduling independent periodic parallel tasks with implicit deadlines on multi-core processors. They propose a new task decomposition method that decomposes each parallel task into a set of sequential tasks and prove that their task decomposition achieves a resource augmentation bound when the decomposed tasks are scheduled using global EDF and partitioned deadline monotonic scheduling, respectively. They do not consider shared resources.

III. System model

In this section we introduce our system model, comprised of the resource model and the application model.

A. Resource model

Definition 1 (Multi-unit resource)**.** *Let $\mathcal{R}$ be the set of all resources in the system. A multi-unit resource $r \in \mathcal{R}$ consists of multiple units, where each unit is a serially accessible entity. A resource r is specified by its capacity $N_r \geq 1$, which represents the maximum number of units the resource can provide simultaneously.*

Memory space is an example of a *multi-unit resource*. In this paper, when talking about the memory space resource we are interested in the memory requirements in terms of memory size, and ignore the specifics of memory allocation and the actual data stored in the memory. A memory, managed as a collection of fixed sized blocks with no external fragmentation, can be regarded as a multi-unit resource with capacity equal to the number of blocks. In this sense our multi-unit resource is similar to a multi-unit resource discussed by [13].

The capacity of a multi-unit resource represents essentially the maximum number of tasks which can use the resource simultaneously. A multi-core processor can therefore be modeled as a resource with capacity equal to the number of cores.

A preemption is the change of ownership of a resource unit before the owner is ready to relinquish the ownership. In terms of the traditional task model, a job (representing the ownership of a resource) may be preempted by another job before it completes. We can classify all resources in one of two categories:

Definition 2 (Preemptive resource)**.** *The usage (or ownership) of a unit of a preemptive resource can be preempted without corrupting the state of the resource. We use $\mathcal{P} \subseteq \mathcal{R}$ to denote the set of all preemptive resources in the system.*

Definition 3 (Non-preemptive resource)**.** *The usage (or ownership) of a unit of a non-preemptive resource may not be preempted without the risk of corrupting the state of the resource. We use $\mathcal{N} \subseteq \mathcal{R}$ to denote the set of all non-preemptive resources in the system.*

Every resource is either preemptive or non-preemptive, i.e.

$$(\mathcal{N} \cup \mathcal{P} = \mathcal{R}) \wedge (\mathcal{N} \cap \mathcal{P} = \emptyset). \tag{1}$$

A processor is an example of a preemptive resource, as the processor state of a running task can be saved upon a preemption and later restored. A bus or a logical resource (e.g. a shared variable) are examples of a non-preemptive resource.

In the remainder of this paper we assume that all preemptive resources are single-unit, i.e.

$$\forall r \in \mathcal{P} : N_r = 1. \tag{2}$$

B. Application model

We consider a set of n synchronous, periodic, parallel tasks denoted by $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is characterized by a sequence of segments S_i, where the j-th segment $\tau_{i,j} \in S_i$ is specified by its worst-case execution time $E_{i,j}$, and a set of resource requirements $R_{i,j}$. Each resource requirement $(r, n) \in R_{i,j}$ represents a requirement for n units of resource r. During runtime, when a segment $\tau_{i,j}$ is scheduled, all its

required resources must be provided simultaneously for the entire duration of $E_{i,j}$. Our task therefore models programs which can be expressed as a sequence of segments, where each segment $\tau_{i,j}$ is wrapped between a $lock(R_{i,j})$ and $unlock(R_{i,j})$ operation. The semantics of these operations is similar to the primitives used in [12] for locking resources collectively.

A task τ_i is further specified by its fixed and unique priority π_i (lower number indicating higher priority), period T_i, which specifies the inter-arrival time between two consecutive instances of task τ_i, and relative deadline D_i, with $D_i \leq T_i$.

We will use $\mathcal{S}$ to denote the set of all segments among all the tasks, i.e. $\mathcal{S} = \bigcup_{\tau_i \in \Gamma} S_i$.

To keep the notation short, if a segment $\tau_{i,j}$ requires only single-unit resources we will write $R_{i,j} = \{r_1, r_2, r_3\}$ instead of $R_{i,j} = \{(r_1, 1), (r_2, 1), (r_3, 1)\}$. We will also use the shorthand notation $r \in R_{i,j}$ instead of writing $r \mid (r, n) \in R_{i,j}$.

IV. Recap of the MSRP protocol

In this section we summarize MSRP [3], which forms the bases for the PSRP algorithm proposed in this paper.

MSRP is an extension of SRP [13] to multiprocessors. The authors in [3] assume partitioned multiprocessor scheduling, meaning that each task is statically allocated to a processor. Depending on this allocation, they distinguish between local and global resources: *local resources* are accessed by tasks assigned to the same processor, while *global resources* are accessed by tasks assigned to different processors. The MSRP protocol is defined by the following five rules[1]:

1) For local resources, the algorithm is the same as the SRP algorithm. In particular, for every local resource r we define a resource ceiling $\varphi(r)$ greater or equal to the maximum priority among the tasks using the resource, and for every processor p we define a system ceiling $\Pi(p)$ which at any moment is equal to the maximum resource ceiling among all resources locked by the tasks on p. A task is allowed to preempt a task already executing on p only if its priority is higher than $\Pi(p)$.
2) Tasks are allowed to access local resources through nested critical sections. It is possible to nest local and global resources. However, it is not possible to nest global critical sections; otherwise a deadlock can occur.
3) For each global resource r, every processor p defines a resource ceiling $\varphi(r)$ greater than or equal to the maximum priority of the tasks on p.
4) When a task τ_i, allocated to processor p accesses a global resource r, the system ceiling $\Pi(p)$ is raised to $\varphi(r)$ making the task non-preemptable. Then, the task checks if the resource is free: in this case, it locks the resource and executes the critical section. Otherwise, the task is inserted in r's global FIFO queue, and then performs a spin-lock.
5) When a task τ_i, allocated to processor p, releases a global resource r, the algorithm checks the corresponding FIFO queue, and, in case some other task τ_j is waiting, it grants access to r, otherwise r is unlocked. Then, the system ceiling $\Pi(p)$ is restored to the previous value.

[1]In this paper we consider fixed-priority scheduling, so we ignore the preemption levels in SRP, which are needed for EDF scheduling.

Our PSRP algorithm presented in the following section differs from MSRP in the following ways:

- MSRP disallows nested global critical sections, allowing a task to acquire only a single global resource at a time. PSRP supports global nested critical section by allowing each segment to acquire several global resources, effectively shifting the inner critical sections outward [12].
- PSRP supports multi-unit non-preemptive resources, while MSRP supports only single-unit non-preemptive resources.
- PSRP allows a task segment to require several preemptive resources (e.g. several processors in parallel), while under MSRP each task requires exactly one preemptive resource.
- Under MSRP, segments requiring global resources execute non-preemptively. In our model we extend the notion of a global resource, allowing to schedule parallel segments requiring several preemptive resources non-preemptively.

V. Parallel-SRP (PSRP)

We present the Parallel-SRP (PSRP) algorithm, which is inspired by MSRP and can be regarded as its generalization to the parallel task model.

A. Local vs. global resources

MSRP distinguishes between local and global resources. A resource is called local if it is accessed only by tasks assigned to the same processor, otherwise it is called global. When task τ_i tries to access a local resource which is already acquired by another task τ_j (executing on the same processor), τ_i must be suspended to allow τ_j to continue, so that eventually the resource will be released. When task τ_i tries to access a global resource which is already acquired by another task τ_j executing on a different processor, then we have two options for τ_i: we can either suspend it and allow another task assigned to the same processor to do useful work while τ_i is waiting, or we can have τ_i perform a spin-lock (also called a "busy wait").

In either case, the blocking time has to be taken into account in the schedulability analysis. When a task suspends on a global resource, it allows lower priority tasks to execute and acquire other resources, potentially leading to priority inversion. When a task spins on a global resource, it wastes the processor time which could have been used by other tasks. It therefore makes sense to distinguish between local and global resources and to treat them differently.

Similar to MSRP, our PSRP algorithm relies on the notion of local and global resources. Unfortunately, the definition of local and global resources in Section IV assumes that a task requires exactly one processor, and hence it is not sufficient for our parallel task model. We therefore generalize the notion of

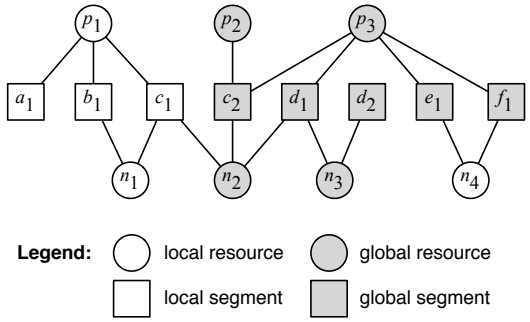

Fig. 1. Example illustrating local and global resources and segments in a segment requirements graph, for a system comprised of $\mathcal{P} = \{p_1, p_2, p_3\}$, $\mathcal{N} = \{n_1, n_2, n_3, n_4\}$, $\mathcal{S} = \{a_1, b_1, c_1, c_2, d_1, d_2, e_1, f_1\}$, $\Gamma = \{a, b, c, d, e, f\}$ with $S_a = \langle a_1 \rangle$, $S_b = \langle b_1 \rangle$, $S_c = \langle c_1, c_2 \rangle$, $S_d = \langle d_1, d_2 \rangle$, $S_e = \langle e_1 \rangle$, $S_f = \langle f_1 \rangle$, and $R_{a_1} = \{p_1\}$, $R_{b_1} = \{p_1, n_1\}$, $R_{c_1} = \{p_1, n_1, n_2\}$, $R_{c_2} = \{p_2, p_3, n_2\}$, $R_{d_1} = \{p_3, n_2, n_3\}$, $R_{d_2} = \{n_3\}$, $R_{e_1} = R_{f_1} = \{p_3, n_4\}$.

local and global resources. The essential property of a global resource is that it is required by segments which can attempt to access their resources independently of each other (e.g. segments which are not synchronized one shared processor).

Definition 4 (Local and global resources)**.** *We define a resource r as* local *if (i) it is preemptive and accessed only by segments which require only non-preemptive resources besides r, or (ii) it is non-preemptive and accessed only by segments which also share one and the same preemptive resource p. Otherwise the resource is* global*. We use $\mathcal{R}^L$ and $\mathcal{R}^G$ to denote the sets of local and global resources, respectively.*

Notice that the local/global classification in MSRP is limited only to non-preemptive resources, while in our definition it also includes preemptive resources. Figure 1 illustrates the difference between local and global resources.

B. Local vs. global segments

Similarly to MSRP distinguishing between local and global critical sections (guarding access to local and global resources, respectively), in PSRP we distinguish between local and global segments.

Definition 5 (Local and global segments)**.** *We define a local segment as one requiring at least one local preemptive resource, otherwise the segment is global. We use $\mathcal{S}^L$ and $\mathcal{S}^G$ to denote the sets of local and global segments, respectively.*

Figure 1 illustrates the difference between local and global segments. The intention of PSRP is to schedule segments a_1, b_1, c_1 preemptively and in priority order, and c_2, d_1, d_2, e_1, f_1 non-preemptively and in FIFO order.

Resource holding time is the duration of a continuous time interval during which a segment owns a resource, preventing other segments to access it. Minimizing the holding time is important in the schedulability analysis, as it adds to the blocking time. Non-preemptive scheduling of global segments will keep the holding times of global resources short. Our choice for executing global segments in FIFO order is in line with MSRP.

C. The PSRP algorithm

Under PSRP, a segment may be *ready*, *executing*, *waiting* (on a global resource), or *blocked* (on a local resource). A segment may be both waiting and blocked at the same time. A task job inherits the state from its currently active segment.

The PSRP algorithm follows the following set of rules:

1) For local resources the algorithm is the same as SRP. In particular, for every local non-preemptive resource $r \in \mathcal{R}^L \cap \mathcal{N}$, we define a *resource ceiling* $\varphi(r)$ to be equal to the highest priority of any task requiring r. For every local preemptive resource $p \in \mathcal{R}^L \cap \mathcal{P}$ we define a system ceiling $\Pi(p)$ which at any moment is equal to the maximum resource ceiling among all local non-preemptive resources locked by any segment that also locks p. We also equip p with a *ready queue* $queue(p)$, which stores tasks waiting for or executing on p sorted by priority. The task at the head of $queue(p)$ is allowed to preempt a task already executing on p only if its priority is higher than $\Pi(p)$.
2) Each global resource $r \in \mathcal{R}^G$ is equipped with a FIFO *resource queue* $queue(r)$[2]. The resource queue stores tasks which are waiting for or executing on the resource.
3) When a task τ_i attempts to lock a set of resources R using $lock(R)$, it is inserted into the resource queues of all global resources in R. Moreover, this insertion is atomic, meaning that no other task can be inserted into any of the resource queues in R before τ_i has been inserted into all queues in R.
   When a task τ_i releases a set of resources R using $unlock(R)$, it is removed from the resource queues of all global resources in R. Each $unlock(R)$ must be preceded by a $lock(R)$ call, with the same R.
4) A task τ_i is said to be *ready* at time t if for all resources r required by its currently active segment $\tau_{i,j}$: r has enough units available, and if $r \in \mathcal{R}^G \cup (\mathcal{R}^L \cap \mathcal{P})$ then τ_i is at the head of $queue(r)$, and if $r \in \mathcal{R}^L \cap \mathcal{P}$ then $\pi_i < \Pi(r)$.
5) If after adding a task τ_i to the queue of resource r the head of the queue has changed (i.e. if τ_i ends up at the head), or if after removing a task from a resource queue the queue is not empty, then the scheduler checks if the head task is ready. If so, then the task is scheduled and becomes executing. Otherwise, τ_i performs a spin-lock (at the highest priority) on each resource containing τ_i at the head of its queue and becomes waiting[3].
   Notice that if a task τ_i requires several units of a resource r, then it will spin-lock until enough of the segments currently using r have completed and released a sufficient number of units of r for τ_i to continue.
6) The following invariant is maintained: the system ceiling of each local preemptive resource is equal to the top

[2]MSRP also equips each global resource with a FIFO queue.

[3]On some resources "spinning" may not make sense (e.g. spinning on a bus). Spinning essentially "reserves" a resource, preventing other tasks from executing on it, and can be implemented differently on different resources.

priority whenever a segment requiring a global resource is spinning or executing on it.

VI. Schedulability analysis for PSRP

We first show that PSRP does not suffer from deadlock nor from live lock, and then we proceed to bound the worst-case response time (WCRT) of tasks.

Lemma 1. *PSRP does not suffer from deadlock.*

Proof: Since every $lock(R)$ is accompanied by a corresponding $unlock(R)$, with the same R, each locked resource will eventually be unlocked, provided no segment is blocked indefinitely. Access to local resources is synchronized using SRP, which was proved to be deadlock free by [13]. We therefore need to show the absence of dependency cycles when accessing global resources, in particular when (i) segments are waiting for resources and (ii) after they have started executing.

(i) Let us assume a segment $\tau_{i,j}$ is executing $lock(R_{i,j})$ and it needs to wait for some of the resources in $R_{i,j}$. Since the resource queues handle the tasks in FIFO order and since the addition to all queues in $R_{i,j}$ is atomic, a dependency cycle when segment $\tau_{i,j}$ is waiting for resources is not possible.

(ii) Since we have assumed that critical sections do not span across task segments and that all resources required by a segment are provided simultaneously (i.e. either all or none), once a segment starts executing it will be able to complete and release the acquired resources. Hence a deadlock cannot occur. ■

Lemma 2. *PSRP does not suffer from live lock.*

Proof: Similarly to Lemma 1, we need to show the absence of livelock when segments are accessing global resources. In particular, we need to show that every segment $\tau_{i,j}$ requiring global resources will eventually start executing. According to Lemma 1, a segment will not deadlock during its waiting phase, so, as long as other segments waiting in a resource queue in front of it eventually complete, it will eventually start executing. Since each segment needs to execute for a finite amount of time, and since, according to Lemma 1, it will not deadlock during the execution phase either, every segment inserted into a resource queue will have to wait for at most a finite amount of time. Hence a livelock cannot occur. ■

To show that tasks will meet their real-time constraints, we derive the bound on their WCRT. To see if a task is schedulable we check if the WCRT of its last segment, measured from the arrival time of the task, is within the task's deadline.

Lemma 3. *A local segment requires exactly one preemptive local resource.*

Proof: According to Definition 4, segments which share a local resource share exactly one preemptive resource. According to Definition 5, every local segment requires at least one local preemptive resource. Lemma 3 follows. ■

Lemma 4. *A global segment requires at least one global resource.*

Proof: Let s be a global segment. Then, according to Definition 5, all resources required by s are either local non-preemptive or global.

We can safely assume that each segment requires at least one resource. We now show by contradiction that no segment can require only local non-preemptive resources. So, let us assume that there exists a segment x which does require only local non-preemptive resources. According to Definition 4, all segments which require any of the segments in R_x will also require exactly one preemptive resource. But this also holds for segment x itself, which contradicts the assumption that all resources required by x are local non-preemptive.

Since no segment can require only local non-preemptive resources, and since each resource required by a global segment is either local non-preemptive or global, each global segment must require at least one global resource. ■

According to Lemma 3, a local segment requires exactly one preemptive resource. This preemptive resource will dictate the behavior of the local segments sharing it. PSRP will use the priority-ordered ready queue to schedule local segments based on their priority. According to Lemma 4, a global segment requires at least one global resource, and according to Definition 5 it does not require any local preemptive resources. PSRP will use the resource queues attached to the global resources to schedule the global segments non-preemptively in FIFO order. In the remainder of this section we derive an equation for the WCRT for global and local segments.

A. Response time of global segments

A global segment spin-locks and executes on all its required resources at the highest priority. Consequently, as a global segment cannot be preempted, its response time is comprised of three time intervals, as illustrated in Figure 2.

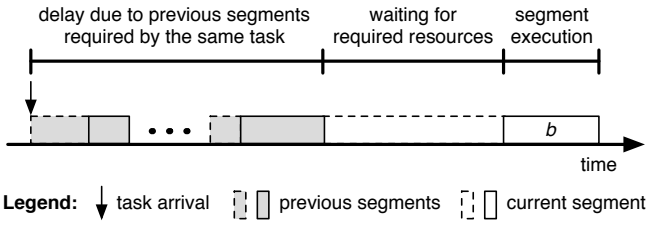

Fig. 2. Response time of a global segment b.

The delay due to segments preceding $\tau_{i,j}$ in S_i is equal to the response time of the previous segment, which can be computed by iterating through the sequence starting with the first segment. The execution time of segment $\tau_{i,j}$ is simply $E_{i,j}$. The interesting part is the time that segment $\tau_{i,j}$ spends waiting for resources in $R_{i,j}$.

The MSRP algorithm assumes that at any time each global segment $\tau_{i,j}$ requires one preemptive and at most one non-preemptive resource. Also, access to a global resource is granted to segments in FIFO order. Consequently, they observe that the worst-case spinning time of segment $\tau_{i,j}$ on a preemptive resource is equal to the sum of the segment execution times of all segments sharing the non-preemptive

resource with $\tau_{i,j}$. In our model, a segment can require an arbitrary number of preemptive and non-preemptive resources, which may result in a longer spinning time.

Definition 6. *The requirements of all segments can be represented by a* segment requirements graph $G = (V, E)$ *where the set of vertices $V = \mathcal{R} \cup \mathcal{S}$, and the set of edges $E \subseteq 2^{\mathcal{S} \times \mathcal{R}}$ represents the resource requirements of segments, i.e.*

$$(\tau_{i,j}, r) \in E \Leftrightarrow (\tau_{i,j} \in \mathcal{S} \wedge r \in \mathcal{R} \wedge r \in R_{i,j}). \quad (3)$$

The graph is tripartite, as we can divide E into two disjoint sets $E^{\mathcal{P}}$ and $E^{\mathcal{N}}$, such that

$$\forall (\tau_{i,j}, r) \in E^{\mathcal{P}} : (\tau_{i,j} \in \mathcal{S} \wedge r \in \mathcal{P}), \quad (4)$$

$$\forall (\tau_{i,j}, r) \in E^{\mathcal{N}} : (\tau_{i,j} \in \mathcal{S} \wedge r \in \mathcal{N}). \quad (5)$$

Example 1 Consider a platform comprised of four processors $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$, executing an application consisting of four tasks, each containing one segment. We name these segments $\mathcal{S} = \{a, b, c, d\}$, and define their resource requirements as follows: $R_a = \{p_1\}$, $R_b = \{p_1, p_2\}$, $R_c = \{p_2, p_3\}$, and $R_d = \{p_3, p_4\}$, as shown in Figure 3. Notice that all resources and segments are global.

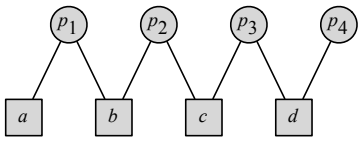

Fig. 3. A segment requirements graph for a system comprised of $\mathcal{P} = \{p_1, p_2, p_3, p_4\}, \mathcal{N} = \emptyset, \mathcal{S} = \{a, b, c, d\}$ with $R_a = \{p_1\}$, $R_b = \{p_1, p_2\}$, $R_c = \{p_2, p_3\}$, and $R_d = \{p_3, p_4\}$.

Let us assume a scenario, where the processors are idle and segments a, b, c, d arrive soon after each other, as shown in Figure 4. When segment a arrives and processor p_1 is idling, it is immediately scheduled and starts executing. When segment b arrives, requiring processors p_1 and p_2, and encounters a busy processor p_1, it is added to the resource queues of $queue(p_1)$ and $queue(p_2)$. Since it is at the head of $queue(p_2)$ it starts spinning on p_2 (at the highest priority). Soon after segment c arrives and similarly is inserted into the resource queues of $queue(p_2)$ and $queue(p_3)$ and starts spinning on p_3. When segment d arrives soon after segment c, it is inserted into $queue(p_3)$ and $queue(p_4)$ and starts spinning on p_4. When segment a completes and releases p_1, it is removed from $queue(p_1)$, enabling segment b, which starts executing. This process continues, subsequently releasing segments c and d. Notice that segment d cannot start executing before c has completed, which cannot start before b has completed, which cannot start before a has completed. □

A segment may be required to wait inside of a resource queue, either passively waiting in the queue's tail or actively spinning at its head. Example 1 suggests that, under PSRP, a segment may need to wait on its required resources until all segments which it "depends on" in the segment requirements

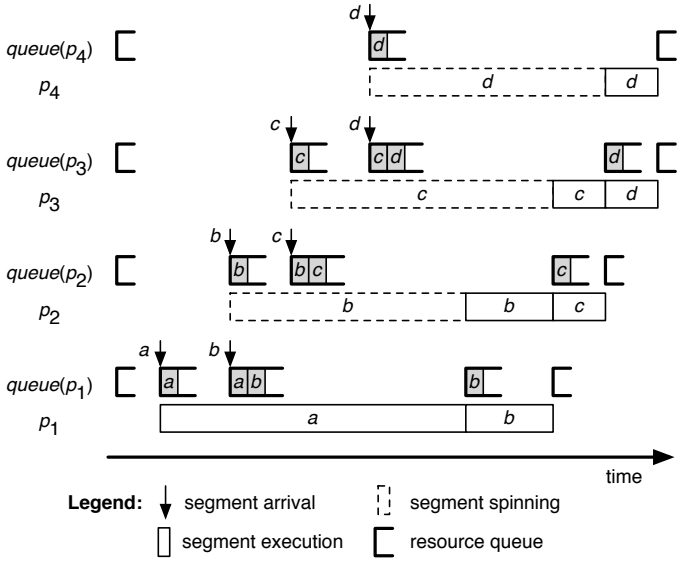

Fig. 4. Example of parallel-chained blocking of global segments. The figure shows the arrival and execution of segments $\mathcal{S} = \{a, b, c, d\}$ on preemptive resources $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$ and the contents of their resource queues. Since we assumed that each task contains only one segment, for ease of presentation we refer to the tasks inside the resource queues by the corresponding segment names.

graph have completed. We can observe, however, that segments belonging to the same task are executed sequentially (by definition), and therefore cannot interfere with each other. The time that segment $\tau_{i,j}$ may need to wait is therefore limited to those segments, which $\tau_{i,j}$ "depends on" if we ignore segments belonging to the same task. Moreover, only segments which require at least one global resource may wait inside of a resource queue. Segments which require only local resources will never be inserted into a resource queue, because resource queues are associated exclusively with global resources.

We now define the notion of a *partial segment requirements graph*, which includes only those dependencies in a segment requirements graph which are indeed feasible. We use these graphs later to formalize the notion of dependency.

Definition 7. *A partial segment requirements graph $G' = (V', E')$ derived from segment requirements graph $G = (V, E)$ is a subgraph of G, with $V' \subseteq V$ and $E' \subseteq E$, such that*

1) V' contains global resources, but no local resources, i.e.

$$\mathcal{R}^G \subseteq V' \wedge \mathcal{R}^L \cap V' = \emptyset,$$

2) if τ_i requires at least one global resource, then there is exactly one *segment from S_i in V', i.e.*

$$\forall \tau_i \in \Gamma : ((\exists \tau_{i,j} \in S_i : R_{i,j} \cap \mathcal{R}^G \neq \emptyset) \Rightarrow |\{\tau_{i,j} \mid \tau_{i,j} \in V'\}| = 1),$$

3) segments requiring only local resources are ignored

$$\forall \tau_{i,j} \in \mathcal{S} : R_{i,j} \subseteq \mathcal{R}^L \Rightarrow \tau_{i,j} \notin V',$$

4) E' contains all the edges (and only those edges) from E which have both endpoints in V', i.e.

$$\forall \{a, b\} \in E : (a \in V' \wedge b \in V') \Leftrightarrow \{a, b\} \in E'.$$

Condition 1 in Definition 7 makes sure that segments which require only local resources will be unreachable from global segments. Condition 3 removes those segments from a partial segment requirements graph to keep it concise.

Definition 8. *We define $partial(G)$ as the set of all possible partial segment requirements graphs which can be derived from the segment requirements graph G.*

Figure 5 illustrates the partial graphs derived from the segment requirements graph in Figure 1.

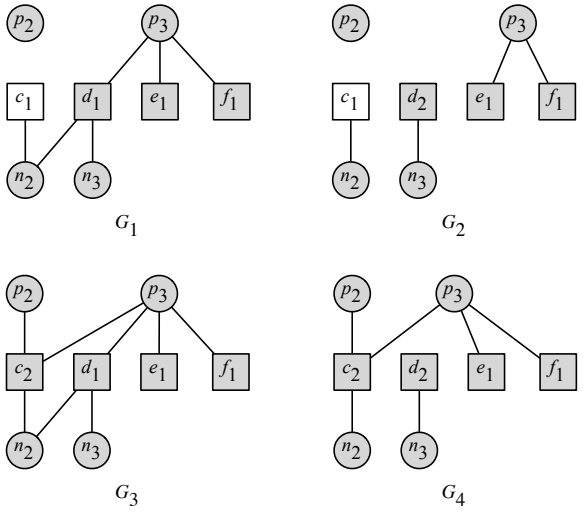

Fig. 5. Partial segment requirements graphs derived from the segment requirements graph in Figure 1, assuming tasks $\Gamma = \{a, b, c, d\}$, with $S_a = \langle a_1 \rangle$, $S_{i,j} = \langle b_1 \rangle$, $S_c = \langle c_1, c_2 \rangle$, $S_d = \langle d_1, d_2 \rangle$.

Definition 9. *Let $G = (V, E)$ be a segment requirements graph. We define $\delta(\tau_{i,j}, g)$ to be the set of segments which $\tau_{i,j}$ can reach in the partial segment requirements graph $g \in partial(G)$. We say that "$\tau_{i,j}$ can reach $\tau_{x,y}$ in g" iff both segments belong to the same connected subgraph of g, and $\tau_{i,j} \neq \tau_{x,y}$. We say that "$\tau_{i,j}$ depends on $\tau_{x,y}$" iff*

$$\exists g \in partial(G) : \tau_{x,y} \in \delta(\tau_{i,j}, g). \tag{6}$$

Notice that the dependency relation is symmetric, i.e.

$$\tau_{x,y} \in \delta(\tau_{i,j}, g) \;\Leftrightarrow\; \tau_{i,j} \in \delta(\tau_{x,y}, g), \tag{7}$$

and transitive, i.e.

$$\tau_{x,y} \in \delta(\tau_{i,j}, g) \wedge \tau_{i,j} \in \delta(\tau_{a,b}, g) \;\Rightarrow\; \tau_{x,y} \in \delta(\tau_{a,b}, g). \tag{8}$$

Example 2 Figure 6 shows an example of the dependencies in the partial segment requirements graphs in Figure 5. □

$\tau_{i,j}$	$\delta(\tau_{i,j}, G_1)$	$\delta(\tau_{i,j}, G_2)$	$\delta(\tau_{i,j}, G_3)$	$\delta(\tau_{i,j}, G_4)$
c_1	$\{d_1, e_1, f_1\}$	$\emptyset$	$\emptyset$	$\emptyset$
c_2	$\emptyset$	$\emptyset$	$\{d_1, e_1, f_1\}$	$\{e_1, f_1\}$
d_1	$\{c_1, e_1, f_1\}$	$\emptyset$	$\{c_2, e_1, f_1\}$	$\emptyset$
d_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
e_1	$\{c_1, d_1, f_1\}$	$\{f_1\}$	$\{c_2, d_1, f_1\}$	$\{c_2, f_1\}$
f_1	$\{c_1, d_1, e_1\}$	$\{e_1\}$	$\{c_2, d_1, e_1\}$	$\{c_2, e_1\}$

Fig. 6. Dependencies for segments in Figure 5, where G_1, G_2, G_3, G_4 represents the partial segment requirements graphs in Figure 5.

Lemma 5. *Under PSRP, each segment $\tau_{i,j} \in \mathcal{S}$ will have to wait on global resources before it can start executing for at most*

$$wait(\tau_{i,j}) = \max\left(0, \max_{g \in partial(G)} \sum_{\tau_{x,y} \in \delta(\tau_{i,j}, g)} E_{x,y}\right), \tag{9}$$

where G is the segment requirements graph.

Proof: Consider the situation when a segment $\tau_{i,j}$ tries to start executing and acquire resources in $R_{i,j}$. If any of the resources is not available, $\tau_{i,j}$ will have to wait. Let $wait\_resource(\tau_{i,j}, r)$ be the worst-case time that segment $\tau_{i,j}$ may spend waiting due to resource r.

When $\tau_{i,j}$ tries to access a global resource r which is not available, then τ_i will be inserted at the end of $queue(r)$. Since $queue(r)$ is a FIFO queue, a segment $\tau_{x,y}$ residing inside of $queue(r)$ in front of $\tau_{i,j}$ will have to complete first, before τ_x can be added at the end of $queue(r)$ again. Hence a task may be represented only once inside of a resource queue, and therefore the length of the resource queue is at most equal to the number of tasks requiring r. In other words, a task τ_x sharing resource r with segment $\tau_{i,j}$ will interfere with $\tau_{i,j}$ (during the time $\tau_{i,j}$ is waiting on r) for the duration of at most one of its segments in S_{τ_x}.

Let $B(\tau_{i,j}, r)$ be the worst-case set of segments which are waiting in $queue(r)$ in front of $\tau_{i,j}$. Each segment $\tau_{x,y} \in B(\tau_{i,j}, r)$ can itself be waiting on other resources: for each resource $s \in R_{x,y}$, segment $\tau_{x,y}$ may need to wait for all segments in $B(\tau_{x,y}, s)$. For each of those segments in $B(\tau_{x,y}, s)$ we can apply the same reasoning. In effect, segment $\tau_{i,j}$ may need to wait for many segments which it indirectly depends on. A straightforward approach would be to designate all segments which are reachable from $\tau_{i,j}$ in G as the set that segment $\tau_{i,j}$ depends on. We now show how to bound this set by removing the unnecessary vertices from G.

(i) The fact that $\tau_{x,y}$ is inside of a resource queue implies that its priority is higher or equal to the system ceiling of any preemptive resource it may require, meaning that it cannot be waiting any more for local resources. We therefore need to consider only global resources.

(ii) At any moment in time only one segment of a task can be active. Therefore, segment $\tau_{i,j}$ will not depend on segments belonging to the same task, i.e segments in $S_i \setminus \{\tau_{i,j}\}$.

(iii) Moreover, segment $\tau_{i,j}$ will not depend on any segment which a segment $\tau_{i,k}$ from the same task depends on, unless $\tau_{i,j}$ also depends on it after removing $\tau_{i,k}$ from G. The same holds for any other segment in $\mathcal{S}$.

According to (i), (ii) and (iii) we need to consider only segments which are reachable from $\tau_{i,j}$ in G, after we remove the vertices corresponding to the local resources, and segments belonging to the same task from G. In other words, segment $\tau_{i,j}$ depends only on segments $\tau_{x,y}$, such that (according to Definition 8 and 9) $\tau_{x,y} \in \delta(r, g)$, where $g \in partial(G)$. Moreover, since (according to Lemma 1) there are no dependency cycles, we need to consider only a single job of each $\tau_{x,y}$.

Segment $\tau_{i,j}$ will have to wait for $wait\_resource(\tau_{i,j}, r)$ time on all resources $r \in R_{i,j}$. Since a segment is inserted into the resource queues of all resources $r \in R_{i,j}$ simultaneously, and any spin-locks are performed concurrently, its total waiting time is given by (9). ∎

Example 3 Figure 7 shows an example of the waiting times for segments in the partial segment requirements graphs in Figure 5 for example values of $E_{i,j}$. □

$\tau_{i,j}$	$E_{i,j}$	$wait(\tau_{i,j})$
c_1	2	3
c_2	2	3
d_1	2	3
d_2	16	0
e_1	0.5	4.5
f_1	0.5	4.5

Fig. 7. Waiting times for segments in Figure 5.

Note that (9) is pessimistic. Figure 8 illustrates the source of the pessimism for $wait(b)$. According to PSRP, segment b may be delayed by a or c, but not both. Lemma 5, however, assumes that in the worst-case b will have to wait for both a and c, which is pessimistic in case a and c do not share a common resource.

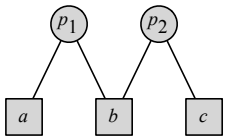

Fig. 8. A segment requirements graph for a system comprised of $\mathcal{P} = \{p_1, p_2\}$, $\mathcal{N} = \emptyset$, $\mathcal{S} = \{a, b, c\}$ with $R_a = \{p_1\}$, $R_b = \{p_1, p_2\}$, and $R_c = \{p_2\}$.

Corollary 1. *A segment $\tau_{i,j}$ which requires* only *local resources will never have to wait inside of a resource queue, i.e.*

$$\forall \tau_{i,j} \in \mathcal{S} : R_{i,j} \subseteq \mathcal{R}^L \Rightarrow wait(\tau_{i,j}) = 0.$$

Definition 10. *For segment $\tau_{i,j}$ we use $E'(\tau_{i,j}) = wait(\tau_{i,j}) + E_{i,j}$ to denote the execution time of $\tau_{i,j}$ extended with its waiting time.*

Definition 11. *We define $A(\tau_{i,j})$ to be the* worst-case activation time *of segment $\tau_{i,j}$ relative to the arrival time of its parent task τ_i. $A(\tau_{i,j})$ is equal to the WCRT of the previous segment in S_i, or 0 in case $\tau_{i,j}$ is the first segment in S_i, i.e.*

$$A(\tau_{i,j}) = \begin{cases} WCRT(\tau_{i,j-1}) & \text{if } j > 1, \\ 0 & \text{otherwise.} \end{cases} \quad (10)$$

Theorem 1. *Under PSRP, the WCRT of a global segment $\tau_{i,j} \in \mathcal{S}^G$, measured since the arrival of the parent task, is bounded by*

$$WCRT(\tau_{i,j}) = A(\tau_{i,j}) + E'(\tau_{i,j}). \quad (11)$$

Proof: Since each segment $\tau_{i,j}$ belonging to task τ_i is dispatched only after the previous segment $\tau_{i,j-1}$ has completed (or when τ_i has arrived, in case of the first segment), and since we assumed $D_i \leq T_i$ for all tasks τ_i, segments belonging to the same task do not interfere with each other. Since each segment is dispatched immediately after the previous one has completed (or at the moment τ_i has arrived, in case of the first segment), there is no idle time between the segments. Therefore, segment $\tau_{i,j}$ will attempt to lock its required resources at time $A(\tau_{i,j})$.

At this moment it will start waiting on all the resources which it requires but which are unavailable. It will wait for at most $wait(\tau_{i,j})$ time units.

Since segments spin at the highest priority, immediately after it stops spinning it will start executing. Also, since we assumed that all nested critical sections have been shifted outwards and since the system ceiling of all resources in $R_{i,j}$ is raised to the top priority at the moment $\tau_{i,j}$ starts executing, segment $\tau_{i,j}$ cannot be preempted nor blocked once it starts executing. In the worst-case it will therefore execute for $E_{i,j}$ time before completing. In order to compute the WCRT of a global segment $\tau_{i,j}$, we therefore simply have to sum up its release jitter, total waiting time and execution time. ∎

B. Response time of local segments

In this section we derive the WCRT of a local segment.

Lemma 6. *Under PSRP, the maximum blocking that a local segment $\tau_{i,j}$ can experience is given by*

$$B(\tau_{i,j}) = \begin{cases} \max\{B^L(\tau_{i,j}), B^G(\tau_{i,j})\} & \text{if } \forall_{r \in R_{i,j}} : r \in \mathcal{R}^L, \\ B^L(\tau_{i,j}) & \text{otherwise.} \end{cases} \quad (12)$$

where

$$B^L(\tau_{i,j}) = \max\{E_{x,y} \mid R_{x,y} \cap \mathcal{R}^G = \emptyset \wedge \pi_x > \pi_i \wedge (\exists r \in R_{x,y} \cap R_{i,j} : \varphi(r) \leq \pi_i)\} \quad (13)$$

$$B^G(\tau_{i,j}) = \max\{E'(\tau_{x,y}) \mid R_{x,y} \cap \mathcal{R}^G \neq \emptyset \wedge \pi_x > \pi_i \wedge R_{x,y} \cap R_{i,j} \neq \emptyset\}. \quad (14)$$

Proof: A local segment $\tau_{i,j}$ can be blocked by local and global segments. Let $B^L(\tau_{i,j})$ and $B^G(\tau_{i,j})$ be the blocking time experienced by $\tau_{i,j}$ due to local and global resources, respectively.

Global segments use *only* global resources (according to Definition 5). Local segments therefore only compete with local segments on local resources. Access to local resources is managed using SRP. According to SRP, segment $\tau_{i,j}$ may be blocked by a lower priority segment only once, before $\tau_{i,j}$ starts executing. Moreover, this blocking time is equal to the length of the longest segment among those which have a lower priority than $\tau_{i,j}$ and share resources with $\tau_{i,j}$. Equation (13) follows.

A local segment $\tau_{i,j}$ which requires only local resources may also be blocked by a lower priority local segment $\tau_{x,y}$ which requires also global resources, when it spin-locks or

executes on those global resources. According to Lemma 3, every local segment uses exactly one local preemptive resource. The PSRP algorithm allows a segment to execute the $lock()$ operation only if its priority is higher than the system ceiling of the local preemptive resource shared with $\tau_{i,j}$. Since the ready segments are scheduled on the preemptive resource according to their priority, $\tau_{i,j}$ can be blocked by only one segment $\tau_{x,y}$ and at most once. Moreover, $\tau_{x,y}$ must have started executing before $\tau_{i,j}$ has arrived, otherwise $\tau_{i,j}$ would have been scheduled instead. According to Lemma 1 the resource holding time of segment $\tau_{x,y}$ on each of its required resources is bounded by $E'(\tau_k)$. Equation (14) follows.

Since (according to Definitions 4 and 5) exactly one preemptive resource will be shared between all local segments sharing resources with $\tau_{i,j}$, this preemptive resource will synchronize the access to all other (non-preemptive) resources required by $\tau_{i,j}$. Segment $\tau_{i,j}$ which requires only local resources can therefore block on either a local segment or a global segment, but not both. The first condition in (12) follows.

A local segment $\tau_{i,j}$, which requires at least one global resource, will start spinning at the highest priority as soon as it reaches the highest priority on the preemptive resource. Since the spinning time is already taken into account in $E'(\tau_{i,j})$, we only need to consider blocking on local segments, and can ignore blocking on global segments. The second condition in (12) follows. ■

Example 4 Applying Lemma 6 to our leading example in Figure 1 (with segment priorities decreasing alphabetically) will result in the following blocking times for local segments:

$$B(a_1) = \max\{E'(b_1), E'(c_1)\},\ B(b_1) = E'(c_1),\ B(c_1) = 0.$$

Notice that Lemma 6 ignores the fact that c_1 may block on d_1, since it is taken into account in the $E'(c_1)$ term in Theorem 2. □

Theorem 2. *Under PSRP, the WCRT of a local segment $\tau_{i,j} \in \mathcal{S}^L$, measured since the arrival of the parent task, is bounded by*

$$WCRT(\tau_{i,j}) = A(\tau_{i,j}) + w(\tau_{i,j}), \tag{15}$$

where $w(\tau_{i,j})$ is the smallest value which satisfies

$$w(\tau_{i,j}) = B(\tau_{i,j}) + E'(\tau_{i,j}) + \sum_{\tau_{x,y} \in X} \left\lceil \frac{w(\tau_{i,j}) + J(\tau_{x,y})}{T_x} \right\rceil E'(\tau_{x,y}), \tag{16}$$

where $J(\tau_{x,y}) = A(\tau_{x,y}) - \sum_{z<y} E_{x,z}$ is the activation jitter of segment $\tau_{x,y}$, and

$$X = \{\tau_{x,y} \mid \pi_x < \pi_i \ \wedge\ (R_{x,y} \cap R_{i,j} \cap \mathcal{R}^L \neq \emptyset)\}$$

is the set of higher priority segments which share a local resource with $\tau_{i,j}$.

Proof: As soon as a local segment is released, it will try to lock all its required resources in $R_{i,j}$. If any of the resources it requires are not available, it will block for $B(\tau_{i,j})$ given by (12). When $\tau_{i,j}$ is ready to resume after the initial blocking, we distinguish between two cases, depending on whether (i) $\tau_{i,j}$ requires only local resources, or (ii) $\tau_{i,j}$ requires at least one global resource.

In case (i), according to Corollary 1, segment $\tau_{i,j}$ will not wait inside of a resource queue, i.e. $wait(\tau_{i,j}) = 0$. During the time that the segment is blocked or executing, higher priority segments sharing local resources with $\tau_{i,j}$ can arrive and interfere with it. These segments must be local too, otherwise, according to Definition 5, segment $\tau_{i,j}$ would have been global. The inter-arrival time between two consecutive invocations of a higher priority segment $\tau_{x,y}$ is equal to its tasks period, with the first arrival suffering an activation jitter $J(\tau_{x,y})$, which can be bounded by the activation time of $\tau_{x,y}$ minus the execution time of all the segments preceeding it in S_x. Equation (16) follows.

In case (ii), during the time $\tau_{i,j}$ is blocked, higher priority segments may arrive. However, since $\tau_{i,j}$ requires a global resource, as soon as it becomes ready to execute it will be inserted into the resource queue of all resources in $R_{i,j}$ and start spinning at the highest priority on the single local preemptive resource which it requires (according to Lemma 3). The spinning time is included in the $E'(\tau_{i,j})$ term in (16). As soon as all the resources in $R_{i,j}$ are available, it will continue executing at the highest priority on the preemptive resource. Therefore, higher priority segments arriving during the time $\tau_{i,j}$ is waiting or executing (i.e. "during" the $E'(\tau_{i,j})$ term) will not interfere with $\tau_{i,j}$. Since in this theorem we are providing an upper bound, equation (16) follows.

A local segment $\tau_{i,j}$ will be delayed (relative to the arrival of its parent task) by the WCRT of the previous segment (if any), represented by the $A(\tau_{i,j})$ term in (15). ■

C. Response time of tasks

Now that we know how to compute the WCRT of local and global segments, we can easily determine the WCRT of tasks.

Corollary 2. *Under PSRP, the WCRT of a task $\tau_i \in \Gamma$ is given by the WCRT of the last segment in S_i.*

Note that the WCRT of a segment depends on the activation time of another segment. In turn, the activation time of a segment depends on the WCRT of another segment. However, since the priority of all segments of a given task is the same, this mutual dependency problem can be solved by simply computing response and activation times in order from the highest priority task to the lowest priority task.

VII. Evaluation

In this section we demonstrate the effectiveness of PSRP in exploiting the inherent parallelism of a platform comprised of multiple heterogeneous resources. We consider a task set where some task segments require several processors at the same time and also share non-preemptive resources. We schedule it using two approaches: (i) using PSRP, and (ii) by collapsing all the processors into one virtual processor, i.e. by treating the entire platform as a single resource, and applying uniprocessor scheduling (which we refer to as "Collapsed").

To the best of our knowledge, the second approach is currently the best alternative to PSRP for scheduling parallel tasks which can execute on arbitrary subsets of processors and share non-preemptive resources. We compute the WCRT of the complete task for the two approaches. The difference in response times represents potential utilization gain, which can be exploited by e.g. background tasks or tighter timing requirements.

The simulated task set Γ represents a multimedia application, where video frames are captured periodically with period T and subsequently processed by a set of filters. Some of the filters are computationally intensive, but can exploit functional parallelism and execute on several processors in parallel. The video frames are stored in a shared global memory. Each parallel filter loads the necessary frame data from the global memory into its local buffer, operates on it, and writes the result back to the global memory. The data is transferred using a DMA controller. The simulated platform corresponds to a PC with a multicore processor. For simplicity we assume no caches.

PSRP approach: The platform consists of M processors p_j, $1 \leq j \leq M$, a global memory m, local memories accessible by individual processors (or groups of processors) where m_i represents the memory region in a local memory allocated to task τ_i, and a DMA controller *dma* for transferring data between the global and local memories. It can be expressed in terms of our model as $\mathcal{P} = \{p_1, p_2, \ldots, p_M\}$ and $\mathcal{N} = \{dma, m, m_1, m_2, ..., m_{|\Gamma|}\}$.

We consider several scenarios. In each scenario we divide the processors into H groups $\mathcal{P}_g$, $1 \leq g \leq H$. Each group contains W processors, with $H * W = M$. On each group of processors we execute a set of K parallel tasks. Each parallel task τ_i belonging to group g is specified by $S(\tau_i) = \langle(0.5, \{dma, m, m_i\}), (5, \mathcal{P}_g \cup \{m_i\}), (0.5, \{dma, m, m_i\})\rangle$. On each processor $p_j \in \mathcal{P}$ we also execute a sequential task τ_i with $S(\tau_i) = \langle(2, \{p_j, m_i\})\rangle$. All tasks share the same period T, $\forall \tau_i \in \Gamma : D_i = T_i$, and the parallel tasks have higher priority than the sequential tasks.

Collapsed approach: We can model the Collapsed approach by replacing all processors by one preemptive resource p and having each segment require at least the resource p. For a scenario with H, W and K defined above, the "collapsed" task set then consists of $H * K$ tasks τ_i with $S(\tau_i) = \langle(0.5, \{p, dma, m, m_i\}), (5, \{p, m_i\}), (0.5, \{p, dma, m, m_i\})\rangle$, and $H * W$ tasks τ_i with $S(\tau_i) = \langle(2, \{p, m_i\})\rangle$.

Figure 9 compares the maximum WCRT among all tasks in Γ between the PSRP and Collapsed approaches for $H = 2$. We vary the number of tasks per processor group K and the number of processors required by parallel tasks W. We have computed the WCRT for the PSRP approach using the analysis presented in this paper, and for the Collapsed approach using the Fixed Priority Preemptive Scheduling analysis [23].

The results show that PSRP experiences lower WCRT than the Collapsed approach. Moreover, since the difference in WCRT increases for larger values of K and W, the benefits of PSRP increase with larger task sets and more parallelism, i.e. when tasks execute on more processors in parallel. The results therefore demonstrate that PSRP indeed outperforms the Collapsed approach.

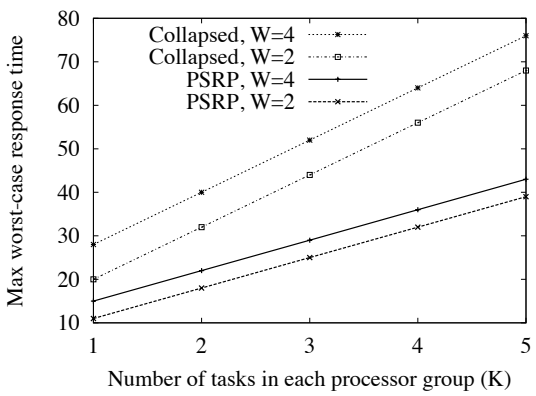

Fig. 9. Comparison of WCRTs for cases (i) and (ii) for $H = 2$ and varying W and K.

VIII. Discussion

In this section we discuss the pros and cons of the proposed system model and PSRP.

A. Multi-unit preemptive resources

We can model a homogenous multiprocessor containing n cores in two ways: as a preemptive resource p with $N_p = n$, or as n preemptive resources $p_1, p_2, \ldots, p_n$, with capacities $N_{p_i} = 1$, for all $1 \leq i \leq n$. Existing literature on parallel task scheduling on multiprocessors assumes the first option, where each task segment specifies a requirement for a number of units of a multi-unit resource p. The system is then responsible for allocating tasks to processors during runtime. This model will ignore potentially large migration overheads, e.g. in memory intensive applications as data locality cannot be guaranteed. Using the second approach, our model allows to partition the task set upfront, e.g. optimizing data locality.

B. Non-preemptive execution on preemptive resources

When a preemptive resource p is required by a segment which requires also another preemptive resource, then p is marked as a global resource, resulting in non-preemptive execution on p. This may appear overly pessimistic, especially compared to the work in [20] which describes a preemptive gang scheduling algorithm. However, they assume independent tasks. In multiprocessor scheduling with shared resources it is critical to keep the holding time of global non-preemptive resources as short as possible (which is the rationale between the spin-lock based approach to locking global resources in MSRP). If we were to schedule all preemptive resources preemptively, then segments requiring several preemptive resources in a chained fashion (illustrated in Figure 3) would increase the resource holding time. We therefore decided to limit preemptive execution to preemptive resources which are required by segments which do not require any other preemptive resource.

C. Pessimistic analysis for local segments

Theorem 2 describes the WCRT of a local segment. It treats all local segments alike, whether they require global resources or not. However, only a local segment which requires *only* local resources can be preempted by higher priority segments while it is executing. A segment which requires at least one global resource will be scheduled non-preemptively on all preemptive resources it requires. We can therefore lower the bound on WCRT of local segments which require global resources by ignoring the interference of higher priority tasks during the execution of those segments. For this purpose we can adopt the schedulability analysis for Fixed-Priority with Deferred Preemption Scheduling by [24].

D. Nested global critical sections

FMLP supports nested critical sections by means of resource groups. The resource groups partition the set of resources into independent subsets. Consequently, a task trying to access resource r may become blocked on *all* resources in the resource group $G(r)$.

Under PSRP, if a task has nested critical sections we can move the inner critical sections outwards until they overlap exactly with the outer most critical section. This new task can be expressed in our system model, where segments require several resources at the same time. Each segment can be blocked only on resources which it requires (rather than the complete resource group). PSRP therefore provides a more flexible approach for dealing with nested global critical sections than FMLP. In the worst case, under FMLP a segment requiring resource r will be blocked by every task for the duration of the longest segment which requires a resource from $G(r)$, while under PSRP a segment will be indirectly blocked by all dependent segments (see Figure 3).

Under FMLP every time a job is resumed it may block on a local resource. This is the same for PSRP, where we have to include the blocking time for each segment (rather than once per task).

IX. Conclusion

In this paper we addressed the problem of multi-resource scheduling of parallel tasks with real-time constraints. We proposed a new resource model, which classifies different resources (such as bus, processor, shared variable, etc.) as either a preemptive or non-preemptive multi-unit resource. We then presented a new scheduling algorithm called PSRP and the corresponding schedulability analysis. Simulation results based on an example application show that it can exploit the inherent parallelism of a platform comprised of multiple heterogeneous resources.

Currently, PSRP requires that the preemptive resources have only a single unit. In the future we want to extend PSRP to handle multi-unit preemptive resources.

References

[1] J. Ousterhout, "Scheduling techniques for concurrent systems," in *International Conference on Distributed Computing Systems*, 1982, pp. 22–30.

[2] J. H. Anderson and J. M. Calandrino, "Parallel real-time task scheduling on multicore platforms," in *Real-Time Systems Symposium (RTSS)*, December 2006, pp. 89 –100.

[3] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Real-Time Systems Symposium (RTSS)*, 2001, pp. 73–83.

[4] E. W. Dijkstra, "The mathematics behind the Banker's Algorithm," in *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982, pp. 308–312.

[5] A. N. Habermann, "Prevention of system deadlocks," *Commun. ACM*, vol. 12, pp. 373–382, July 1969.

[6] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Real-Time Systems Symposium (RTSS)*, 1988, pp. 259 –269.

[7] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007, pp. 47–56.

[8] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2003, pp. 189–198.

[9] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008, pp. 342–353.

[10] B. B. Brandenburg and J. H. Anderson, "A comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}," in *International Conference on Principles of Distributed Systems (OPODIS)*, 2008, pp. 105–124.

[11] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Real-Time Systems Symposium (RTSS)*, 2009, pp. 469 –478.

[12] J. W. Havender, "Avoiding deadlock in multitasking systems," *IBM Systems Journal*, vol. 7, no. 2, pp. 74 –84, 1968.

[13] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[14] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, no. 2-3, pp. 117–134, 1994.

[15] J. J. G. García, J. C. P. Gutiérrez, and M. G. Harbour, "Schedulability analysis of distributed hard real-time systems with multiple-event synchronization," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2000, pp. 15–24.

[16] D. G. Feitelson, "Distributed hierarchical control for parallel processing," *Computer*, vol. 23, no. 5, pp. 65–77, 1990.

[17] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 306 – 318, 1992.

[18] X. Li and M. Malek, "Analysis of speedup and communication/computation ratio in multiprocessor systems," in *Real-Time Systems Symposium (RTSS)*, dec 1988, pp. 282 –288.

[19] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Spring joint computer conference*, 1967, pp. 483–485.

[20] S. Kato and Y. Ishikawa, "Gang EDF scheduling of parallel task systems," in *Real-Time Systems Symposium (RTSS)*, 2009, pp. 459 –468.

[21] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium (RTSS)*, 2010, pp. 259 – 268.

[22] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Real-Time Systems Symposium (RTSS)*, 2011, pp. 217–226.

[23] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[24] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2007, pp. 269–279.