Accepted for SEUS 2014 - 10th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, Reno, NV, USA, Jun 9, 2014

Resource Synchronization in Hierarchically Scheduled Real-Time Systems using Preemptive Critical Sections

Tom Springer, Steffen Peter, and Tony Givargis
Center for Embedded Computer Systems
University of California, Irvine, USA
{tspringe, st.peter, givargis}@uci.edu

***Abstract*— In this paper we outline a novel approach for accessing mutually exclusive resources in hierarchically scheduled real-time systems. Our method known as the Resource Access Control Protocol with Preemption (RACPwP) is an improved resource allocation protocol which utilizes preemptive critical sections to provide guaranteed determinism for hard real-time tasks and comparable response times for soft real-time tasks. Our experiments demonstrated that RACPwP outperforms other state-of-the-art resource access control protocols used in hierarchically scheduled systems. RACPwP was implemented as part of VxWorks and evaluated in an actual embedded application used in the aerospace industry. As a result, the response times for hard real-time tasks were improved over a traditional resource synchronization protocol.**

Keywords—real-time systems, hierarchical level scheduling, resource access control, resource preemption and rollback.

I. Introduction

Real-time systems that can more effectively adapt to their changing environment are getting increased attention by embedded system researchers [1]. As embedded computing devices become more prevalent they are increasingly being introduced into environments where their requirements are unstable and difficult to predict. This uncertainty is due in part, to the fact that modern embedded systems are being composed with a more open source architectural approach then the tightly-coupled systems of the past. However, these general-purpose architectures are typically designed to optimize average performance and, as a result introduce large fluctuations to the task execution times. Furthermore, specific applications can also contribute to these variable execution times. Consider a digital processing application, such as a software defined radio. Specific waveforms have different processing requirements which could introduce various task execution times. As a result, the overall workload of the computing system could vary significantly which may create an overload condition and degrade the performance of the overall system.

One effect that could occur in an overloaded system is that tasks can start missing deadlines. Researchers [2] have shown that a system which is overloaded while being managed by the Earliest Deadline First (EDF) scheduling algorithm [3] could effectively cause all the other tasks in the system to miss their deadlines. This is known as the “domino effect” and is depicted in Figure 1 where the execution of task τ_0 generates an overload condition causing all the tasks to miss their deadlines. Similarly, systems that are overloaded and scheduled by the Rate Monotonic (RM) scheduling algorithm [3] may cause all lower priority tasks to miss their deadlines. Consequently, it is much more likely with today’s embedded systems, which are typically resource constrained, that an overload condition of a single application could adversely affect the whole system. The traditional solution has been to “over-engineer” for worst case scenarios. The problem with this approach is the system becomes under-utilized and a valuable resource sits idle for most of the time.

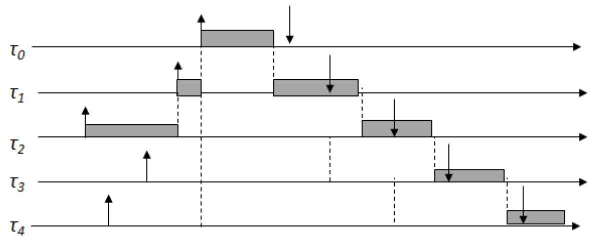

Figure 1: Domino Effect of a Transient Overload

Therefore, it is particularly critical for embedded systems to more effectively adapt to overload conditions. The real-time system needs to be able to react to load variations without it adversely affecting the other tasks in the system. There needs to be a type of temporal isolation where a temporarily overloaded task will not cause the domino effect.

Hierarchical scheduling is a framework that has been introduced to provide this temporal protection. The Hierarchical Scheduling Framework (HSF) has shown to be particularly useful in the area of open systems [4] where applications can be developed, integrated and validated independently. A primary goal of hierarchical scheduling is to bind the temporal behavior of those applications whose execution times deviate considerably, allowing for the predictable operation of the various subsystems.

In order to provide this temporal isolation the basic HSF model must assume that each subsystem is independent, however most systems are not entirely independent and resource sharing may be needed locally (within the same subsystem) or globally (across subsystems) for correct behavior. While traditional resource access protocols can be used to synchronize resources locally global resource access presents added challenges such as the unpredictable holding times between globally shared resources.

In this paper we present the Resource Access Control Protocol with Preemption (RACPwP) which is compatible with a HSF but provides better temporal isolation between globally shared resources as compared to other resource synchronization algorithms. Our approach utilizes software transactional memory to support preemption inside critical sections. The result being improved response times for higher-priority tasks and comparable average case response times for soft real-time tasks. Therefore the contributions of this paper are as follows:

- A novel resource allocation protocol which allows for the preemption of mutually exclusive resources via software based transaction mechanisms.
- Extension of the HSF model for support of hard real-time system requirements.
- Schedulability analysis of RACPwP as compared to other state-of-the-art synchronization protocols.
- An implementation of the RACPwP protocol as part of the VxWorks real-time operating system.
- Demonstrated practicability of the RACPwP protocol in an actual ground-based command/control embedded system used in the aerospace industry.

A. *Organization of the paper*

The sections of the paper are organized as follows: Section II provides an overview of the hierarchical scheduling framework along with its current limitations. Section III summarizes some of the related work in hierarchical scheduling along with the current approaches employed for resource access control. Section IV discusses our approach to resource synchronization in hierarchically scheduled systems along with the preemptable critical section mechanism used by RACPwP. Section V provides schedulability analysis between RACPwP and other state-of-the-art resource access controls used in hierarchical scheduled systems. A brief architectural overview of the command/control embedded system is presented in Section VI along with a comparison between RACPwP and the more traditional resource access control protocol. Section VII provides a brief summary and discussion for potential future work.

II. Preliminaries

A. *Hierarchical Scheduling*

The basic architecture of a HSF is generally represented as a two-level tree of nodes [6]. The root node represents the overall system while the leaf nodes represent the various subsystems. Therefore, a hierarchical scheduled system consists of one or more subsystems.

A hierarchically scheduled system provides temporal isolation [5] for each individual subsystem executed on a single processor. Each subsystem is composed of an application and each application could be composed of multiple tasks (see Figure 2). As a result, this scheme allows for different subsystems to use different scheduling algorithms (e.g., one subsystem could use RM scheduling while another subsystem could use EDF scheduling). In this way, each subsystem can employ a scheduling mechanism that is most appropriate for the application. The added benefit of this approach is that applications can be developed and validated independently which makes this framework applicable to the emerging trend of open systems development [5]. For example, the use of a hierarchical scheduling framework allows for a subsystem to be developed with its own scheduling algorithm and then later integrated along with other subsystems using different local and global schedulers.

B. *System Model*

The HSF model consists of a single global system S and up to n local systems S_n such that $S_i \in S$. The global system S contains the global scheduler which controls which subsystem S_i can use the processor while the local scheduler determines which application's task should actually execute.

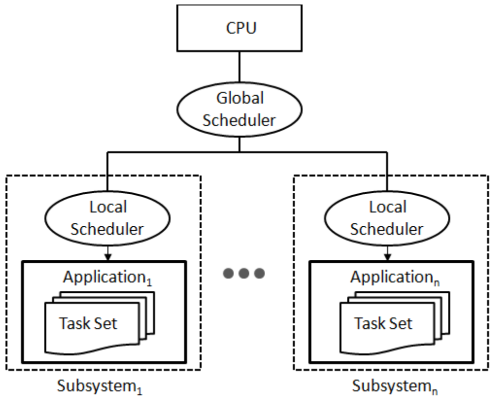

Figure 2: Hierarchical Scheduling Framework

Every application is allocated a separate service manager, known as server. Each server is allocated a CPU capacity reserve, which is assigned as a pair (Q_i, P_i) where Q_i is defined as the time quantum and P_i is defined as the period. Each task gets to execute for the assigned time quantum Q_i, when the server's time quantum Q_i is exhausted the task is blocked until its next period (see Figure 3). Each server has its own priority which is used by the global scheduler to determine which server is allocated to the processor. However, execution of a subsystem server could be delayed or pre-empted as a result of a higher priority subsystem server. The local scheduler, as part of each server, determines which task should execute when the server is re-activated. In effect, the server functions as an independent processor virtually limiting the bandwidth of each application.

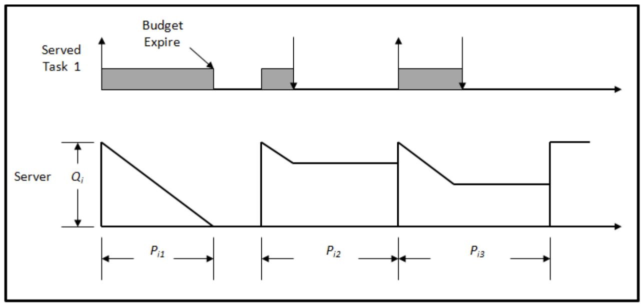

Figure 3: Periodic Server Example

C. Resource Sharing

Resource sharing in a HSF can also be classified as either local or global allocation. Tasks that share resources within the same application or subsystem are considered local resource sharing. Tasks that share resource across applications are classified as global resource sharing.

Local resource sharing can be managed by traditional resource access protocols such as Priority Inheritance Protocol (PIP) [8], the Priority Ceiling Protocol (PCP) [8] or the Stack Resource Policy (SRP) [9]. Global resource sharing, on the other hand, requires that a resource be protected at the local as well as global level. Which means a task that locks a global resource will also cause its server to lock the resource. However, there is an added complication with sharing a global resource across applications.

When a task locks a global resource (which can be locked by tasks in the same subsystem or by tasks in other subsystems) then there is a requirement that the mutual exclusion is honored across subsystems. What this means is the protocol needs to control the execution of the server for each subsystem. Therefore, each subsystem will have to manage local as well as global resource access. Additionally, global resource access will need to block a subsystem server in the case of global resources. However, care must be taken when a task has a global resource locked but the server budget is depleted. Consider the following scenario, illustrated in Figure 4, a high priority task $Task_1$ shares a *mutex* with a lower priority task $Task_2$ which is managed by a simple periodic server. Given a server budget of $Q_i = 4$ and a period of $T_i = 10$, at time *t3*, task $Task_1$ preempts $Task_2$ then is blocked on the critical resource. However, when $Task_2$ resumes execution its server budget is not enough to finish the task requiring $Task_2$ to wait until its budget is replenished, thereby creating an additional delay for task $Task_1$.

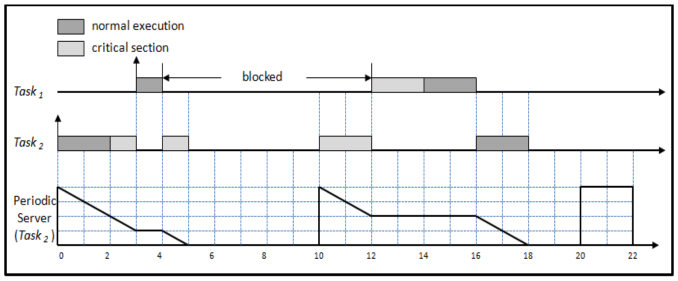

Figure 4: Budget exhaustion inside a critical region with a fixed priority periodic server

Researchers have proposed several solutions to the problem of added delay in critical sections due to server budget exhaustion. One such approach called budget check checks to see if there is sufficient server budget before allowing a task to enter a critical section. If the budget is insufficient the task is not granted access to the resource until the next budget replenishment.

The other approach allows the task to enter a critical section without checking for a sufficient budget. As a result, if the budget is exhausted while still inside the critical section the task is just allowed to continue and consume extra budget until the end of the critical section. There are two slight variations to the protocol on how they handle budget overruns. One variation consumes the extra budget at the expense of other tasks. The result being that other tasks in the subsystem may not receive their full budget allotment. The other method does "payback" to other tasks in the subsystem by taking away a portion of the full budget allotment, of the task that overran, during subsequent replenishment periods.

D. Server Budget Exhauston

The problem of budget exhaustion can be amplified during periods of overload as it could further increase the time a critical task would have to wait for the resource. Overload conditions can result because tasks execute longer than expected. Hierarchical scheduling is a general technique that can be used to limit the effects of overruns in tasks with these variable execution times. However, in the interest of timing guarantees there are distinctions between *hard* and *soft* tasks in a hierarchical scheduled system. A hard reservation allows a task to execute at most Q_i (budget) units of time for every P_i (period), whereas a soft reservation allows the task to execute for at least Q_i time units for every P_i. This way a soft real-time task can execute more if there is some idle time available. The issue with this is when a hard and soft real-time task share a global resource budget overrun allows the soft real-time task to continue affecting the budget for the hard real-time task. This is one reason hierarchical scheduled systems are generally considered only for soft real-time systems, such as video processing. As an example, consider the same task model mentioned in the previous section and the budget overrun depicted in Figure 5.

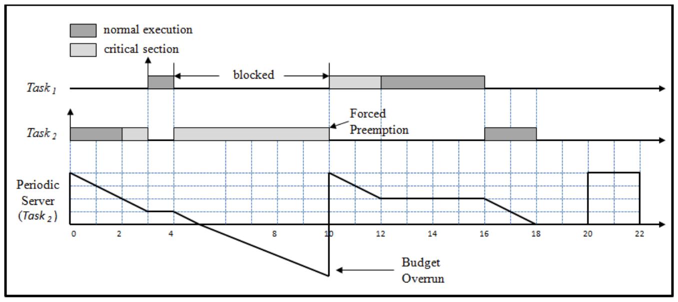

Figure 5: Budget overrun inside a critical section (no payback)

During an overload condition if a served task is allowed to continue while still inside its critical section it can violate the temporal isolation between subsystems. Therefore, our proposal is that current resource sharing algorithms (in hierarchical scheduled systems) are insufficient for hard real-time tasks when a resource is shared across subsystems, specifically during periods of overload.

III. Related Work

The schedulability of HSF has been analyzed using fixed-priority global scheduling [6] and EDF bound global scheduling [7]. Initially, HSF designed systems were meant to be independent but researchers realized this approach was not practical as many embedded systems are semi-independent via the sharing of global resources. Research on the HSF was extended to perform schedulability analysis of semi-independent real-time components [8] [9]. The main focus of this work was to reduce the resource holding times that were being incurred during budget expiration.

The SIRAP [14] protocol was developed for fixed-priority preemptive scheduling while the BROE [15] protocol was developed for dynamic-priority scheduling. Their work used a form of budget check to determine if there was enough budget left to enter the critical section. If the remaining budget was deficient to complete the critical section the task was blocked from locking the resource until the next budget replenishment. The limitation with this approach is that the critical section execution time is based upon worst-case analysis. This could lead to resource under utilization due to conservative WCET estimations. Additionally, *a priori* knowledge of the WCET for a critical section is required which is often difficult to evaluate in applications with variable execution times.

Hierarchical scheduling with resource sharing HSRP [4] and later extended to OPEN-HSRP [12] utilized the budget overrun approach to reduce the resource holding times during budget expiration. Two variations to budget overrun were compared, budget overrun with and without payback. While this approach does provide better flexibility for applications with variable execution times there are some drawbacks. Even though a task is allowed to overrun its budget there still has to be a limit placed upon the maximum overrun time. In order to prevent unbounded blocking a task is forcefully preempted if it is still holding the resource during the next budget replenishment. This leads to limitations being placed upon the types of shared resources used to those that can safely be aborted to relatively short critical section execution times. Another consideration is because a task can overrun its budget the strict temporal isolation between subsystems could be violated. It is for these reasons that HSRP based systems are generally used for soft real-time systems.

Other recently published work, known as RRP [16], took a different approach to the problem, of resource sharing in hierarchical scheduled systems. Instead of performing a budget check the task was allowed to enter the critical section and unlike HSRP if the budget had expired the task was simply preempted and rolled back. The RRP protocol improved the average case response times and task schedulability as compared to SIRAP and the OPEN-HSRP protocols. However, the limitation with RRP it that can only be used with shared resources that can be safely rolled back (i.e. databases).

RACPwP does not rely on WCET analysis of critical section executions times so the protocol is allowed to be more aggressive during task admission which provides improved task schedulability over SIRAP. Given that RACPwP does not use overrun mechanisms higher priority task response time is improved and temporal isolation is strictly enforced as compared to the OPEN-HSRP. Finally, because RACPwP utilizes preemptable critical sections the type of shared resources that can be used is expanded to include other shared resources and not just databases as in RRP.

IV. Resource Access Control Protocol with Preemption

This section provides a description of the RACPwP resource access policy. For the purpose of providing a resource access policy in hierarchical scheduled systems there is a need to maintain a global and local perspective. In order to manage global resources as well as local resources in a HSF RACPwP uses HSRP, this is an extended version of the SRP protocol. A brief overview is provided below for a complete description of SRP and HSRP readers are encouraged to read the references [9] [4].

A. Hierarchical Stack Resource Policy

In order to provide support for global resource management the SRP protocol has been expanded [6]. Each task τ_i has a preemption level $\pi_i = \frac{1}{D_i}$ and each subsystem S_i has an associated preemption level $\prod_s = \frac{1}{P_s}$, where D_i is the relative deadline of the task and P_s is the subsystem deadline. Each globally shared resource R_j is associated with two types of resource ceilings; one for local resource scheduling and one for global resource scheduling.

$$rc_j = max\{\pi_i \middle| T_i\ accesses\ R_j\} \tag{1}$$

The global and local systems system ceilings dynamic and could change during execution. The global resource ceiling is defined as the following:

$$\Pi_s = \max_j \left\{C_{R_j}\right\} \tag{2}$$

Where Π_s is the system ceiling for the global resource $\mathrm{R_j}$. If a task has a global resource locked the priority of the server is raised to the system ceiling Π_s associated with that resource. If a task has a global resource locked then the priority of the task is raised to the highest level priority within the application. If the server capacity is exhausted while a task has a global resource locked the server will overrun until the task has released the resource. If the task does not release the resource before the server's budget is depleted then the server will abort the task regardless of the state of the resource. The reason for taking such a drastic measure as forceful pre-emption and risking leaving a resource in an inconsistent state is a non-preemptable global resource could result in increasing blocking times.

Consider just one long global resource access by a low priority task. The long critical section execution time of the global resource could result in a large blocking factor for all the higher applications. This large blocking factor would apply regardless of whether the resource is shared by the higher priority applications.

B. Protocol Definition

Similar to other hierarchical scheduling frameworks (SIRAP and OPEN-HSRP) the RACPwP protocol is based upon a hierarchical scheduling framework. Unlike SIRAP which performs a budget check before entering a critical section RACPwP always grants access to a global resource. If the task's subsystem budget is depleted while still holding a lock the task is not allowed to continue, as it is in budget overrun mechanisms (e.g. OPEN-HSRP), but instead is preempted.

Comparable to RRP our method utilizes check pointing and rollback to recover from a forced preemption but RACPwP incorporates a new technique known as Preemptable Critical Sections (PCS). The benefit of this approach is it lifts the restriction RRP has by only being able to handle resources that can be aborted or aborted and rolled back (e.g. database applications). The protocol is defined as follows:

- Tasks are scheduled based upon their active priorities. Tasks with the same priority are executed in a first come first served basis.
- When a task τ_i requests a local resource and the resource is available the task's priority is raised to the local ceiling priority of the resource C_{R_j}.
- If task τ_i requests a local resource that is locked then T_i is blocked for the duration of the longest critical section among the tasks that access resource R_j.
- If the subsystem S_i capacity is exhausted while task τ_i has resource R_j locked then the server suspends τ_i.
- If a task τ_i requests a global resource and the resource is available then the subsystem server's S_i priority is raised to the global ceiling priority of the resource.
- If task τ_i requests a global resource that is locked then τ_i is blocked for B_i which is defined as the longest time a task in the same application can execute.
- If server S_i capacity is exhausted while τ_i still has the resource locked then the task is preempted for a maximum duration of its replenishment period.

C. Preemptable Critical Sections

In order to provide a safe mechanism for forceful preemption we introduce a new programming construct for resource synchronization known as preemptable critical sections (PCS). PCS utilizes a form of software transactional memory (STM) to restart a transaction that has been preempted by a higher priority task. After the higher priority task has released the resource the lower-priority task is rolled back and restarted.

The main benefit of this approach is that a higher-priority task gets to execute quickly. In fact, the worst case blocking time is equal to the remaining budget of the lower priority task that is sharing the global resource. Another benefit of this approach includes the elimination of the increased blocking times incurred when a lower-priority task is allowed to overrun its budget as used in the OPEN-HSRP protocol. As a result, RACPwP provides improved response times of hard real-time tasks while relaxing the restriction placed upon critical section execution times by hierarchical scheduled systems.

In order to illustrate the RACPwP, consider the scenario presented previously and depicted in Figure 5. At time *t = 2*, $Task_2$ requests and is granted access to the critical section. At time *t = 3* the hard real-time task $Task_1$ preempts $Task_2$ and executes. At time *t = 4*, $Task_1$ requests the shared resource locked by $Task_2$ and is blocked. At time *t = 5*, the budget for $Task_2$ has expired. $Task_2$ is pre-empted and $Task_1$ is allowed to lock the resource and continue. Finally at time *t = 11*, $Task_1$ completes and $Task_2$ is allowed to once again lock the resource and enter its critical section.

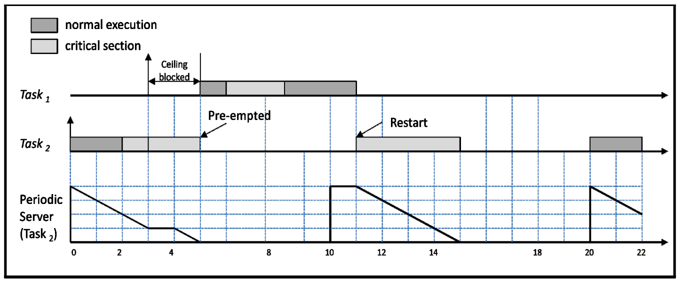

Figure 6: Example Resource Access Control Protocol with Preemption

Figure 7 provides an example function that simulates a bank balance transfer function implemented using traditional locking mechanisms (i.e. semaphores). Figure 8 provides the same function implemented using a preemptable critical section.

```
int transfer ( ) {
1.  sem_wait ( );
2.  value = source.balance
3.  value = value – amount
4.  source.balance = value;
5.  value = destination.balance
6.   value = value + amount
7.  destination.balance = value
8.  sem_give( )
}
```

Figure 7: Traditional locking mechanism (semaphores)

```
int transfer ( ) {
1.  PCS_START ( );
2.  value = PCS_LOAD(source.balance)
3.  value = value – amount
4.  PCS_STORE(source.balance)
5.  value = PCS_LOAD(destination.balance)
6.  value = value + amount
7.  PCS_STORE(destination.balance)
8.  PCS_COMMIT
}
```

Figure 8: Preemptable Critical Section

In Figure 7 the semaphore is acquired at line 1 and for the remaining 6 instructions cannot be preempted. Regardless of the priority the task cannot be preempted until the semaphore is released in line 8. In Figure 8 the code snippet illustrates the PCS mechanism which is implemented as a collection of macros. The *PCS_START* macro at line 1 performs the initial checkpoint required if the function is preempted and requires restarting. The other macros *PCS_STORE* and *PCS_LOAD* perform the memory access transactions and the *PCS_COMMIT* macro at line 8 commits the transactions to memory. The benefit of these macros is unlike traditional synchronization mechanisms the transfer function can be preempted at any point of its execution (except during an atomic PCS operation). Therefore if a task has exceeded its budget but still inside the critical section (i.e. a PCS_COMMIT has not been performed) then RACPwP can safely preempt the task. The task is then blocked until the next budget replenishment and allowed to restart again.

It is important to note that there are some limitations associated with using PCS. One is the added computational overhead that software transactional memory imposes of the system. The other limitation is traditional I/O should not be executed within a transaction.

V. Performance Analysis

This section provides the performance analysis of RACPwP as part of a hierarchical scheduled system. The performance is evaluated using worst-case response time analysis. Using the method provided by authors in [17] the worst-case response time of a task T_i served by a subsystem S_i occurs during one of the following scenarios:

- The subsystem's budget is exhausted as soon as the lower priority tasks begin to run and if the task is inside a critical section it is preempted.
- The task T_i and all other higher priority tasks in the application arrive right after the subsystem's budget is exhausted.
- The subsystem's budget is replenished but the execution is delayed for as long as possible due to the interference from other higher priority subsystems.

Based upon the scenarios provided above the worst-case response time of a task can be computed by identifying the interval of time where tasks at priority level *i* or higher can execute. This interval of time or execution window *w* is determined by three components:

1. The execution of task T_i along with all higher priority tasks at the *i*th priority level.
2. The replenishment periods of any complete servers.
3. Interference from higher priority servers (tasks running in higher priority subsystems).

Therefore, the worst-case response time of a task T_{si} can be calculated using the equation:

$$wcrt = w_{si}^{n} + J_s \tag{3}$$

where w_{si}^{n} can be determined by a recurrence function and J_s is the release jitter of task T_{si}.

$$w_{si}^{n+1} = L(w_{si}^{n}) + G(w_{si}^{n}) + I(w_{si}^{n}) \tag{4}$$

The worst-case response time analysis was extended by authors in [4] to include resource sharing across subsystems. The load $L(w_s^n)$ at the *i*th priority level is expanded by one term to including the affects of local and global blocking factors and is defined as:

$$L(w_{si}^{n}) = B_{si} + C_{si} + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_s^n + J_i}{T_i} \right\rceil C_{sj} \tag{5}$$

where B_{si} is the blocking factor due to local and global resource access. The replenishment period $G(w_{si}^{n})$ extended to include global blocking factors is defined as:

$$G(w_{si}^{n}) = \left(\left\lceil \frac{L(w_s^n)}{C_s} \right\rceil - 1 \right) (T_s - C_s) + B_s \tag{6}$$

where B_s represents the longest time that a server S_i could be blocked from executing by a lower priority server. The interference $I(w_{si}^{n})$ from any higher priority servers is defined as:

$$I(w_{si}^{n}) = \sum_{\forall X \in hp(S)} \left\lceil \frac{\max\left(0, w_{si}^{n} - \left(\left\lceil \frac{L(w_{si}^{n})}{C_S} \right\rceil - 1 \right) T_S \right)}{T_X} \right\rceil (C_x + B_{xo}) \tag{7}$$

where B_{so} represents the server overrun time which is the longest time a server S_i may execute. Additionally equation (10) represents a server overrun with no payback.. In order to analyze budget overrun with payback $I(w_{si}^{n})$ is defined as follows:

$$I(w_{si}^{n}) = \sum_{\forall X \in hp(S)} B_{xo} + \sum_{\forall x \le i} \left\lceil \frac{\max\left(0, w_{si}^{n} - \left(\left\lceil \frac{L(w_{si}^{n})}{C_S} \right\rceil - 1 \right) T_S \right)}{T_x} \right\rceil C_x \tag{8}$$

(Note that for RACPwP the server overrun term B_s in equation (6) is zero since the task would be preempted if the server budget expired while holding a lock on a global resource).

A. Results

In the following subsection, we provide a simple example for evaluation purposes between RACPwP and other protocols that use budget overrun mechanisms. For our example, we compare the server and task worst-case response times for RACPwP, OPEN-HSRP with budget payback and OPEN-HSRP without budget payback. The server response time for OPEN-HSRP with budget payback is calculated based upon the following:

$$w_s^{n+1} = C_s + B_s + \sum_{\forall X \in hp(S)} B_{xo} + \sum_{\forall X \in hp(S)} \left\lceil \frac{w_{si}^{n}}{T_x} \right\rceil C_x \tag{9}$$

The server response time for OPEN-HSRP without budget payback is calculated based upon the following:

$$w_s^{n+1} = C_s + B_{so} + \sum_{\forall X \in hp(S)} \left\lceil \frac{w_s^{n}}{T_x} \right\rceil (C_x + B_{xo}) \tag{10}$$

The recurrence functions for equations (9) and (10) begin with $w_s^0 = 0$ and terminate when $w_s^{n+1} = w_s^n$ which is the worst-case response time of the server. If $w_s^{n+1} > T_s$ then the server is not schedulable and therefore not considered. The simulated systems is composed of three separate subsystems each scheduled by the global preemptive periodic server. A global resource is shared between each subsystem S_i and the critical section execution time is represented as *CSET*$_i$. The J_i term represents the subsystem server jitter.

In order to evaluate the worst case server response times we used 100 simulation runs that used a uniform random number generator to vary the server capacity, server budget and the critical section execution times. The server parameters ranges that were used to generate the server parameters are provided in the Table 1.

Table 1: Server parameters

S_i	C_i	T_i	J_i	$CSET_i$
S_1	[50,500]	[200, 2000]	[150, 1500]	[35,200]
S_2	[1250,2500]	[5000,10000]	[3750,7500]	[35,200]
S_3	[3000,5000]	[12000,20000]	[9000,15000]	[35,200]

Table 2 provides the worst-case response times for RACPwP, OPEN-HSRP without budget payback (HSRPnP) and OPEN-HSRP with budget payback (HSRPwP).

Table 2: Server average worst-case response times

Si	*RACPwP*	*HSRPnP*	*HSRPwP*
S_1	390	390	390
S_2	2360	3400	2900
S_3	5550	12150	10012

As shown by the schedulability analysis given in Table 2 server response times are improved with RACPwP since does not perform any budget overrun. Server response times are practical identical for subsystem S_1 since the highest priority server is not subject to overruns. However, notice that RACPwP does significantly improve server response times for subsystems S_2 and S_3 which is not affected by the overrun mechanism.

The next step is to evaluate the worst-case task response times of RACPwP as compared to OPEN-HSRP with and without budget payback. The recurrence function for worst-case task response times begins with $w_i^0 = 0$ and ends when $w_i^{n+1} = w_i^n$ where the worst-case response time is $w_i^n + J_i$.

The task is not schedulable if $w_i^{n+1} > D_i - J_i$ in which case it is not considered for analysis. Subsystem S_2 was chosen to execute the tasks as it is the mid-level priority subsystem. For this example, a global shared resource $CSET_i$ is shared among tasks as well as a local resource. The local resource critical section execution time is represented by $CSET_{si}$. Similar to the server parameters a random number generator was used to vary the task worst case execution time, the task period and deadline. The local resource execution was also varied. The task parameter ranges that were used to vary the task parameters are defined in Table 3.

Table 3: Task parameters

τ_i	*Ci*	*Ti*	*Di*	*CSETsi*
T_1	[1180,2375]	[12500,25000]	[12500,25000]	[150,350]
T_2	[3150,4500]	[35000,50000]	[35000,50000]	[150,350]

Table 4 represents the worst-case response times for all tasks in subsystem S_2. The subsystem S_2 was chosen since it's the mid-level priority subsystem and would best illustrate the effects of our protocol on the overall system. The task worst-case response times are calculated according to the recurrence function (4).

Table 4: Task average worst-case response times

τ_i	*RACPwP*	*HSRPnP*	*HSRPwP*
T_1	7665	8860	9388
T_2	28900	23800	26200

As shown in Table 4 it is evident that shared resource access can have a significant impact on the overall task load. The result being that lower priority tasks may get preempted and have to be restarted. The result is illustrated in the increased response times. The lower priority tasks pays the steepest price as it may suffer from multiple preemptions incurred by the higher priority tasks. Notice how RACPwP provides improved response times over HSRPnP and HSRPwP for higher priority tasks but lower priority tasks may experience degraded response times. This is the inherent tradeoff with RACPwP. Higher priority tasks which are generally hard real-time benefit from the lower response times, which provides improved determinism. However lower priority tasks which are typically soft real-time are more tolerant of the increased response times.

VI. PRACTICAL APPLICATION

For the purpose of evaluation we implemented RACPwP as part of a hard real-time embedded application. We used a ground-based satellite command and control embedded system as our use case. A hardware-in-the-loop simulation was used to provide the workload generator for our system. This particular use case was chosen because the satellite telemetry can vary significantly. Bit rates can range from 4 Kbps to 1 Mbps and the amount of processing required to process a telemetry frame can vary significantly depending upon how densely a telemetry frame is populated.

The hierarchical scheduler was implemented as part of WindRiver's VxWorks 6.5 real-time operation system. The hardware for the embedded system consisted of a PowerPC MPC7455/MPC7457 single board computer. A separate special-purpose telemetry processor which is used for bit synchronization and decommutation of the pulse code modulated telemetry stream. The serial communication is provided by a FPGA-based RS422 PMC module. All devices were connected via a VME32 backplane. The PCS construct was implemented using TinySTM[18] which is a lightweight and portable software transactional memory library. The TinySTM library was ported to VxWorks and included as part of the kernel.

The three main software components of the system include a hard real-time periodic task that performs the bit synchronization, frame decommutation and frame processing of a telemetry stream. The second component is a soft real-time task that provides health and status monitoring for the vehicle. The third component is an *aperiodic* task that transmits serial uplink commands to the vehicle. The hard real-time telemetry processing task and the soft real-time monitoring task share a global resource which is the decomutated telemetry buffer.

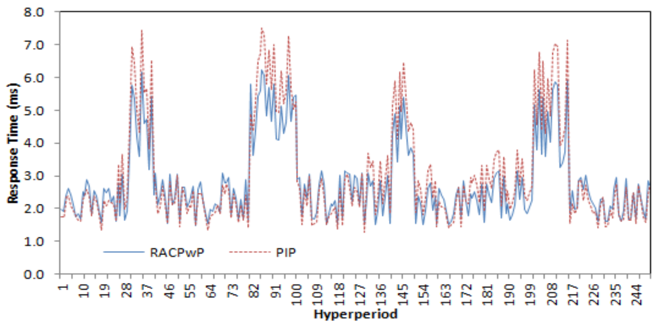

Figure 9: Hard real-time task response times

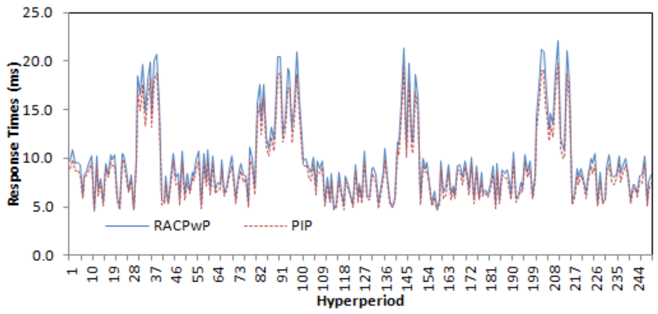

Figure 10: Soft real-time task response times

The data illustrated in Figures 9 and 10 represents the recorded task response times for the telemetry processing and health/status monitoring tasks. The response times were recorded at the completion of each periodic task.

As the figures confirm RACPwP provides improved task response times for hard real-time tasks and comparable response times for soft real-time tasks. Notice that for the during the periods of increased processing in Figure 9 with the hard real-time task using RACPwP outperformed PIP in terms of response times. (Note: the PIP protocol was chosen as a comparison because priority inheritance is the traditional resource allocation protocol used in real-time systems). This reduced response time leads to better determinism for hard real-time tasks because some of the issues like unbounded blocking that plague PIP are eliminated with RACPwP. Also notice in Figure 10 that on occasion RACPwP recorded comparable or even slightly better response times than PIP. This indicates that acceptable soft real-time response times do not have to be sacrificed at the expense of hard real-time determinism.

VII. Summary and Future Work

In this paper we considered the problem of sharing global resources in a hierarchical scheduled system. Traditionally, HSF was designed for soft real-time applications, in part due to problem of unbounded resource holding times between global resources. Our approach which combined software transactional memory provided better response times, than other state-of-the-art synchronization protocols, for higher priority tasks without drastically sacrificing soft real-time performance. Our motivation for this work stems for the aerospace industry where systems are routinely over engineered in the interest of real-time determinism. It is a common perception that an embedded system is considered "safe" at only 50% total utilization and considered unsafe above 75%. We propose that we can build more efficient embedded systems by more effectively managing the tasks within that system and in doing so reducing the total number of processing elements required. Some of the future work could include a more robust synthetic workload generator where large subsystem and task parameters could be quickly analyzed for schedulability and response times. The other main area would be to examine the possibilities of PCS to include other traditional non-preemptable resources such as standard I/O.

Acknowledgment

This work was supported in part by the National Science Foundation under NSF grant number 1136146.

References

[1] ARTIST Advanced Real-Time Systems, "Selected topics in Embedded Systems Design: Roadmaps for Research," Project IST-2001-34820 (2004).

[2] G. C. Buttazzo, Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications. Springer, Real-Time System Series, 2011.

[3] J.W. S. Liu, Real-Time Systems, Prentice-Hall, USA, 2000.

[4] R.I. Davis and A. Burns, "Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems," in Proc. of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)

[5] C.W. Mercer, S. Savage, H. Tokuda, "Temporal protection in real-time operating systems," in proceedings of the 11th IEEE workshop on Real-Time Operating System and Software, 1997, pp. 79-83.

[6] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in an open environment," in Proc. of IEEE Real-Time Systems Symp, 1997, pp. 308−319.

[7] G. Lipari and S.K. Baraugh, "Efficient scheduling of real-time multi-task applications in dynamic systems," in Proc. 6th IEEE Real-Time Technol. Appl. Symp. (RTAS'00), pp166-175.

[8] L. Sha, R. Rajkumar and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE trans. Comput. Vol 39, 1990, pp. 1175-1185.

[9] T.P. Baker, "Slack-Based Scheduling of Real-Time Processes," Real-Time Systems, vol. 3, 1991, pp. 67-99.

[10] G. C. Buttazzo and J. Stankovic, "Adding robustness in dynamic preemptive scheduling. Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems," 1995.

[11] M. Behnam, I. Shin, T. Nolte and M. Nolin, "Scheduling of semi-independent real-time compontnes: Overrun methods and resource holding times," (ETFA 2008).

[12] M. Behnam, T. Nolte, M Sjodin and I Shin, "Overrun Methods and Resource Holding Times for Semi-Independent Real-Time Systems," IEEE trans. on Indus. Informatics. 2010.

[13] T-W. Kuo, C-H. Li, "A Fixed Priority Driven Open Environment for Real-Time Applications," in Proc. of IEEE Real-Time Systems Symposium, 1999, pp. 256-267.

[14] M. Behnam, T. Nolte, M Sjodin and I Shin, "SIRAP: A synchronization protocol for hierarchical resource sharing real-time open systems," in Proc. 7th ACM and IEEE Int. Conf. Embedded Software (EM-SOFT 07).

[15] N. Fisher, M. Bertogna and S. Baraugh, "The Design of an EDF-Scheduled Resource-Sharing Open Environment," in RTSS '07.

[16] M. Asberg, T. Nolte and M. Behnam 2013, "Resource Sharing Using the Rollback Mechanism in Hierarchically Scheduled Real-Time Open Systems,". in RTSA '13.

[17] R.I. Davis and A. Burns, "Hierarchical Fixed Priority Pre-emptive Scheduling," Dept. Comp. Sci.Univ of York, 05.

[18] http://www.tmware.org.