Support for Legacy Real-Time Applications in an HSF-Enabled FreeRTOS

Rafia Inam, Moris Behnam, Mikael Sjödin
Mälardalen Real-Time Research Centre,
Mälardalen University, Västerås, Sweden
Email: {rafia.inam, moris.behnam, mikael.sjodin}@mdh.se

November 12, 2014

Abstract

This paper presents a runtime support to consolidate legacy together with other real-time applications, running a single instance of a real-time operating system (RTOS), and sharing system resources. In this context, we resort to the hierarchical scheduling framework (HSF) to provide temporal partitions for different applications, supporting their independent development and real-time analysis, thus resulting on a predictable integration. In particular, the paper focuses on a constructive element, the legacy server that allows executing code that is unaware of the temporal partition within which it is deployed. Furthermore, we discuss the challenges that need to be addressed to execute a legacy application in an HSF without modifications to the original code. We focus on the challenge of enabling sharing system resources, both hardware and software, as typically found in most industrial software systems. We propose a novel solution based on wrappers for the required RTOS system calls.

We implement our ideas in a concrete implementation on FreeRTOS OS, taking advantage of a prior HSF implementation. The validation is performed by a proof-of-concept case study that shows a successful integration of a legacy application that uses shared resources in a system that executes other applications.

1 Introduction

The trend of software reuse is observed in many industrial embedded software applications, e.g. automotive, consumer electronics and avionics. The reuse of legacy applications is an answer to many industrial challenges like development cost, time to market, and increasing complexity. For instance, the new Boeing 787 "Dreamliner" is a recent example with a significant proportion of reused modules from another Boeing airplane (Adams, 2005a,b). Furthermore, many industrial systems are developed in an evolutionary fashion, reusing applications from previous versions or from related products. It means that applications are reused and re-integrated in new environments.

Integration of real-time applications[1] can be explained as the mechanical task of wiring applications together (Crnkovic and Larsson, 2002). For real-time embedded systems, integrating legacy real-time application with new applications must achieve both (1) functional correctness and (2) satisfy extra-functional timing properties. Temporal behaviour of real-time software applications poses difficulties in their integration. Upon their integration, tasks of one application affect the scheduling of tasks of other applications. This means that for an embedded system with real-time constraints; an application that is found correct during unit testing may fail due to a change in temporal behaviour when integrated in a system.

Virtualization is a resource-management technique to solve these problems by partitioning the resources in a way that provides the illusion of a full resource but with a fractional capacity (Gu and Zhao, 2012). Using virtualization, a CPU resource is partitioned in several smaller virtual machines (VMs), each running a separate operating system instance either without any modification, e.g. KVM-based solutions (Cucinotta et al., 2011), or with modifications, e.g. Xen (Dragovic et al., 2003). However, executing multiple operating systems is frequently unfeasible and the performance overhead introduced by virtualization/hypervisor layer is a big challenge for the resource constrained embedded hardware nodes particularly 8-, 16- or 32-bit microcontrollers. Further, system administration of virtualization can become a time-consuming task due to complex configuration interactions between supposedly disjoint applications. A comparatively lightweight technique to allow the partitioning of OS environment into multiple domains and to execute a separate real-time application in each domain is *OS virtualization.* It has only one copy of OS kernel at runtime shared among multiple applications, thus better suited for the resource constrained embedded hardware. It is typically implemented using *server-based scheduling*, also called *Hierarchical Scheduling Framework (HSF)* (Deng and Liu, 1997; Shin and Lee, 2003).

HSF offers an efficient mechanism (i) to provide predictable integration of applications by rendering temporal partitioning among applications (Nolte et al., 2009), (ii) to support independent development and analysis of real-time applications (Shin and Lee, 2003), and (iii) to provide analysis of integrated applications at the system level (Shin and Lee, 2003; Abeni et al., 2009). These advantages of HSF could provide an even better leverage for reusing real-time legacy applications. However, most of the existing research focus is on providing analysis tools and algorithms in order to enable predictable reusability of applications (Shin and Lee, 2003; Nolte et al., 2009). Similar approaches have been proposed targeting specifically legacy applications, even if the timing characteristics of the applications are not known in advance (Palopoli and Abeni, 2009). Moreover, predictable reuse of legacy applications with HSF requires additional runtime support which, to the best of our knowledge, has not been investigated previously.

We provide an implementation support to execute the legacy application in a server within a two-level HSF. Our method enforces the creation of a *legacy server*, the creation of legacy tasks and their allocation to the legacy server. Thus the legacy server encapsulates the legacy application and becomes

[1]A real-time application consists of a set of executable real-time tasks. Please note that we focus only on the timing properties of the real-time applications.

a container for a set of legacy tasks. The use of legacy server upholds the independent development of the legacy application from the rest of the system, encapsulates internal temporal properties of the legacy application, and ensures the predictable temporal behaviour of the system. To support resource sharing among tasks of the same server (called *local resource sharing*) and among tasks of different servers (called *global resource sharing*), we implement two resource sharing protocols: *Stack Resource Policy (SRP)* (Baker, 1991), and *Hierarchical Stack Resource Policy (HSRP)* (Davis and Burns, 2006; Behnam et al., 2010) respectively.

For implementation, we choose our existing two-level HSF implementation for the FreeRTOS operating system (Inam et al., 2011a). In this paper we extend the existing implementation with the following contributions:

- Identification of new challenges and requirements for both the implementation of and for the information required from, the legacy applications in order to include them (preferably without changes) in the hierarchical framework.

- A runtime support for reusing legacy real-time applications. This entails: (1) *an implementation of a legacy server.* (2) *developing new wrappers for the original OS APIs of operating system* to support software and hardware resource sharing among legacy and other applications. To the best of our knowledge, it is the first work to identify the challenges and provide implementation support to execute the legacy real-time application with other application in HSF.

  (3) *implementations of resource sharing protocols.* This implementation was initially presented in our preliminary work (Inam et al., 2011b) and is subsumed by this work.

- Experimental validation of the proposed solution and implementation. We *apply a case study to evaluate the implementation* in terms of correctness and runtime overhead. A legacy application that uses FreeRTOS resource sharing APIs is executed within the legacy server to check (1) the automatic creation of legacy tasks and their execution within the legacy server, and (2) the correctness of wrappers. We also perform tests for HSRP implementation and wrappers for a hardware resource, that is shared between a new task and a legacy task. And finally, we *test and measure the performance* for our implementations for synchronization protocols at both levels on an AVR-based 32-bit board EVK1100 (EVK1100). We also compare overheads of the wrapper APIs against the original FreeRTOS APIs.

Organisation: Challenges in executing the legacy application in an HSF are described in Section 2. Section 3 provides our system model. Section 4 gives a background on FreeRTOS and reviews the HSF implementation in FreeRTOS. Section 5 overviews the resource sharing techniques for HSF. Sections 6 and 7 present our implementation of resource sharing and legacy support respectively. Section 8 presents a case study of a legacy application, and Section 9 experimentally evaluates the behavior of legacy application and resource sharing and presents overhead measures. Section 10 describes related work. Section 11 concludes the paper and finally, lists of APIs and macros of the implementation are given in the Appendix.

2 Challenges {and implementation issues} in executing legacy applications in an HSF

Integrating a legacy application, originally developed for full CPU access, in a two-level hierarchical framework raises many challenges. Our goal is to make minimum changes in the legacy application, changing old APIs calls with the newly developed APIs for HSF is not feasible. The legacy application is already tried-and-tested, and has been already deployed and executed, thus is more reliable. Making changes in the code of legacy application is tedious, time consuming, and error prone.

The first challenge is to create a legacy server itself and execute legacy tasks inside. New APIs are required to create the legacy server, create legacy tasks and assign legacy tasks to the server.

The second challenge is to execute the legacy application without modifying it. The code of legacy application still calls the original APIs of the OS. However, to execute the legacy application in the hierarchical environment, specifically developed APIs for HSF should be called instead. For example, in FreeRTOS the `xTaskCreate` system call is used for task creation, but the `xServerTaskCreate` system call is used in HSF for tasks creation within a server.

A third challenge arises when the legacy application accesses a shared resource and uses synchronization primitives of the OS. Consider that a resource is shared among tasks of a legacy application using OS synchronization primitives. When the same code is executed in a legacy-server within a two-level HSF along with other servers, the resource which is shared among tasks of the same server, i.e. legacy-server, is considered as a local resource. It is important to create and retain the resource within the server, and tasks of other servers should not be allowed to access this resource. It is important to keep the temporal isolation among servers in HSF. In addition, for the legacy application, resources that might be shared with other applications are considered as global resources and the HSRP-based resource access APIs should be used in this case. This adaptation could also be done by changing the respective system calls invoked in the legacy application to a convenient resource sharing protocol. However, changing the original code is error-prone and time consuming. Moreover, changing the synchronization protocol would change the semantics of the legacy application (e.g. changing semantics of PIP to SRP) which is undesired.

To overcome these challenges and execute the legacy application in HSF without modification, we develop wrappers around the original OS APIs. A wrapper is a middleware that wraps the operating system APIs. Wrappers exhibit the same interface as of the original APIs, but extend these with some extra functionality to call the new system calls (Stevens et al., 2003). This allows invoking the new system calls from within the legacy application without changing the legacy code. The advantage of wrapping over conventional redevelopment is that it requires less effort and lower development cost while keeping the original code unchanged. Moreover, it retains the semantics of the original operating system code.

In addition, the choice of synchronization protocol to be used in HSF depends on whether the legacy application is sharing resource with other applications or not. If it is not, the original system calls should be used i.e., the legacy application is exclusively using the processor resource. Otherwise, global synchronization primitives should be used instead. To make the legacy application general, this choice should be delayed until the deployment phase (similar to the principle of opaque analysis presented in (van den Heuvel et al., 2011)).

3 System model

In this paper, we consider a two-level HSF using the periodic resource model (Shin and Lee, 2003) in which a system $\mathcal{S}$ consists of a set of independently developed and analyzed subsystems S_s[2]. The HSF can be viewed as a tree with one parent node and many leaf nodes as illustrated in Figure 1. The parent node is a *global scheduler* and leaf nodes are subsystems. Each subsystem S_s consists of its own internal set of tasks that are scheduled by a *local scheduler*, and is executed by a server. The global scheduler schedules the system and is responsible for dispatching the servers according to their resource reservations. The local scheduler of each subsystem then schedules its task set according to a server-internal scheduling policy. The system contains a set of *global shared resources*, shared among tasks of different subsystems, and each subsystem has a set of *local shared resources*, shared among tasks of the same subsystem. In the rest of this paper, we use the term subsystem and server interchangeably.

3.1 Subsystem model

Each subsystem S_s is specified by a timing interface $S_s = \langle P_s, Q_s, p_s, X_s \rangle$, where P_s is the period for that server ($P_s > 0$), Q_s is the capacity allocated periodically to the server ($0 < Q_s \leq P_s$), and X_s is the maximum execution-time that any subsystem-internal task may lock a global shared resource $0 < X_s \leq Q_s$. Each server S_s has a priority p_s. The idle server has lowest priority i.e. 0 in the system. At each instant during run-time, B_s represents a remaining budget, $B_s \leq Q_s$. During execution of a subsystem, B_s is decremented by one at every time unit until it depletes. When $B_s = 0$, the budget is depleted and S_s will be suspended until its next period when B_s is replenished with Q_s. It should be noted that X_s is used for schedulability analysis only and our HSRP-implementation does not depend on the availability of this attribute.

3.2 Task model

We consider a simple periodic task model represented by a set $\mathcal{T}$ of n number of tasks. Each task τ_i is represented as $\tau_i = \langle T_i, C_i, \rho_i, cs \rangle$, where T_i denotes the period of task τ_i with worst-case execution time C_i where $0 < C_i \leq T_i$, ρ_i as its priority, and cs is the set of critical section execution times of all resources that the task accesses. For simplicity, we do not consider the case of nested resource access in this paper. A task, τ_i has a higher priority than another task, τ_j, if $\rho_i > \rho_j$. There can be 256 different task priorities, from lowest priority 1 (only

[2]In HSF terminology an application is also called a subsystem.

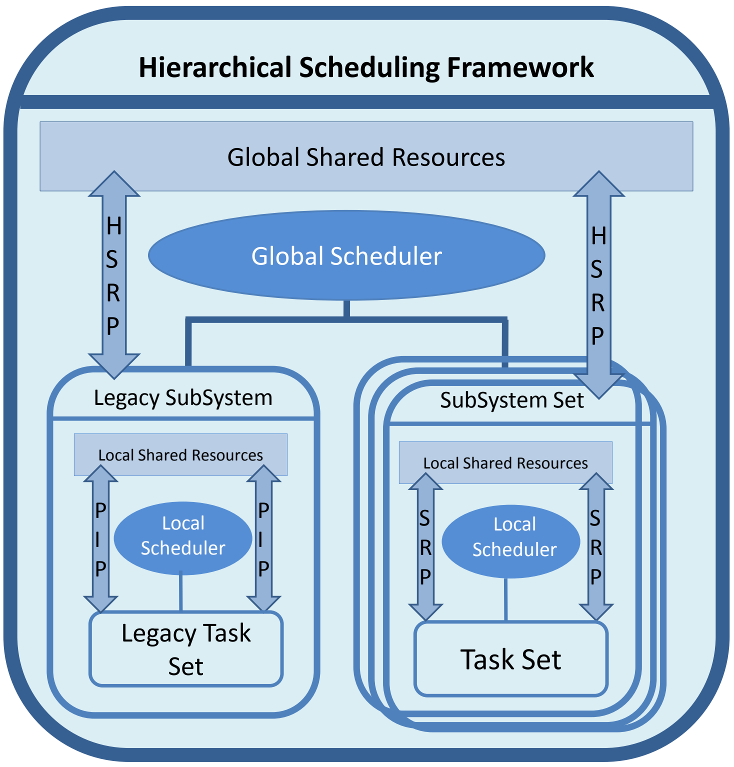


Figure 1: Two-level Hierarchical Scheduling Framework

idle task has priority 0) to the highest 255. For simplicity, the deadline for each task is equal to T_i.

3.3 Scheduling policy

We use fixed-priority preemptive scheduling (FPPS) at both global and local levels of scheduling. FPPS is flexible and is the de-facto industrial standard for task scheduling (OSEK Group). Our implementation supports shared priorities, which are then handled in FIFO order (both in global and local scheduling).

3.4 The legacy model

Each legacy application consists of a legacy task set, and a set of resources shared among those tasks. We assume that the legacy application is developed for FreeRTOS operating system and the source code is available. In HSF, the legacy application is now executed as a legacy server.

3.5 Summary of analytical framework

To perform real-time analysis techniques, a priori knowledge of tasks parameters and server parameters is required, which are generally not known for legacy applications. Given parameters of FreeRTOS tasks, a task's period and priority are derived for the legacy application. Given interfaces of tasks, the server interface can be derived using available technique (Palopoli and Abeni, 2009) that not only identifies the execution requirements of unknown applications, but can also be used to self-tune the scheduling parameters of legacy applications by using feedback scheduling. However, the resource sharing among tasks is not considered in that approach.

Given the system model, analytical frameworks exist to perform the schedulability analysis that can be used to integrate the newly developed applications for HSF and the legacy application together (Shin and Lee, 2003; Behnam et al., 2010; Palopoli and Abeni, 2009). Since analysis framework has been established, we complement it with practical implementation to allow it for practise. Note that the focus of this work is on integration and implementation; we leave the identification of blocking times for locked resources of legacy application as a future work.

4 FreeRTOS and its HSF implementation

This section presents the background on FreeRTOS and its synchronization primitives. Further it presents a brief overview of a HSF implementation in FreeRTOS. The HSF implementation is already presented in (Inam et al., 2011a) and is included here for the sake of completeness.

4.1 FreeRTOS and its synchronization primitives

FreeRTOS is a portable, open source (licensed under a modified GPL), mini real-time operating system developed by Real Time Engineers Ltd (Barry, 2010). It is ported to more than 20 hardware architectures ranging from 8-bit to 32-bit micro-controllers, and supports many development tools. Its main advantages are portability, scalability and simplicity. The core kernel is simple and small, consisting of three or four (depends on the usage of coroutines) C files, with a few assembler functions, resulting in a binary image between 4 to 9KB. Thus it is suitable for resource constraint micro-controllers. FreeRTOS kernel supports preemptive, cooperative, and hybrid scheduling. Using FPPS, tasks with the same priority are scheduled using the round-robin policy. It supports an arbitrary number of tasks, with both static and dynamic (changed at run-time) priorities, and 256 different priorities for tasks. Its scheduler runs at the rate of one tick per milli-second by default. It implements a very efficient task context-switch (i.e $10\mu s$ for the rate 1 milli-second).

FreeRTOS supports basic synchronization primitives like *binary, counting* and *recursive semaphore*, and *mutexes*. The mutexes employ priority inheritance protocol (PIP) (Sha et al., 1990), in which a lower priority task that is locking a shared resource inherits the priorities of all tasks that have higher priority and try to access the same resource. After returning the mutex, the task's priority is lowered back to its original priority. Priority inheritance mechanism minimizes the *priority inversion* but it cannot cure deadlock.

FreeRTOS implements all the above mentioned synchronization primitives using the message queues without buffering. The message queue structure `xQueue` is initiated at the creation of a semaphore or mutex and message queue APIs are called to handle synchronization among tasks. Each semaphore creates a separate queue to handle synchronization. `xSemaphorehandle` pointer points to a queue structure `xQueueHandle` created for that semaphore.

4.2 HSF implementation in FreeRTOS

A two-level HSF implementation (Inam et al., 2011a) on FreeRTOS supports idling periodic (Sha et al., 1986) and deferrable servers (Strosnider et al., 1995). *Idling periodic* means that tasks in the server execute and use the server's capacity until it is depleted. If the server has capacity but there is no task ready then it simply idles away its budget until a task becomes ready or the budget depletes. If a task arrives before the budget depletion, it will be served. *Deferrable server* means that tasks execute and use the servers capacity. If the server has capacity left but there is no task ready then it suspends its execution and preserves its remaining budget until its period ends. If a task arrives later before the end of servers period, it will be served and consumes servers capacity until the capacity depletes or the servers period ends. If the capacity is not used till the period end, then it is lost. In case there is no task (of any server) ready in the whole system, an idle server with an idle task will run instead.

To follow the periodic resource model (Shin and Lee, 2003), our servers and tasks are activated periodically. Servers behave like periodic tasks, they replenish their budget Q_s every constant period P_s. Since FPPS is used at both global and local scheduling levels, a higher priority server/task can preempt the execution of lower priority servers/tasks respectively. A brief overview of the implementation (Inam et al., 2011a) is given below:

4.2.1 Terminology

Terms used in the implementation are:
Active servers: Those servers whose remaining budget (B_s) is greater than zero. They are in the ready-server list.
Inactive servers: Those servers whose budget has been depleted and waiting for their next activation when their budget will be replenished. They are in the release-server list.
Ready-server list: It is a priority queue containing all active servers, and is arranged according to servers' priorities.
Release-server list: It is a priority queue containing all inactive servers, and is arranged according to servers' activation times. It is used to keep track of the replenishment of periodic servers.
Running server: The only server from the ready-server list that is currently running. At every system tick, its remaining budget is decreased by one time unit, until it exhausts.
Idle server: The lowest priority server that runs when no other server is active. In the deferrable server, it runs when there is no ready task in the system. This is useful for maintaining and testing the temporal separation among servers and also useful in testing system behavior. This information is useful in detecting over-reservations of server budgets and it can be used as feedback to resource management.
Ready-task list: Each server maintains a separate ready-task list to keep track of its ready tasks. Only one ready-task list will be active at any time in the system: the ready list of the running server.

Idle task: A lowest priority task existing in each server. It runs when its server has budget remaining but none of its task are ready to execute (in the idling server). In the deferrable server, the idle task of the idle server will run instead.

4.2.2 Data structures

The system maintains two lists: a ready-server list and a release-server list as mentioned earlier. The details of the data structures of these two lists can be found in (Inam et al., 2011a). The currently executing server in the system is pointed by a running-server pointer (see Figure 4). At any time instance, only the tasks of the currently running server execute.

Each server within the system contains the `subSystem control block` structure, as depicted in Figure 2. It maintains two lists: a ready-task list and a delayed-task list. The delayed-task list is the FreeRTOS list and is used to maintain the tasks when they are not ready (either suspended or delayed) and waiting for their activation.

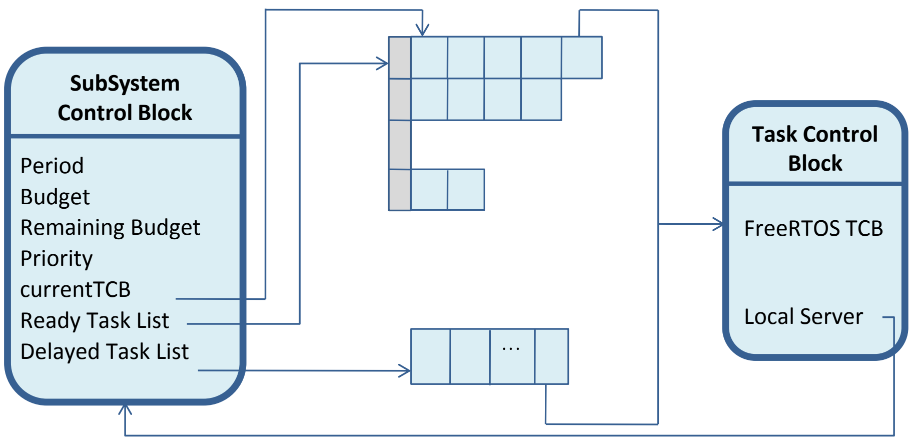


Figure 2: Data structures to implement HSF in FreeRTOS

4.2.3 Hierarchical scheduler

The hierarchical scheduling is started by calling `vTaskStartScheduler()` API and the tasks of the highest-priority ready server starts execution.
Tick-Interrupt handler: At each tick interrupt, the `interrupt_handler` routine performs the following functionality:

- The system tick is incremented.
- Check for the server's activation events. The newly activated servers' budgets are replenished to the maximum values and the servers are moved to the ready-server list.
- The global scheduler is called to handle the server events like execution, activation/replinishment, preemption of lower priority server, suspending the server at budget depletion, etc.

- The local scheduler is called to handle the task events like task execution, activation, preemption of lower priority task, suspension, etc.

Global scheduler: The functionality of the global scheduler is as follows:

- At each tick interrupt, the global scheduler decrements the remaining budget B_s of the running server by one and handles the budget expiration event (i.e. at the budget depletion, the server is moved from the ready-server list to the release-server list).
- Selects the highest priority ready server to execute and makes a server context-switch if required. Either `prvChooseNextIdlingServer()` or `prvChooseNextDeferrableServer()` is called to select idling or deferrable server, depending on the value of the `configGLOBAL_SERVER_MODE` macro in the `FreeRTOSConfig.h` file.
- `prvAdjustServerNextReadyTime(pxServer)` is called to set up the next activation time to activate the server periodically.

In idling periodic server, the `prvChooseNextIdlingServer()` function selects the first node (with highest priority) from the ready-server list and makes it the current running server. While in the case of a deferrable server, the `prvChooseNextDeferrableServer()` function checks the ready-server list for the next ready server that has any task ready to execute when the currently running server has no ready task even if it's budget is not exhausted. It also handles the situation when the server's remaining budget is greater than 0, but its period ends, in this case the server is replenished with its full capacity.

The server context-switch is very light-weight, and consists only of changing the running-server pointer from the currently executing server to the newly running server. The ready-task list of the newly running server is activated and all tasks of the list become ready for execution.

Local scheduler: The local scheduler is called from within the tick-interrupt handler routine using an adopted kernel function `vTaskSwitchContext()`. It is the original FreeRTOS scheduler with the following modification:

Instead of a single ready-task or delayed-task list (as in original FreeRTOS), now the local scheduler accesses a separate ready-task and delayed-task list for each server.

5 Resource sharing in HSF

We implement SRP (Baker, 1991) and HSRP (Davis and Burns, 2006; Behnam et al., 2010) to access local and global shared resources respectively. Since HSRP is an extension of SRP protocol, the SRP terms are extended to implement HSRP and some mechanisms must be implemented to prevent excessive blocking. To use SRP in a hierarchical setup, terms are extended as follows:

- *Preemption level (Priority):* According to SRP, each task τ_i has a static preemption level. Using FPPS, the task's priority ρ_i is used to indicate the preemption level. Similarly, for each subsystem S_s, its priority p_s is used as the preemption level.

- *Resource ceiling:* Each globally shared resource is associated with a *global ceiling* for global scheduling. This global ceiling is the highest priority of any subsystem whose task is accessing the global resource. Similarly each locally shared resource also has a *local ceiling* for local scheduling. This local ceiling is the highest priority of any task (within the subsystem) using the resource.

- *System and subsystem ceilings:* System and subsystem ceilings are dynamic parameters that change during runtime and the scheduler needs to be extended with the notion of these ceilings. The system ceiling is equal to the currently locked highest global resource ceiling in the system, while the subsystem ceiling is equal to the currently locked highest local resource ceiling in the subsystem.

Following the rules of SRP, a task τ_i can preempt the currently executing task within a subsystem only if τ_i has a priority higher than that of running task and, at the same time, the priority of τ_i is greater than the current subsystem ceiling.

Following the rules of HSRP, a task τ_i of a subsystem S_i can preempt the currently executing task of another subsystem S_j only if S_i has a priority higher than that of S_j and, at the same time, the priority of S_i is greater than the current system ceiling. Moreover, whilst a task τ_i of the subsystem S_i is accessing a global resource, no other task of the same subsystem can preempt τ_i.

The local and global schedulers are updated with the SRP and HSRP rules respectively and the details are described in Section 6.

Now we explain two overrun mechanisms used by HSRP to handle budget expiry during a critical section in the HSF. Consider a global scheduler that schedules subsystems according to their periodic interfaces. The subsystem budget is said to be expired at the point when one or more internal tasks have executed a total of Q_s time units within the subsystem period P_s. Once the budget is expired, no new task within the same subsystem can initiate its execution until the subsystem's budget is replenished at the start of the next subsystem period.

To prevent excessive priority inversion due to global resource lock, it is desirable to prevent subsystem rescheduling during critical sections of global resources. In this paper, we employ the overrun strategy to prevent such rescheduling. According to the overrun concept, upon the budget expiration of a subsystem while its task τ_i has still locked a global resource, the task τ_i is allowed to continue (overrun) its execution until either it releases the locked resource or its overrun time becomes equal to its subsystem budget. The extra time needed to execute after the budget expiration is denoted as *overrun time* θ. We implement two different overrun mechanisms (Behnam et al., 2010):

1. A basic overrun mechanism without payback denoted as BO: here no further actions will be taken after the event of an overrun.

2. The overrun mechanism with payback, denoted as PO: when an overrun happens, the subsystem S_s pays back this consumed amount of overrun in its next execution instant, i.e., the subsystem's budget Q_s will be decreased by θ_s i.e. $(Q_s - \theta_s)$ for the subsystem's execution instant following the overrun (note that only the instant following the overrun is affected since $\theta_s \leq Q_s$).

6 Support for resource sharing in HSF

Here we describe the design and implementation details of the resource sharing in two-level HSF. SRP and HSRP are implemented for local and global resource sharing respectively along with overrun mechanisms. The macro `configGLOBAL_SRP` in the configuration file is used to activate the resource sharing. The type of overrun can be selected by setting the macro `configOVERRUN_PROTOCOL_MODE` to either `OVERRUN_WITHOUT_PAYBACK` or `OVERRUN_PAYBACK`.

6.1 Support for SRP

For local resource sharing, we implement SRP to avoid problems like priority inversions and deadlocks.

The data structures for the local SRP: Each local resource is represented by the structure `localResource` that stores the resource ceiling and the task that currently holds the resource as shown in Figure 3. The locked resources are stacked onto the `localSRPList`; the FreeRTOS list structure is used to implement the SRP stack. The list is ordered according to the resource ceiling, and the first element of the list has the highest resource ceiling, and represents the `SubSystemCeiling`.

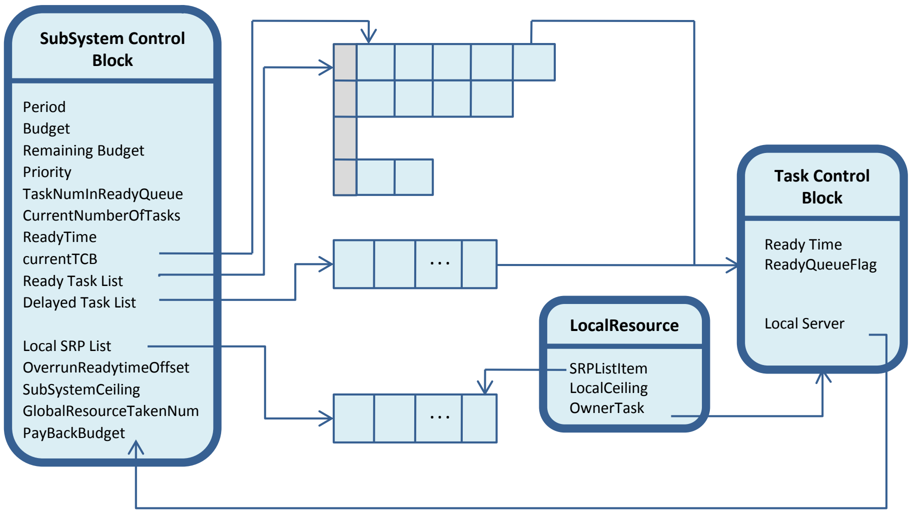


Figure 3: Data structures to implement SRP in HSF-enabled FreeRTOS

The extended functionality of the local scheduler for SRP: The only functionality we extended is the searching for the next ready task to execute. Now the scheduler selects a task to execute if the task has the highest priority among all the ready tasks and its priority is greater than the current `SubSystem-Ceiling`, otherwise the task that has locked the highest (top) resource in the `localSRPList` is selected to execute. The API list for the local SRP is provided in the Appendix.

6.2 Support for HSRP

HSRP is implemented to support global resource sharing among servers. The details are as follows:
The data structures for the global HSRP: Each global resource is represented by the structure `globalResource` that stores the global-resource ceiling and the server that currently holds the resource as shown in Figure 4. The locked resources are stacked onto the `globalHSRPList`; the FreeRTOS list structure is used to implement the HSRP stack. The list is ordered according to the resource ceiling, the first element of the list has the highest resource ceiling and represents the `SystemCeiling`.

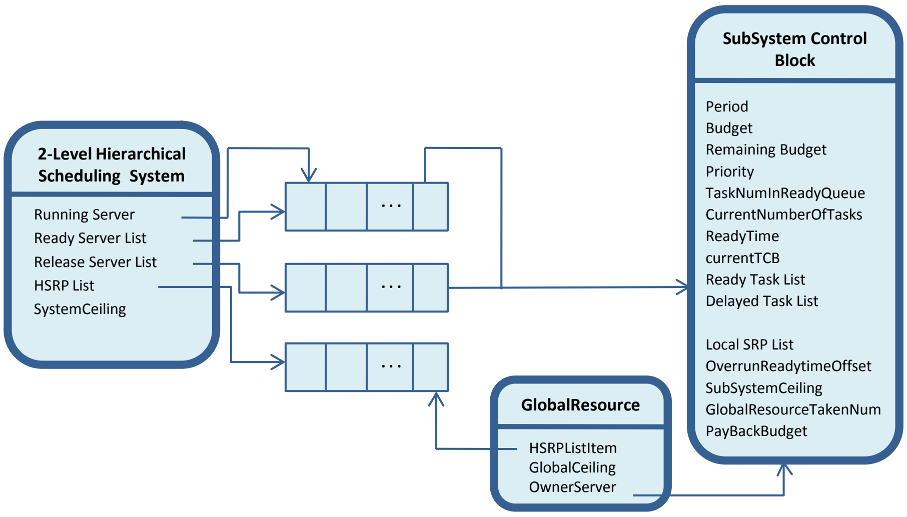


Figure 4: Data structures to implement HSRP

The extended functionality of the global scheduler for HSRP: To incorporate HSRP into the global scheduler, `prvChooseNextIdlingServer()` and `prvChooseNextDeferrableServer()` macros are appended with the following functionality: The global scheduler selects a server if the server has the highest priority among all the ready servers and the server's priority is greater than the current `SystemCeiling`, otherwise the server that has locked the highest (top) resource in the `HSRPList` is selected to execute. The API list for the global HSRP is provided in Appendix.

6.3 Managing local and/or global system ceilings

To ensure the correct access of shared resources at both local and global levels, the local and global ceilings should be updated properly upon the locking and unlocking of those resources. This functionality is implemented at both local and global levels within the SRP and HSRP APIs respectively, and is used to lock and unlock the local and global resources.

When a task locks a local/global resource whose ceiling is higher than the `SubSystem/System Ceiling`, the resource mutex is inserted as the first element onto the `localSRPList`/`HSRPList` respectively. Moreover, the `SubSystemCeiling`

/`SystemCeiling` is updated to the currently locked highest `LocalCeiling` /`GlobalCeiling` of the resource mutex respectively, and the task/server becomes the owner of the local/global resource accordingly. Each time a global resource is locked, the `GlobalResourceTakenNum` is also incremented.

Similarly upon unlocking a local/global resource, that resource is simply removed from the top of the `localSRPList`/`HSRPList` respectively. The `SubSystemCeiling` /`SystemCeiling` is updated accordingly, and the owner of this resource is set to NULL. For global resource, the `GlobalResourceTakenNum` is decremented.

6.4 Support for overrun protocols

To implement overrun mechanisms in order to prevent excessive priority inversion, the server should continue its execution even if its budget depletes while accessing a global shared resource; its currently executing task should not be preempted and the server should not be switched out by any other higher priority server (whose priority is not greater than the `SystemCeiling`) until the task releases the resource.

We have implemented two types of overrun mechanisms; (i) without payback (BO) and (ii) with payback (PO). The implementation of BO is very simple, the server simply executes and overruns its budget until it releases the shared resource, and no further action is required. For PO, we need to measure the overrun amount of time in order to pay it back at the server's next activation.

The data for overrun mechanisms: The `GlobalResourceTakenNum` is used as an overrun flag. As mentioned earlier, it is incremented and decremented at the global resource locking and unlocking respectively. When its value is greater than zero (means a task of the currently executing server has locked a global resource), no other higher priority server (whose `priority` is not greater than the `SubSystemCeiling`) can preempt this server, even if its budget depletes.

Two variables `PayBackBudget` and `OverrunReadytimeOffset` are added to the subsystem structure in order to keep a record of the overrun amount to be deducted from the next budget of the server as shown in Figure 4. The overrun time is measured and stored in `PayBackBudget`.

The extended functionality of the global scheduler for overrun: A new API `prvOverrunAdjustServerNextReadyTime(*pxServer)` is used to embed overrun functionality into the global scheduler. For PO, the amount of overrun, i.e. `PayBackBudget` θ_s is deducted from the server `RemainingBudget` B_s at the next activation period of the server, i.e. $B_s = Q_s - \theta_s$.

7 Support for legacy application and wrappers

Here we describe the design and implementation details of the legacy server and wrappers.

7.1 Creating the legacy server

To utilize the legacy support, a macro `configHIERARCHICAL_LEGACY` must be set in the configuration file `FreeRTOSConfig.h`. The user should rename the old `main()` function, and remove the `vTaskStartScheduler()` API from the legacy code.

The legacy application is executed in a separate legacy server. The user provides the server parameters like period, budget, and priority for the legacy server. The user also provides a function pointer to the legacy code (the old main function that has been renamed). The `xLegacyServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle, *pfLegacyFunc)` API is provided for this purpose, where `*pfLegacyFunc` is a function pointer pointing to the old main function of legacy application.

The legacy tasks are dynamically attached to the server. The `xLegacyServerCreate()` function first creates a server by calling `xServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle)` function. Second, it creates a highest priority private (hidden from the user) task called `vLegacyTask(*pfLegacyFunc)` within the legacy server using the `xServerTaskCreate( vLegacyTask, pcName, usStackDepth, (void *) pfLegacyFunc, configMAX_PRIORITIES - 1, NULL, *pxLegacyServerHandle)` API.

`vLegacyTask` function executes only once and its main functionality is 1) initializing legacy code (execution of the old main function which creates the initial set of tasks for the legacy application), 2) assigning legacy tasks to the legacy server, and 3) destroying itself; as presented in the Figure 5. When the legacy server is executed for the first time, all legacy tasks are created dynamically within the currently running legacy server and start execution.

```
// Legacy task function
// function called from xLegacyServerCreate()
static void vLegacyTask (void * pfLF)
{
   ((pfLegacyFunc)pfLF)();   //Initializes legacy code
   vTaskDelete(NULL);        //Destroys itself
}
```

Figure 5: Pseudo-code for legacy task implementation

We have adopted the original FreeRTOS `xTaskCreate` function and developed a wrapper to implement legacy support.

7.2 Wrapping FreeRTOS APIs

Our wrapper implementation consists of repackaging source code interfaces, hence there is no need for modifying the legacy code as depicted in Figure 6. The modified functionality is added as *Extended API*, while the original APIs are kept intact as *FreeRTOS APIs.* The wrapper provides links to both APIs, and depending on the configuration of `configHIERARCHICAL_LEGACY`, either the modified code or the original FreeRTOS code is executed. This process facilitates the execution of legacy application within the hierarchical environment without making any major modification in the code.

Wrapping FreeRTOS APIs is done in three steps: first, wrappers are constructed, secondly, the original APIs are adapted, and thirdly, the interaction between the wrapper and the legacy programs is tested. The wrappers descriptions are provided in this section, while their testing is performed in Section 9.

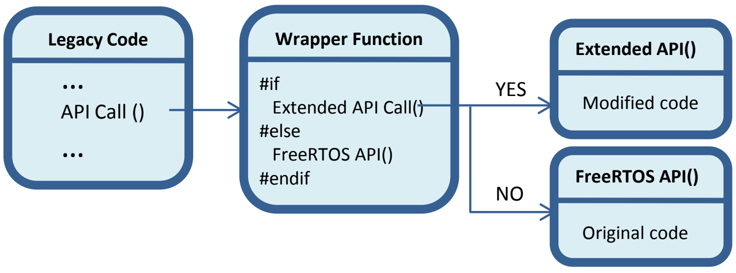


Figure 6: Wrappers implementation

7.2.1 Wrapper for the legacy task creation

A wrapper is provided for the `xTaskCreate` API, which redirects the task creation functionality to the `xServerTaskCreate` function by passing an additional parameter of legacy server handle `pxCurrentTCB->pxServer`. This is used to create legacy tasks within the currently executing legacy server, instead of executing the original code of `xTaskCreate` function as shown in Figure 7.

```
// Wrapper function for xTaskCreate() API
xTaskCreate (pxTaskCode, pcName, usStackDepth,
*pParameters, uxPriority, *xTaskHandle, *pStackBuffer,
xMemoryRegion, xRegions)
{
#if (configHIERARCHICAL_SCHEDULING == 1)
   #if (configHIERARCHICAL_LEGACY == 1)
      return xServerTaskCreate (pxTaskCode,
      pcName, usStackDepth, pvParameters, uxPriority,
      pxCreatedTask, pxCurrentTCB->pxServer, puxStack
      Buffer, xRegions);
   #endif
   return pdFALSE;
#endif
// original FreeRTOS code of xTaskGenericCreate
}
```

Figure 7: Pseudo-code of wrapper implementation for xTaskCreate API

7.2.2 Wrappers for resource sharing APIs

To handle synchronization among tasks of a legacy server in HSF and to meet the third challenge, we support the existing FreeRTOS resource sharing APIs with wrappers. We encapsulate each legacy shared resource within the legacy server by attaching an `owner` server to it, as shown by the newly designed structure `xLegacyQueue` in Figure 8. The pointer `pvQueue` points to the original FreeRTOS structure `xQueue`. The definition of semaphore handle `xQueueHandle` is modified to the new structure, as explained by the pseudo-code in Figure 9.

An example: To use the wrappers, no change is made in the code of legacy application. For example `xSemaphoreCreateMutex()` is used to create a mutex, which internally calls the `xQueueCreate` function. This function creates either

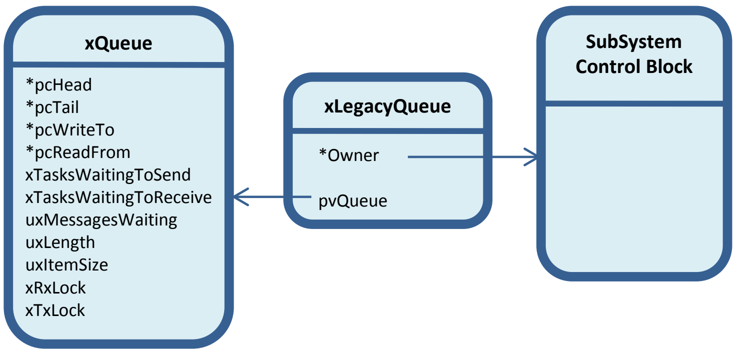


Figure 8: Data structures to implement wrappers for FreeRTOS resource sharing APIs

```
#if (configHIERARCHICAL_LEGACY == 1)
      typedef xLegacyQueue *xQueueHandle
#else
      typedef xQUEUE *xQueueHandle
#endif
```

Figure 9: Pseudo-code for the new definition of semaphore handle

`xLegacyQueue` or `xQueue` structure, and uses either wrapper code or the original FreeRTOS code depending on the configuration of `configHIERARCHICAL_LEGACY`.

7.2.3 Wrappers for Hardware drivers

A hardware device is accessed by using a hardware driver. The driver provides a software interface to hardware device and it is hardware dependent. A hardware device is usually considered as a global resource that can be shared among tasks of any application. The newly developed applications for HSF can take advantage of using our HSRP protocol implementation for global resource sharing, but the legacy application is not using the newly developed HSRP APIs. Sharing hardware resources among the legacy and the newly developed applications is a challenge. One simple method is to add the HSRP protocol in the hardware driver by developing wrappers for device drivers as shown in Figure 10.

The `Extended API Call()` in Figure 10 calls the modified code that uses the HSRP protocol within it. USART (universal synchronous/asynchronous receiver/transmitter) is a highly used driver in embedded systems to send and receive data to and from the embedded device. As an example we present our test results of using wrappers for USART in Section 9.2.4.

8 Case study: The legacy applications

In this section we present two legacy applications which are originally developed as stand-alone applications. For both applications, we use the task set presented in Table 1, except the resource sharing APIs which are different. The first legacy

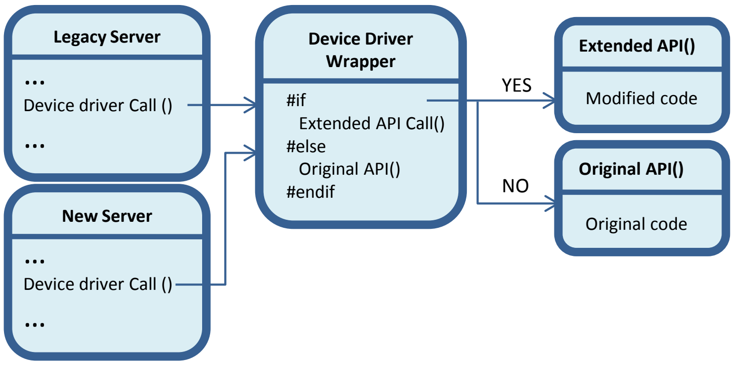


Figure 10: Wrappers implementation structure

application develops a system that endures a priority inversion problem where a high priority task is delayed by the execution of a lower priority task which is not sharing any resource. Second legacy application executes the same code but uses the PIP protocol that solves the priority inversion problem.

The purpose of selecting such a legacy application is to evaluate the wrappers that we have developed for the resource sharing APIs. These legacy applications are very suitable for such an evaluation because they use different resource sharing APIs (with and without PIP protocol). The intention is to preserve the behavior of both applications, when executed within a two-level hierarchical setup along with other applications.

8.1 The legacy applications' design

Each legacy application contains three tasks `TaskL`, `TaskM` and `TaskH` with priorities low, medium, and high respectively, as described in Table 1. Note that a higher number in the priority row in Table 1 means a higher priority for the task. A resource is shared between `TaskL` and `TaskH`. In `Execution Time` row of Table 1, cs represents the execution time of the task within the critical section and the other number shows task execution outside the critical section, e.g. $2cs + 1$ means first 2 time-units inside and then 1 time-unit outside the critical section, $(3 + 4cs$ means first 3 time-unit outside and then 4 time-units inside the critical section). The time unit is given in *system tick* which is equal to $1ms$ in our configuration.

`Tasks`	`TaskL`	`TaskM`	`TaskH`
`Priority`	1	2	3
`Period`	120	120	120
`Execution Time`	$2cs + 1$	6	$3 + 4cs$

Table 1: Legacy Tasks' properties.

8.2 The execution of legacy application in FreeRTOS

Both applications are executed as standalone applications on FreeRTOS using an EVK1100 board. Figure 11 provides the pseudo-code of three tasks for both applications. For the first application the shared resource is locked and unlocked using binary semaphore (i.e. for `lock resource R;` and `unlock resource R;` in Figure 11), while for the second application it is locked and unlocked using mutex.

```
// TaskH function body
// High  priority task sharing resource
while (1)  {
   execute for 1 tick;
   vTaskDelay(1);           //sleeps for 1 tick
   execute for 2 ticks;
   lock resource R;         //bin.  semaphore or mutex
        execute for 4 ticks;
   unlock resource R;       //bin.  semaphore or mutex
   vTaskWaitforNextPeriod(120);
}
```

```
// TaskM function body
// Medium priority task not sharing resource
while (1)  {
   vTaskDelay(1);
   execute for 6 ticks;
   vTaskWaitforNextPeriod(120);
}
```

```
// TaskL function body
// Low  priority task sharing resource
while (1)  {
   lock resource R;         //bin.  semaphore or mutex
        execute for 2 ticks;
   unlock resource R;       //bin.  semaphore or mutex
   execute for 1 tick;
   vTaskWaitforNextPeriod(120);
}
```

Figure 11: Pseudo-code of TaskH, TaskM, and TaskL used for both applications

Figure 12 shows the execution-traces of these tasks as two standalone applications. The left part of the figure demonstrates the execution of tasks using FreeRTOS binary semaphore APIs and suffering from priority inversion. At tick 6, `TaskH` requests for the shared resource and gets blocked since `TaskL` is accessing the resource. At this point of time, the medium priority `TaskM` executes, thus delaying the highest priority `TaskH` even it is not sharing any resource.

The right part of Figure 12 demonstrates the execution of a second Legacy application with the same code but using mutex instead of semaphores. It is obvious from Figure 12 (right part) that now the medium priority `TaskM` does not delay the execution of `TaskH`, thereby the priority inversion problem has been solved by using PIP protocol. Note that our goal is to demonstrate that

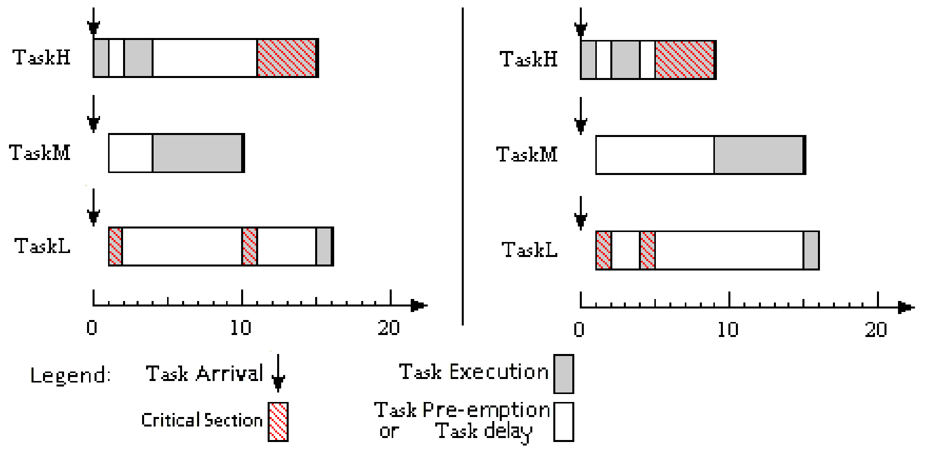


Figure 12: The behaviour of both legacy applications: using binary semaphores (left) and mutex (right)

the behaviour of legacy applications has been preserved when executed in a server within a hierarchical setup and is shown in the next section.

9 Experimental evaluation - Results and analysis

This section presents the evaluation of behavior and performance of our legacy server, wrappers, and resource sharing protocols'(SRP, HSRP) implementations. Overheads to execute the newly developed APIs and wrappers are also measured.

9.1 Experiment setup

All experiments are performed on an AVR-based 32-bit `EVK1100` board (EVK1100). The `AVR32UC3A0512` micro-controller runs at the frequency of `12MHz`. The HSF-enabled FreeRTOS is executed on the micro-controller using FPPS policy at both levels for idling periodic servers. The scheduler resolution (system tick) is set to `1ms` (milli seconds).

Four servers are created to perform the behaviour testing. A legacy server named as ***LegacyS*** is created to execute the legacy application. Two servers `S1` and `S2` are used in the system. Additionally, an `Idle` server is generated in the system with the lowest priority of all the other servers, i.e. 0, containing an idle task in it. All the other servers in the system have the priority higher than 0. Note that higher number means higher priority for both servers and tasks. The priorities, periods and budgets for these servers are given in Table 2.

Our implementation supports both idling periodic and deferrable servers, however, in this paper we present results with only idling periodic servers. An *idle task* per server is also generated automatically with the lowest priority. It runs when its server has budget remaining but none of its task is ready to execute.

Server	S1	S2	LegacyS
Priority	2	3	1
Period	40	20	60
Budget	15	5	10

Table 2: Servers used to test system behavior.

9.2 Behaviour testing

The purpose of these tests is to study:

1. the creation, behaviour, and correctness of the legacy server and the legacy tasks.
2. the correct behaviour of wrappers for FreeRTOS resource sharing APIs, i.e. semaphores, mutexes, etc.
3. the behaviour of global resource sharing using the HSRP protocol:
   (a) between two new servers.
   (b) between the legacy server and a new server.

Tasks	NT1	NT2	NT3	TaskL	TaskM	TaskH
Server	S1	S1	S2	LegacyS	LegacyS	LegacyS
Priority	1	2	1	1	2	3
Period	40	30	60	120	120	120
Execution Time	3	$3cs1 + 1$	$2 + 5cs1$	$2cs2 + 1$	6	$3 + 4cs2$

Table 3: Tasks properties and their assignment to servers.

Four experiments are performed to test different behaviours of the implementation. First, the legacy application using FreeRTOS semaphore APIs is realized within the legacy server and is exercised to test the functionality of the legacy server. Second, the legacy application using FreeRTOS mutex APIs is realized within the legacy server and the results are compared with the first experiment to validate the correctness of the wrappers for FreeRTOS resource sharing APIs. Third, the global resource sharing between newly developed applications is tested, and finally in the forth experiment, the global resource sharing between the legacy and the new applications is tested using the HSRP protocol.

All experiment are executed on the micro-controller and the execution traces are visualized using the Grasp tool (Holenderski et al., 2013). The experimental results are presented in the form of visualization of execution-traces in Figures 13, 14 and 15. In these traces, the horizontal axis represents the time in ms, starting from 0. In the server's visualization, numbers along the vertical axis are the server's capacity, the diagonal line represents server execution while the horizontal line represents either the waiting time for the next activation (when budget has depleted) or the waiting time for its turn to execute (when some other server is executing). Since these are idling periodic servers, all the servers in the system execute until their budget is depleted. If no task is ready then the idle task of that server executes till its budget is depleted.

9.2.1 Testing the execution of legacy application in a hierarchical setup

To test the execution of legacy application along with other servers in the system, the first legacy application that endures priority inversion is executed with the previously described servers in Table 2. Task properties and their assignments to the servers are given in Table 3.

Figure 13 visualizes the execution of legacy application within the legacy server along with other servers in the system. The `vLegacyTask` is created within `LegacyS`. It executes at the start of the server only once (at time 25 in Figure 13), and creates all other legacy tasks (i.e. `TaskL`, `TaskM`, `TaskH`, and an idle task), assigns them to `LegacyS` and destroys itself. From time 30, the legacy tasks start execution until the server depletes at time 35. The tasks start their execution again, when the server is replenished with its full budget at its next activation period.

Hence, by using the legacy server and a private `vLegacyTask`, the legacy tasks are automatically created and executed within `LegacyS` along with other servers in a hierarchical setup.

9.2.2 Testing wrappers for FreeRTOS APIs by executing legacy tasks within the legacy server

Here the main focus is to test the behaviour of newly developed wrappers for FreeRTOS APIs.

To perform this test, the first and second legacy applications using FreeRTOS semaphores and mutex APIs respectively are executed. Servers and tasks properties are provided in Table 2 and Table 3 respectively. The execution of both applications exhibit priority inversion problems with binary semaphore and its solution with mutex is visualized and presented in Figure 13 and Figure 14 respectively.

Tasks	NT1	NT2	NT3	LT1	LT2
Server	S1	S1	S2	LegacyS	LegacyS
Priority	1	2	1	1	2
Period	40	30	60	120	120
WCET	4	5	$2 + 5cs1$	$4cs1 + 4$	5

Table 4: Tasks properties and their assignment to servers.

`TaskL` and `TaskH` of both applications share a resource, which now becomes a local resource in the hierarchical setup as it is shared among tasks of the legacy server only. The wrappers are executed instead of the original semaphore and mutex APIs for resource sharing and the critical section is specified by $cs2$. From Figure 13 it is obvious that the legacy application suffers from priority inversion, as the TaskH's execution is delayed by the TaskM's execution which is not sharing any resource.

The solution of priority inversion using mutex APIs in `TaskL` and `TaskH` is demonstrated in Figure 14. Since the mutex implements PIP within them; therefore, the priority inversion problem is solved now. As obvious from the figure that `TaskM` executes after `TaskH`'s completion.

This test depicts that the FreeRTOS APIs are kept intact. It also manifests that the legacy application retains its original semantics while executing wrappers for these APIs in a hierarchical environment. Moreover, the legacy server does not overrun to prevent excessive blocking in both figures since it is not accessing a global shared resource and is not executing newly implemented HSRP protocol.

9.2.3 Testing the global resource sharing between new servers

In this section we test the behaviour of HSRP and overrun in the case of global resource sharing in the HSF implementation. We consider the same servers and tasks as used in the previous tests and which are provided in Tables 2 and 3 respectively. The trace of execution is visualized in Figure 13. Two tasks `NT2` and `NT3`, belonging to servers `S1` and `S2` respectively, are sharing a global resource. The overrun with the payback mechanism is assumed.

In Figure 13, `S2` depletes its budget at time 5, but continues to execute in its critical section until it unlocks the global resource at time 7, hence delaying the execution of `S1` by $2ms$. In case of an overrun with payback, the overrun time is deducted from the budget at the next server activation, as shown in Figure 13. At time 20 the server `S2` is replenished with a reduced budget, i.e 3. While in case of an overrun without payback, the server will be always replenished with its full budget.

9.2.4 Testing the modified hardware-driver APIs

The purpose of this experiment is to test the behaviour of HSRP in the case of global resource sharing between a legacy task and a new task. We are testing the modified hardware driver in which the resource is locked using HSRP. A legacy task `TaskL` of the legacy server shares a global hardware resource USART with a new task `NT3` of server `S2`. Three servers, as described in Table 2, are used in the system. Task properties and their assignments to the servers are given in Table 4. The critical section for resource sharing is specified as $cs1$ and the results are visualized in Figure 15. The visualization of the executions for budget overrun without payback (BO) and with payback (PO) for idling periodic server are presented in Figure 15(a) and Figure 15(b) respectively.

In case of budget overrun with payback, the overrun time is deducted from the budget at the next server activation, as shown in Figure 15(b). Since HSRP is used, the legacy server overruns at time 35, and later at time 60. The legacy server is replenished with a reduced budget, while in case of an overrun without payback the server is always replenished with its full budget as it is obvious from Figure 15(a). It is observed that a hardware resource (USART) is successfully shared among the legacy application and the newly developed HSF applications, without making any modification to the legacy code.

9.3 Performance measures

We present the overhead measurements for the wrappers used in legacy application and newly developed resource sharing APIs for shared resources. A second hardware timer-unit for the micro-controller is initiated and started to measure the performance. The APIs `StartTimer()` and `EndTimer()` are developed to measure execution time of different functions. For each data point, a total of 2000 values are measured. The minimum, maximum, and average of these values are calculated and presented for all results. All data points are given in micro-seconds (μs). The following overheads are measured:

1. The time required to run the wrappers in the hierarchical setup needed to be measured and compared against the original FreeRTOS APIs to calculate the overhead. The overhead measures for semaphore APIs are given in Table5.
2. Similarly, the overhead of executing the modified hardware driver for USART is measured and compared with the original driver. Additionally, we have also tested the effect of passing a different number of characters to the USART driver for printing. The overhead measures are given in Table6.
3. We also report the performance measures of lock and unlock functions for the newly developed APIs supporting SRP and HSRP protocols for shared both global and local resources. The execution time of functions to lock and unlock global and local resources is presented in Table 7.

Function	OS	Min.	Max.	Avg.
`xSemaphoreTake()`	wrapper	22	32	28.47
`xSemaphoreTake()`	FreeRTOS	21	32	26.32
`xSemaphoreGive()`	wrapper	21	32	26.05
`xSemaphoreGive()`	FreeRTOS	21	22	21.51

Table 5: The execution time (in micro-seconds μs) of for Semaphore.

Function	Description	Min.	Max.	Avg.
`usart_write_line()`	with global resource sharing	43	54	52.49
`usart_write_line()`	without resource sharing	0	11	9.94

Table 6: The execution time (in micro-seconds μs) of USART driver.

Function	Min.	Max.	Avg.
`vGlobalResourceLock`	21	21	21
`vGlobalResourceUnlock`	32	32	32
`vLocalResourceLock`	21	32	26.48
`vLocalResourceUnlock`	21	21	21

Table 7: The execution time (in micro-seconds μs) of newly developed global and local lock and unlock function.

The overheads for the semaphore wrappers are very low and negligible, i.e. approximately in average 2 μs for `xSemaphoreTake()` and 5 μs for `xSemaphoreGive()` as it is obvious from Table 5. For the hardware driver, we measured the time by passing no character to the USART to exactly measure the overhead as compared by calling the driver API. The overhead is approximately 42.55 μs. Additionally, the performance of the USART driver with a varying number of characters is also measured and the results reveal that the increase in the time to execute the code is linear with the increase in the number of characters.

For the server overheads we have performed evaluations in (Inam et al., 2011a) and the results reveal that the overhead measures are low.

10 Related work

10.1 Consolidating legacy applications

Different types of virtualization techniques are proposed to integrate and execute concurrently multiple applications (including the legacy applications) on a same hardware node using several virtual machines (VMs) and a hypervisor (Gu and Zhao, 2012). Examples are without modifying OS (Cucinotta et al., 2010, 2011; M.Åsberg et al., 2012), or with modifying OS, e.g. Xen-based solution (Cherkasova et al., 2007; Yu et al., 2010). We focus to execute applications on resource constraint small microcontroller (a 32-bit board), thus executing multiple operating systems is unfeasible and the performance overhead introduced by virtualization/hypervisor layer is a big challenge for such microcontrollers.

OS virtualization or HSF is more lightweight than other virtualizations because of having only a single copy of OS, thus better suited for resource constraint hardware. The hierarchical scheduling processor models guarantee that applications are developed and analyzed independently in isolation and are later integrated together by providing temporal isolation among applications (Almeida and Pedreiras, 2004; Feng and Mok, 2002; Davis and Burns, 2005; Shin and Lee, 2003; Abeni et al., 2009). These advantages make HSF suitable for integrating and executing legacy applications (developed to use the full CPU-access) with other applications (developed to execute in hierarchical setup). However, it requires a priori knowledge of legacy application's timing requirements which has been addressed for independent tasks by (Palopoli and Abeni, 2009).

A lot work has been done from the HSF implementation perspective (Saewong and Rajkumar, 2001; Buttazzo and Gai, 2006; Kim et al., 2000; Behnam et al., 2008; Holenderski et al., 2012) on Linux/RK, open source ERIKA Enterprise kernel, SPIRIT-μKernel, VxWorks, μC/OS-II respectively.Although the reuse of legacy application is proposed by the hierarchical scheduling theoretical work, all mentioned implementations have not proposed special support for facilitating the reuse of real-time legacy application which is the main focus of this paper. To the best of our knowledge, our work is the first to identify challenges and implementation issues and to support a practical implementation for legacy code execution within a server in HSF. No other HSF implementation has investigated on this issue before. Next, we present an overview of the existing synchronization protocols and their implementations in HSF.

10.2 Synchronization protocols

10.2.1 Resource sharing for simple single-level scheduling

Here we describe synchronization protocols used to share resources among tasks in a single-level scheduling systems. Priority inheritance protocol (PIP) (Sha et al., 1990) was developed to solve the priority inversion problem but it does not solve the chained blocking and deadlock problems. Sha *et al.* proposed the priority ceiling protocol (PCP) (Sha et al., 1990) to solve these problems. A slightly different alternative to PCP is the immediate inheritance protocol (IIP). In IIP, the locking task raises its priority to the ceiling priority of the resource, when it locks a resource as compared to the PCP where the locking task raises its priority when another task tries to lock the same resource. Baker presented the stack resource policy (SRP) (Baker, 1991) that supports dynamic priority scheduling policies. For fixed-priority scheduling, SRP has the same behavior as IIP. SRP reduces the number of context-switches and the resource holding time as compared to PCP. Like most real-time operating systems, FreeRTOS only support an FPPS scheduler with PIP protocol for resource sharing. We implement SRP for local-level resource sharing in HSF.

10.2.2 Resource sharing for two-level hierarchical scheduling

To perform independent analysis for applications integration, information about tasks accessing which global shared resources should be known. In a two-level hierarchical scheduling, the resource sharing of a global resource requires to consider the priority inversion at both levels of hierarchy, i.e. between applications at the global level and between tasks within the application at the local level. Multiple synchronization protocols based on SRP (Baker, 1991) have been proposed to accommodate such resource sharing. Fisher *et al.* proposed Bounded delay Resource Open Environment (BROE) protocol (Fisher et al., 2007; Bertogna et al., 2009) for global resource sharing under EDF scheduling. Hierarchical Stack Resource Policy (HSRP) (Davis and Burns, 2006) uses the overrun mechanism to deal with the subsystem budget expiration within the critical section and uses two mechanisms (with pay back and without payback) to deal with the overrun. Subsystem Integration and Resource Allocation Policy (SIRAP) (Behnam et al., 2007) uses the skipping mechanism to avoid the problem of application budget expiration within the critical section. While Rollback Resource Policy (RRP) (Åsberg et al., 2013) uses the rollback approach if the budget expires between the critical section. All HSRP, SIRAP, and RRP assume FPPS. The original HSRP (Davis and Burns, 2006) does not support the independent application development for its analysis. Behnam *et al.* (Behnam et al., 2010) extended the analysis for the independent development of applications. In this paper we use HSRP (Behnam et al., 2010) for global resource sharing and implement both forms of the overrun mechanism.

Asberg *et al.* (Åsberg et al., 2010) implemented overrun and skipping techniques at top of their FPPS HSF implementation for VxWorks and compared the two resource-sharing techniques. Van den Heuvel *et al.* extended the μC/OS-II HSF implementation with resource sharing support (van den Heuvel et al., 2012) by implementing SIRAP and HSRP (with and without payback). They measured and compared the system overheads of both primitives. More recently, Asberg *et al.* (Åsberg et al., 2013) implemented and evaluated RRP against HSRP (with and without payback) and SIRAP, and examined that RRP is better in average-case response-times than both protocols.

Unlike (Åsberg et al., 2010; van den Heuvel et al., 2012) and (Åsberg et al., 2013) which implement SIRAP, HSRP, and RRP and comparing protocols against each other, we implement HSRP only. We do not consider SIRAP because of its implementation complexity, i.e., worst case execution times of critical sections should be provided during runtime. In addition, we neither consider BROE due to its limitation in supporting FPPS. Our main focus is to enable the reusability of the legacy application and at keeping the semantics of the application intact rather than evaluating different synchronization protocols. To achieve our goals, we keep all FreeRTOS original APIs intact and call the ones that need to be changed through wrappers implementation.

We aim at efficiency in terms of processor overheads and simplicity in our design with the consideration of minimal modifications in underlying FreeRTOS kernel. Like (Holenderski et al., 2012; Behnam et al., 2008) our implementation limits the interference of inactive servers on system level by deferring the handling of their local events until those servers become active.

11 Conclusions and future work

This paper presented the integration and execution of legacy real-time application along with the newly developed real-time applications. The focus was to present a solution that pertains the semantics and real-time scheduling properties of old and new applications before and after their integration. We proposed to use the hierarchical scheduling approach (HSF) for this purpose and have demonstrated the suitability of HSF to execute legacy real-time applications in a predictable manner along with other applications. We have identified challenges to execute the legacy application in an HSF setup. Furthermore, we have also described the challenges and implementation issues of enabling resource sharing among the legacy and other applications to make the solution more applicable.

We have presented a runtime support for creating a legacy server and executing the real-time legacy tasks within the server. For resource sharing, we implemented SRP and HSRP protocols for local and global resource sharing respectively. Moreover, to achieve the challenge of resource sharing among legacy and other applications, we have presented the solution by implementing wrappers for the FreeRTOS APIs.

We have conducted a number of experiments in order to validate the correctness and the efficiency of the proposed solution. We have run our experiments on the EVK1100 board with a 32-bit AVR32UC3A0512 micro-controller. The collected results from the experiments show a smooth execution of legacy tasks integrated with other applications with minimum changes in the code of the legacy tasks. In addition, we could observe, from the experiments, the correct temporal behavior of applications that use our solution when they share software/hardware global/local resources. Finally, the results reveal that the runtime overheads of the proposed solution (including server, wrappers and APIs) are rather low. It is done without making any major modification to the legacy code. We have evaluated the implementation of the newly developed APIs for resource sharing (i.e. SRP and HSRP protocols) and for the wrappers. The results reveal that overhead of our implemented functionality is low.

In the future we plan to extend our solution for multicore architectures.

Acknowledgements

This work is supported by the Knowledge Foundation (KK-Stiftelsen), via the research programme PPMSched.

References

Abeni, L., Palopoli, L., Scordino, C., Lipari, G., 2009. Resource reservations for general purpose applications. IEEE Transactions on Industrial Informatics 5, 12–21.

Adams, C., 2005a. Product focus: Cots operating systems: Boarding the boeing 787. [Online]. Available: http://www.aviationtoday.com/, last checked: 12.05.2014.

Adams, C., 2005b. Reusable software components: Will they save time and money? [Online]. Available: http://www.aviationtoday.com/, last checked: 12.05.2014.

Almeida, L., Pedreiras, P., 2004. Scheduling within temporal partitions: response-time analysis and server design, in: 4^{th} ACM International Conference on Embedded Software(EMSOFT), pp. 95–103.

Åsberg, M., Behnam, M., Nolte, T., Bril, R.J., 2010. Implementation of overrun and skipping in VxWorks, in: 6^{th} Annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 1–8.

Åsberg, M., Nolte, T., Behnam, M., 2013. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems, in: Proc. 19^{th} IEEE Real-Time Technology and Applications Symposium (RTAS), pp. 129–140.

Baker, T., 1991. Stack-based scheduling of real-time processes. Journal of Real-Time Systems 3, 67–99.

Barry, R., 2010. Using the FreeRTOS Real Time Kernel. Real Time Engineers Ltd.

Behnam, M., Nolte, T., Shin, I., Åsberg, M., Bril, R.J., 2008. Towards hierarchical scheduling on top of VxWorks, in: 4^{th} Annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 63–72.

Behnam, M., Nolte, T., Sjödin, M., Shin, I., 2010. Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems. IEEE Transactions on Industrial Informatics 6, 93–104.

Behnam, M., Shin, I., Nolte, T., Nolin, M., 2007. SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems, in: 7^{th} ACM International Conference on Embedded Software(EMSOFT), pp. 279–288.

Bertogna, M., Fisher, N., Baruah, S., 2009. Resource-sharing servers for open environments. IEEE Transactions on Industrial Informatics 5, 202–219.

Buttazzo, G., Gai, P., 2006. Efficient EDF implementation for small embedded systems, in: 2nd Annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 1–10.

Cherkasova, L., Gupta, D., Vahdat, A., 2007. Comparison of the three CPU schedulers in Xen. ACM SIGMETRICS Performance Evaluation Review 35, 42–51.

Crnkovic, I., Larsson, M. (Eds.), 2002. Building Reliable Component-Based Software Systems. Artech House publisher. ISBN 1-58053-327-2.

Cucinotta, T., Checconi, F., Giani, D., 2011. Improving responsiveness for virtualized networking under intensive computing workloads, in: Proceedings of the 13th Real-Time Linux Workshop.

Cucinotta, T., Giani, D., Faggioli, D., Checconi, F., 2010. Providing performance guarantees to virtual machines using real-time scheduling, in: Proceedings of Euro-Par Workshops, pp. 657– 664.

Davis, R.I., Burns, A., 2005. Hierarchical fixed priority pre-emptive scheduling, in: Proc. 26th IEEE Real-Time Systems Symposium (RTSS), pp. 389–398.

Davis, R.I., Burns, A., 2006. Resource sharing in hierarchical fixed priority pre-emptive systems, in: Proc. 27th IEEE Real-Time Systems Symposium (RTSS), pp. 389–398.

Deng, Z., Liu, J.W.S., 1997. Scheduling real-time applications in an open environment, in: Proc. 18th IEEE Real-Time Systems Symposium (RTSS), pp. 308–319.

Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., Neugebauer, R., 2003. Xen and the art of virtualization, in: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP).

EVK1100, 2015. ATMEL EVK1100 product page. Website. [Online]. Available: http://www.atmel.com/tools/evk1100.aspx, last checked: 15.05.2014.

Feng, X., Mok, A., 2002. A model of hierarchical real-time virtual resources, in: Proc. 23th IEEE Real-Time Systems Symposium (RTSS), pp. 26–35.

Fisher, N., Bertogna, M., Baruah, S., 2007. Resource-locking durations in EDF-scheduled systems, in: Proc. 13th IEEE Real-Time Technology and Applications Symposium (RTAS), pp. 91–100.

Gu, Z., Zhao, Q., 2012. A state-of-the-art survey on real-time issues in embedded systems virtualization. Journal of Software Engineering and Applications (JSEA) 5, 277–290.

van den Heuvel, M.M.H.P., Behnam, M., Bril, R.J., Lukkien, J.J., Nolte, T., 2011. Opaque analysis for resource sharing in compositional real-time systems, in: 4th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS), pp. 1–8.

van den Heuvel, M.M.H.P., Bril, R.J., Lukkien, J.J., 2012. Transparent synchronization protocols for compositional real-time systems. IEEE Transactions on Industrial Informatics 8, 322–336.

Holenderski, M., Bril, R., Lukkien, J., 2013. Grasp: Visualizing the behavior of hierarchical multiprocessor real-time systems. Journal of Systems Architecture 59, 307–314.

Holenderski, M., Bril, R.J., Lukkien, J.J., 2012. Real-Time Systems, Architecture, Scheduling, and Application. InTech. chapter An Efficient Hierarchical Scheduling Framework for the Automotive Domain. ISBN 978-953-51-0510-7.

Inam, R., Mäki-Turja, J., Sjödin, M., Ashjaei, S.M.H., Afshar, S., 2011a. Support for hierarchical scheduling in FreeRTOS, in: Proc. 16^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–10.

Inam, R., Mäki-Turja, J., Sjödin, M., Behnam, M., 2011b. Hard real-time support for hierarchical scheduling in FreeRTOS, in: 7^{th} Annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 51–60.

Kim, D., Lee, Y.H., Younis, M., 2000. Spirit-ukernel for strongly partitione real-time systems, in: Proc. of the 7^{th} International conference on Real-Time Computing Systems and Applications (RTCSA), pp. 73–80.

M.Åsberg, Nolte, T., Kato, S., Rajkumar, R., 2012. Exsched: An external cpu scheduler framework for real-time systems, in: 18th IEEE International Conference (RTCSA' 12).

Nolte, T., Shin, I., Behnam, M., Sjödin, M., 2009. A synchronization protocol for temporal isolation of software components in vehicular systems. IEEE Transactions on Industrial Informatics 5, 375–387.

OSEK Group, . OSEK VDX operating system specification 2.2.3. [Online]. Available: http://www.osek-vdx.org, last checked: 15.05.2014.

Palopoli, L., Abeni, L., 2009. Legacy real-time applications in a reservation-based system. IEEE Transactions on Industrial Informatics 5, 220–228.

Saewong, S., Rajkumar, R., 2001. Hierarchical reservation support in resource kernels. [Online]. Available: http://www.cs.cmu.edu/afs/cs/project/rtml-2/Papers/hrsv.ps.gz, last checked: 15.05.2014.

Sha, L., Lehoczky, J., Rajkumar, R., 1986. Solutions for some practical problems in prioritised preemptive scheduling, in: Proc. 7^{th} IEEE Real-Time Systems Symposium (RTSS), pp. 181–191.

Sha, L., Rajkumar, R., Lehoczky, J.P., 1990. Priority Inheritance Protocols: an approach to real-time synchronization. Journal of IEEE Transactions on Computers 39, 1175–1185.

Shin, I., Lee, I., 2003. Periodic resource model for compositional real-time guarantees, in: Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS), pp. 2–13.

Stevens, R., Fenner, B., Rudoff, M., A., 2003. UNIX Network Programming. Addison-Wesley.

Strosnider, J., Lehoczky, J., Sha, L., 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. IEEE Transactions on Computers 44.

Yu, P.J., Xia, M.Y., Lin, Q., Zhu, M., Gao, S., Qi, Z.W., Chen, K., Guan, H.B., 2010. Real-time enhancement for Xen hypervisor, in: Proceedings of the 8th International Confer-ence on Embedded and Ubiquitous Computing of the (IEEE/IFIP), pp. 23–30.

Appendix

A synopsis of the application program interface of HSF implementation is presented below. The names of these API and macros are self-explanatory. The newly added user API and macro are the following:

1. `signed portBASE_TYPE xLegacyServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle, *pfLegacyFunc);`

The user API to implement the local SPR and the global HSPR are the following:

1. `xLocalResourcehandle xLocalResourceCreate(uxCeiling)`
2. `void vLocalResourceDestroy(xLocalResourcehandle)`
3. `void vLocalResourceLock(xLocalResourcehandle)`
4. `void vLocalResourceUnLock(xLocalResourcehandle)`
5. `xGlobalResourcehandle xGlobalResourceCreate (uxCeiling)`
6. `void vGlobalResourceDestroy(xGlobalResourcehandle)`
7. `void vGlobalResourceLock(xGlobalResourcehandle)`
8. `void vGlobalResourceUnLock(xGlobalResourcehandle)`

The new APIs to implement legacy server are the following:

1. `signed portBASE_TYPE xLegacyServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle, *pfLegacyFunc);`
2. `signed portBASE_TYPE xServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle);`
3. `static void vLegacyTask(*pfLegacyFunc);`
4. `#define xServerTaskCreate(vLegacyTask, pcName, usStackDepth, (void *) pfLegacyFunc, configMAX_PRIORITIES - 1, pxCreatedTask, *pxLegacyServerHandle) xServerTaskGenericCreate((vLegacyTask), (pcName), (usStackDepth), ((void *) pfLegacyFunc), (configMAX_PRIORITIES - 1), (pxCreatedTask), (*pxLegacyServerHandle), (NULL), (NULL))`

Adopted FreeRTOS APIs for wrappers

1. `xSemphoreCreateBinary`
2. `xSemaphoreTake`

3. `xSemaphoreGive`
4. `vSemaphoreCreateMutex`
5. `xSemCreateRecursiveMutex`
6. `xSemaphoreTakeRecursive`
7. `xSemaphoreGiveRecursive`
8. `xSemaphoreCreateCounting`
9. `xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength, unsigned portBASE_TYPE uxItemSize)`
10. `xQueueHandle xQueueCreateMutex(void)`
11. `portBASE_TYPE xQueueGiveMutexRecursive(xQueueHandle pxMutex)`
12. `portBASE_TYPE xQueueTakeMutexRecursive(xQueueHandle pxMutex, portTickType xBlockTime)`
13. `xQueueHandle xQueueCreateCountingSemaphore(unsigned portBASE_TYPE uxCountValue, unsigned portBASE_TYPE uxInitialCount)`
14. `signed portBASE_TYPE xQueueGenericSend(xQueueHandle pxQueue, const void * const pvItemToQueue, portTickType xTicksToWait, portBASE_TYPE xCopyPosition)`
15. `signed portBASE_TYPE xQueueGenericSendFromISR(xQueueHandle pxQueue, const void * const pvItemToQueue, signed portBASE_TYPE *pxHigherPriorityTaskWoken, portBASE_TYPE xCopyPosition)`
16. `signed portBASE_TYPE xQueueGenericReceive(xQueueHandle pxQueue, void * const pvBuffer, portTickType xTicksToWait, portBASE_TYPE xJustPeeking)`
17. `signed portBASE_TYPE xQueueReceiveFromISR(xQueueHandle pxQueue, void * const pvBuffer, signed portBASE_TYPE *pxTaskWoken)`

Adopted FreeRTOS Private function

1. `signed portBASE_TYPE uxQueueMessagesWaiting(const xQueueHandle pxQueue)`
2. `void vQueueDelete(xQueueHandle pxQueue)`
3. `static void prvUnlockQueue(xQueueHandle pxQueue)`
4. `static signed portBASE_TYPE prvIsQueueEmpty(const xQueueHandle pxQueue)`
5. `signed portBASE_TYPE xQueueIsQueueEmptyFromISR(const xQueueHandle pxQueue)`
6. `static signed portBASE_TYPE prvIsQueueFull(const xQueueHandle pxQueue)`
7. `signed portBASE_TYPE xQueueIsQueueFullFromISR(const xQueueHandle pxQueue)`

Adopted Hardware driver user APIs 1. `void vTaskPriorityInherit(xTaskHandle * const pxMutexHolder)`
The newly added private functions and macros are as follows:

1. `portTickType xServerGetRemainingBudget( void );`

2. `static void prvRemoveGlobalResourceFromList(tskTCB *pxTaskToDelete);`
3. `static void prvRemoveLocalResourceFromList(tskTCB *pxTaskToDelete);`

We adopted the following user APIs to incorporate HSF implementation. The original semantics of these API is kept and used when the user run the original FreeRTOS by setting `configHIERARCHICAL_SCHEDULING` macro to 0.

1. `OLD void vTaskStartScheduler( void );`

and adopted private functions and macros:

1. `OLD void vTaskSwitchContext( void );`

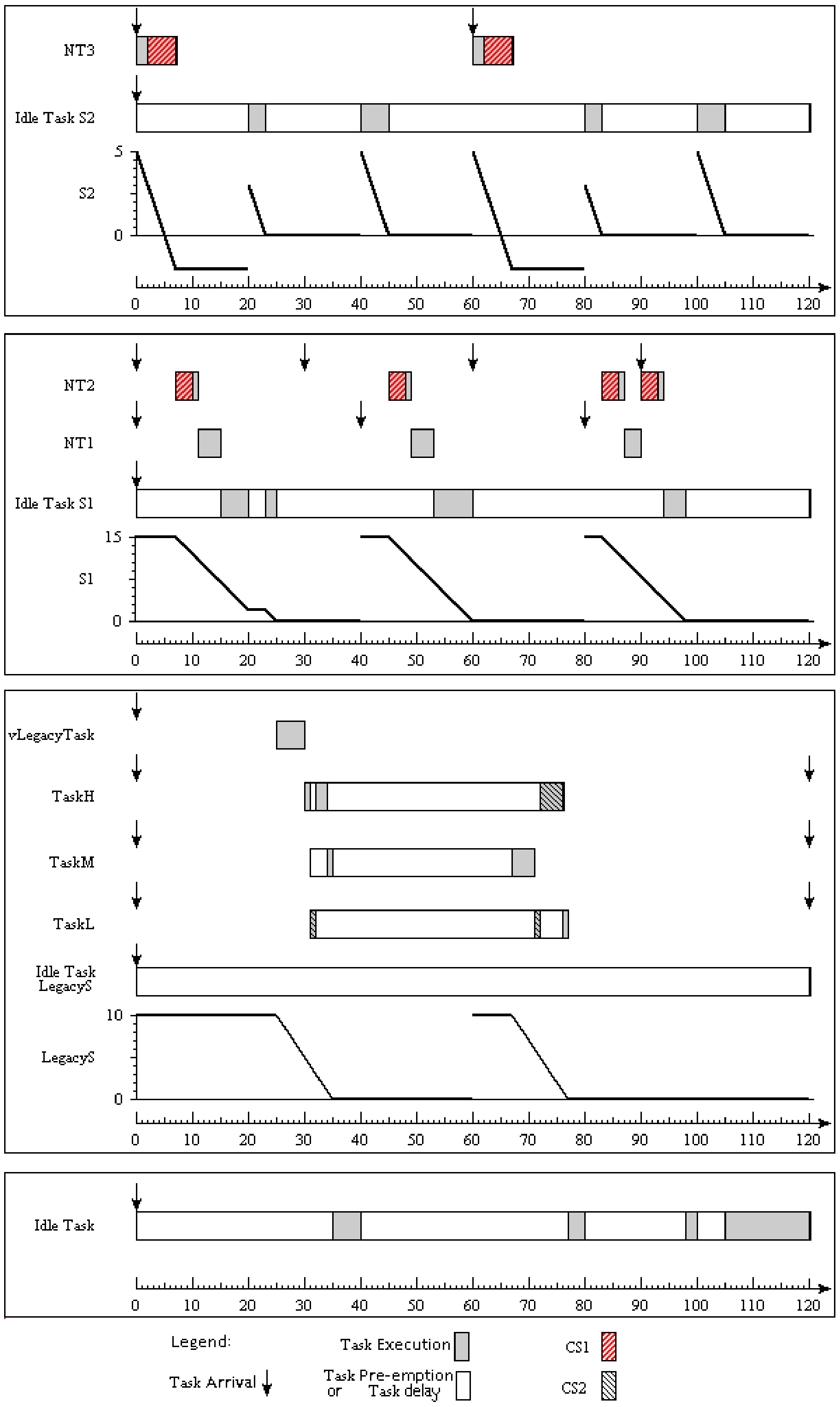


Figure 13: The trace for the execution of legacy application within a legacy server using binary semaphores

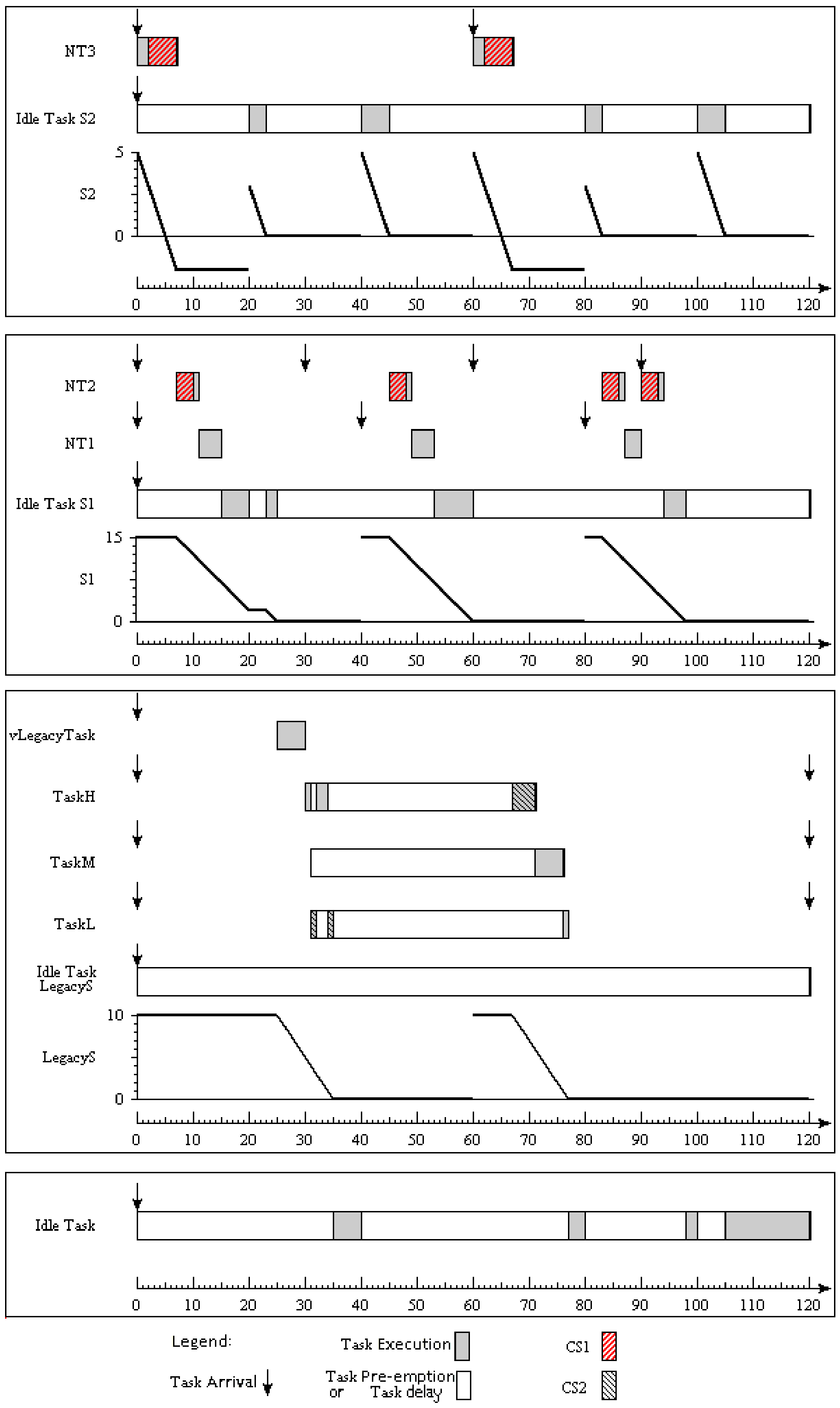
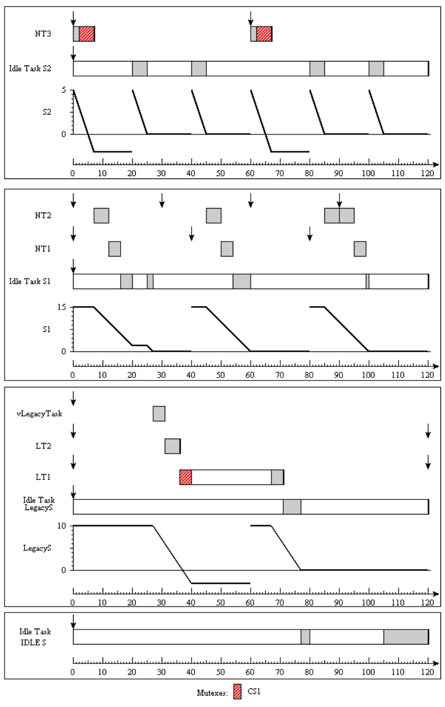
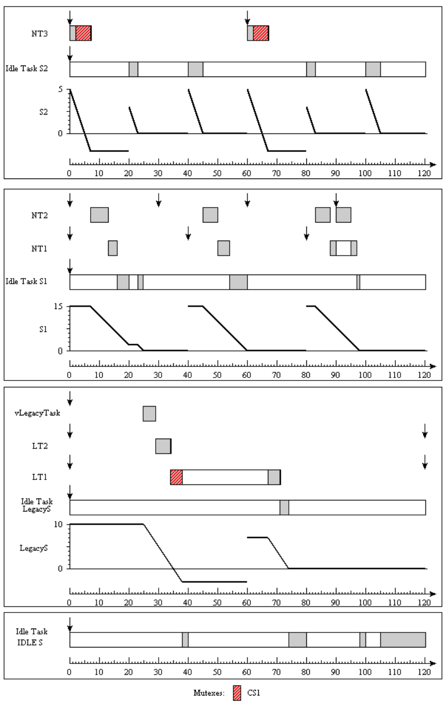


Figure 14: Trace showing the legacy server execution using mutex



(a) Trace of budget overrun without payback (BO)



(b) Trace of budget overrun with payback (PO)

Figure 15: Testing the behaviour of HSRP and budget overrun between the legacy application and a new server