Support for Hierarchical Scheduling in FreeRTOS*

Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, Sara Afshar
Mälardalen Real-Time Research Centre
Västerås, Sweden
Email: rafia.inam@mdh.se

***Abstract*—This paper presents the implementation of a Hierarchical Scheduling Framework (HSF) on an open source real-time operating system (FreeRTOS) to support the temporal isolation between a number of applications, on a single processor. The goal is to achieve predictable integration and reusability of independently developed components or applications. We present the initial results of the HSF implementation by running it on an AVR 32-bit board EVK1100.**

The paper addresses the fixed-priority preemptive scheduling at both global and local scheduling levels. It describes the detailed design of HSF with the emphasis of doing minimal changes to the underlying FreeRTOS kernel and keeping its API intact. Finally it provides (and compares) the results for the performance measures of idling and deferrable servers with respect to the overhead of the implementation.

***Index Terms*—real-time systems; hierarchical scheduling framework; fixed-priority scheduling**

I. Introduction

In real-time embedded systems, the components and component integration must satisfy both (1) functional correctness and (2) extra-functional correctness, such as satisfying timing properties. Temporal behavior of real-time components poses more difficulties in their integration. The scheduling analysis [1], [2] can be used to solve some of these problems, however these techniques only allow very simple models; typically simple timing attributes such as period and deadline are used. In addition, for large-scale real-time embedded systems, methodologies and techniques are required to provide not only spatial isolation but also temporal isolation so that the run-time timing properties could be guaranteed.

The Hierarchical Scheduling Framework (HSF) [3] is a promising technique for integrating complex real-time components on a single processor to overcome these deficiencies. It supplies an efficient mechanism to provide temporal partitioning among components and supports independent development and analysis of real-time systems. In HSF, the CPU is partitioned into a number of subsystems. Each subsystem contains a set of tasks which typically would implement an application or a set of components. Each task is mapped to a subsystem that contains a local scheduler to schedule the internal tasks of the subsystem. Each subsystem can use a different scheduling policy, and is scheduled by a global (system-level) scheduler.

We have chosen FreeRTOS [4] (a portable open source real-time scheduler) to implement hierarchical scheduling framework. Its main properties like open source, small footprint, scalable, extensive support for different hardware architectures, and easily extendable and maintainable, makes it a perfect choice to be used within the PROGRESS project [5].

* This work is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

A. Contributions

The main contributions of this paper are as follows:

- We have provided *a two-level hierarchical scheduling support* for FreeRTOS. We provide the support for a fixed-priority preemptive global scheduler used to schedule the servers and the support for idling and deferrable servers, using fixed-priority preemptive scheduling.
- We describe the *detailed design* of our implementation with the considerations of doing minimal changes in FreeRTOS kernel and keeping the original API semantics.
- We have *evaluated the performance measures* for periodic and deferrable servers on an AVR 32-bit board EVK1100 [6]. We also measure the overhead of the implementation, like tick handler, server context-switch and task context-switch.

B. The Hierarchical Scheduling Framework

A two-level HSF [7] can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 1. A leaf node contains its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler. The parent node is a global scheduler and is responsible for dispatching the servers according to their bandwidth reservation. A major benefit of HSF is that subsystems can be developed and analyzed in isolation from each other [8]. As each subsystem has its own local scheduler, after satisfying the temporal constraints of the subsystem, the temporal properties are saved within each subsystem. Later, the global scheduler is used to combine all the subsystems together without violating the temporal constraints that are already analyzed and stored in them. Accordingly we can say that the HSF provides partitioning of the CPU between different servers. Thus, server-functionality can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification, and unit testing.

Outline: Section II presents the related work on hierarchical scheduler and its implementations. In section III we provide our system model. Section IV gives an overview of FreeRTOS and the requirements to be incorporated into our design of HSF. We explain the implementation details of fixed-priority servers and hierarchical scheduler in section V. In section VI we test the behavior and evaluate the performance of our

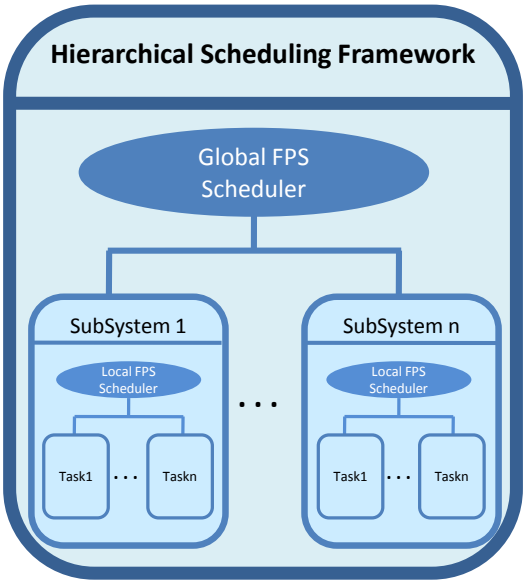

Fig. 1. Two-level Hierarchical Scheduling Framework

implementation, and in section VII we conclude the paper. We provide the API of our implementation in APPENDIX.

II. Related Work

A. Hierarchical Scheduling

HSF has attained a substantial importance since introduced in 1990 by Deng and Liu [3]. Numerous studies has been performed for the schedulability analysis of HSFs [9], [10] and processor models [11], [12], [13], [8] for independent subsystems. The main focus of this research has been on the schedulability analysis and not much work has been done to implement these theories.

B. Implementations of Hierarchical Scheduling Framework

Saewong and Rajkumar [14] implemented and analyzed HSF in CMU's Linux/RK with deferrable and sporadic servers using hierarchical deadline monotonic scheduling.

Buttazzo and Gai [15] present an HSF implementation based on Implicit Circular Timer Overflow Handler (ICTOH) using EDF scheduling for an open source RTOS, ERIKA Enterprise kernel.

A micro kernel called SPIRIT-μKernel is proposed by Kim *et al.* [7] based on two-level hierarchical scheduling. They also demonstrate the concept, by porting two different application level RTOS, VxWorks and eCos, on top of the SPIRIT-μKernel. The main focus is on providing a software platform for strongly partitioned real-time systems and lowering the overheads of kernel. It uses an offline scheduler at global level and the fixed-priority scheduling at local level to schedule the partitions and tasks respectively.

Behnam *et al.* [16] present an implementation of a two-level HSF in a commercial operating system VxWorks with the emphasis on not modifying the underlying kernel. The implementation supports both FPS and EDF at both global and local level of scheduling and a one-shot timer is used to trigger schedulers. The work presented in this paper is different from that of [16]. Our implementation aims at efficiency while modifying the kernel with the consideration of being consistent with the FreeRTOS API.

More recently, Holenderski *et al.* [17] implemented a two-level fixed-priority HSF in μC/OS-II, a commercial real-time operating system. This implementation is based on Relative Timed Event Queues (RELTEQ) [18] and virtual timers [19] and stopwatch queues on the top of RELTEQ to trigger timed events. They incorporated RELTEQ queues, virtual timers, and stopwatch queues within the operating system kernel and provided interfaces for it. Their HSF implementation uses these interfaces. Our implementation is different from that of [17] in the sense that we only extend the functionality of the operating system by providing support for HSF, and not changing or modifying the data structures used by the underlying kernel. We aim at efficiency, simplicity in design, and understandability and keeping the FreeRTOS original API intact. Also our queue management is very efficient and simple that eventually reduces the overhead.

III. System Model

In this paper, we consider a two-level hierarchical scheduling framework, in which a global scheduler schedules a system S that consists of a set of independently developed subsystems S_s, where each subsystem S_s consists of a local scheduler along with a set of tasks.

A. Subsystem model

Our subsystem model conforms to the periodic processor resource model proposed by Shin and Lee [8]. Each subsystem S_s, also called server, is specified by a subsystem *timing interface* $S_s(P_s, Q_s)$, where P_s is the period for that subsystem $(P_s > 0)$, and Q_s is the capacity allocated periodically to the subsystem $(0 < Q_s \leq P_s)$. At any point in time, B_s represents the remaining budget during the runtime of subsystem. During execution of a subsystem, B_s is decremented by one at every time unit until it depletes. If $B_s = 0$, the budget is depleted and S_s will be suspended until its next period where B_s is replenished with Q_s. Each server S_s has a unique priority p_s. There are 8 different subsystem priorities (from lowest priority 1 to the highest 7). Only idle server has priority 0. In the rest of this paper, we use the term subsystem and server interchangeably.

B. Task model

In the current implementation, we use a very simple task model, where each task τ_i is characterized only by its priority ρ_i. 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 idle task has priority 0) to the highest 255.

The local-level resource sharing among tasks of the same subsystem uses the FreeRTOS resource sharing methods. For the global-level resource sharing user should use some traditional waitfree [20] technique.

C. Scheduling Policy

We use a fixed-priority scheduling, FPS, at both the global and the local levels. FPS is the native scheduling of FreeRTOS, and also the predominant scheduling policy used in embedded systems industry. We use the First In First Out, FIFO, mechanism to schedule servers and tasks under FPS when they have equal priorities.

IV. FREERTOS

A. Background

FreeRTOS is a portable, open source (licensed under a modified GPL), mini real-time operating system developed by Real Time Engineers Ltd. It is ported to 23 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, with a binary image between 4 to 9KB.

Since most of the source code is in C language, it is readable, portable, and easily expandable and maintainable. Features like ease of use and understandability makes it very popular. More than $77,500$ official downloads in 2009 [21], and the survey result performed by professional engineers in 2010 puts the FreeRTOS at the top for the question "which kernel are you considering using this year" [22] showing its increasing popularity.

The FreeRTOS kernel supports preemptive, cooperative, and hybrid scheduling. In the fixed-priority preemptive scheduling, tasks with the same priority are scheduled using the Round-Robin (RR) policy. It supports any number of tasks and very efficient context-switching. FreeRTOS supports both static and dynamic (changed at run-time) priorities of the tasks. It has binary, counting and recursive semaphores and the mutexes for resource protection and synchronization, and queues for message passing among tasks. Its scheduler runs at the rate of one tick per milli-second by default, but it can be changed to any other value easily by setting the value of `configTICK_RATE_HZ` in the FreeRTOSConfig.h file.

We have extended FreeRTOS with a two-level hierarchical scheduling framework. The implementation is made under consideration of not changing the underlying operating system kernel unless vital and keeping the semantics of the original API. Hence the hierarchical scheduling of tasks is implemented with intention of doing as few modifications to the FreeRTOS kernel as possible.

B. Support for FIFO mechanism for local scheduling

Like many other real-time operating systems, FreeRTOS uses round robin scheduling for tasks with equal priorities. FreeRTOS uses `listGET_OWNER_OF_NEXT_ENTRY` macro to get the next task from the list to execute them in RR fashion. We change it to the FIFO policy to schedule tasks at local-level. We use `listGET_OWNER_OF_HEAD_ENTRY` macro to execute the current task until its completion. At global-level the servers are also scheduled using the FIFO policy.

C. Support for servers

In this paper we implement the idling periodic [23] and deferrable servers [24]. We need periodic activation of local servers to follow the periodic resource model [8]. To implement periodic activation of local servers our servers behave like periodic tasks, i.e. they replenish their budget Q_s every constant period P_s. A higher priority server can preempt and the execution of lower priority servers.

1) Support for idling periodic server: In the idling periodic server, the tasks execute and use the server's capacity until it is depleted. If server has the 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. One idle task per server is used to run when no other task is ready.

2) Support for deferrable server: In the deferrable server, the tasks execute and use the server's capacity until it is depleted. 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 server's period, it will be served and consumes server's capacity until the capacity depletes or the server's 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.

3) Support for idle server: When there is no other server in the system to execute, then an idle server will run. It has the lowest priority of all the other servers, i.e. 0. It contains only an idle task to execute.

D. System interfaces

We have designed the API with the consideration of being consistent in structure and naming with the original API of FreeRTOS.

1) Server interface: A server is created using the function `vServerCreate(period, budget, priority, *serverHandle)`. A macro is used to specify the server type as idling periodic or deferrable server in the config file.

2) Task interface: A task is created and assigned to the specific server by using the function `xServerTaskCreate()`. In addition to the usual task parameters passed to create a task in FreeRTOS, a handle to the server `serverHandle` is passed to this function to register the newly created task to its parent server. The original FreeRTOS API to create the task cannot be used in HSF.

E. Terminology

The following terms are used in this paper:

- **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:** A priority queue containing all the active servers.
- **Release-server list:** A priority queue containing all the inactive servers. It keeps track of system event: 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 can be used as feedback to resource management.
- **Ready-task list:** Each subsystem 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 idling server). In deferrable server, the idle task of idle server will run instead.

F. Design considerations

Here we present the challenges and goals of a HSF implementation that our implementation on FreeRTOS should satisfy:

1) **The use of HSF and the original FreeRTOS operating system:** User should be able to make a choice for using the HSF or the original FreeRTOS scheduler.
2) **Consistency with the FreeRTOS kernel and keeping its API intact:** To get minimal changes and better utilization of the system, it will be good to match the design of the HSF implementation with the underlying FreeRTOS operating system. This includes consistency from the naming conventions to API, data structures and the coding style. To increase the usability and understandability of HSF implementation for FreeRTOS users, major changes should not be made in the underlying kernel.
3) **Enforcement:** Enforcing server preemption at budget depletion; its currently executing task (if any) must be preempted and the server should be switched out. And similarly at budget replenishment, the server should become active; if its priority is highest among all the active servers then a server context-switch should be made and this server should execute.
4) **Monitoring budget consumption:** The budget consumption of the servers should be monitored to properly handle server budget depletion (the tasks of the server should execute until its budget depletion).
5) **The temporal isolation among servers must be guaranteed:** When one server is overloaded and its task miss the deadlines, it must not affect the execution of other servers. Also when no task is active to consume its server's capacity; in the idling server this capacity should idle away while in deferrable server it should be preserved.
6) **Protecting against interference from inactive servers:** The inactive servers should not interfere in the execution of active servers.
7) **Minimizing the overhead of server context-switch and tick handler:** For an efficient implementation, design considerations should be made to reduce these overheads.

V. Implementation

The user needs to set a macro `configHIERARCHICAL_SCHEDULING` as 1 or 0 in the configuration file `FreeRTOSConfig.h` of the FreeRTOS to start the hierarchical scheduler or the original FreeRTOS scheduler. The server type can be set via macro `configGLOBAL_SERVER_MODE` in the configuration file, which can be idling periodic or deferrable server. We are using FPS with the FIFO (to break ties between equal priorities) at both levels. We have changed the FreeRTOS RR policy to FIFO for the local schedulers, in order to use HSF-analysis in future. Further RR is costly in terms of overhead (increased number of context switches).

Each server has a server control block, `subSCB`, containing the server's parameters and lists. The servers are created by calling the API `xServerCreate()` that creates an idling or deferrable server depending on the server type macro value, and do the server initializations which includes `subSCB` value's initialization, and initialization of server lists. It also creates an idle task in that server. An idle server with an idle task is also created to setup the system. The scheduler is started by calling `vTaskStartScheduler()` (typically at the end of the `main()` function), which is a non-returning function. Depending on the value of the `configHIERARCHICAL_SCHEDULING` macro, either the original FreeRTOS scheduler or the hierarchical scheduler will start execution. `vTaskStartScheduler()` then initializes the system-time to 0 by setting up the timer in hardware.

A. System design

Here we describe the details of design, implementation, and functionality of the two-level HSF in FreeRTOS.

1) The design of the scheduling hierarchy: The global scheduler maintains a running server pointer and two lists to schedule servers: a ready-server list and a release-server list. A server can be either in ready-server or release-server list at any time, and is implied as active or inactive respectively. Only one server from the ready-server list runs at a time.

Running server: The running server is identified by a pointer. This server has the highest priority among all the currently ready servers in the system.

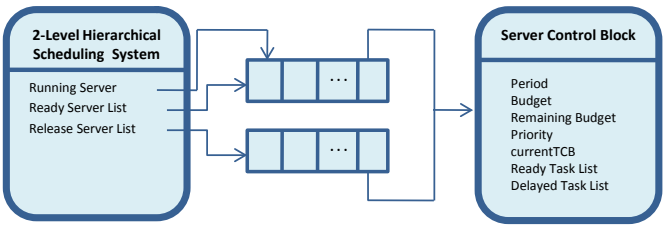

Fig. 2. Data structures for active and inactive servers

At any time instance, only the tasks of the currently running server will run according to the fixed-priority scheduling policy. When a server context-switch occurs, the running server pointer is changed to the newly running server and all the tasks of the new running server become ready for execution.
Ready-server list: contains all the servers that are active (whose remaining budgets are greater than zero). This list is maintained as a double linked list. The `ListEnd` node contains two pointers; `ListEnd.previous` and `ListEnd.next` that point to the last node and first node of the list respectively as shown in Figure 3. It is the FreeRTOS structure of list, and provides a quick access to list elements, and very fast modifications of the list. It is ordered by the priority of servers, the highest priority ready server is the first node of the list.

Fig. 3. The structure of ready-server and release-server lists

Release-server list: contains all the inactive servers whose budget has depleted (their remaining budget is zero), and will be activated again at their next activation periods. This list is maintained as a double linked list as shown in Figure 3 and is ordered by the next replenishment time of servers, which is the absolute time when the server will become active again.

2) The design of the server: The local scheduler schedules the tasks that belong to a server in a fixed-priority scheduling manner. Each server is specified by a server Control Block, called `subSCB`, that contains all information needed by a server to run in the hierarchical scheduling, i.e. the period, budget, remaining budget, priority and the queues as presented in Figure 4.

Each server maintains a currently running task and two lists to schedule its tasks: a ready-task list, and a delayed-task list. Ready task and delayed task lists have the same structure as the FreeRTOS scheduler has. Delayed-task list is the FreeRTOS list and is used by the tasks that are delayed because of the FreeRTOS `vTaskDelay` or `vTaskDelayUntil` functions.
Current running task: `currentTCB` is a FreeRTOS pointer that always points to the currently running task in the system. This is the task with the highest priority among all the currently ready tasks of the running server's ready task list.

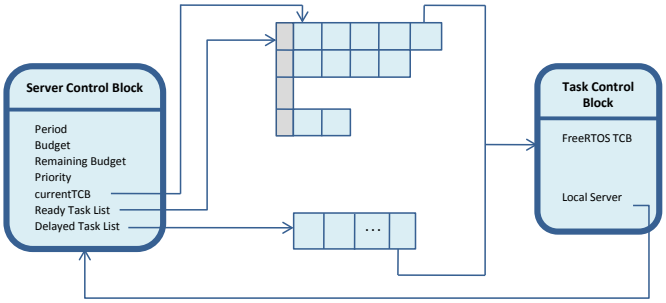

Fig. 4. Data structures for ready and delayed tasks

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 currently running server. When a server starts executing, its ready-task list becomes active, and `currentTCB` points to the highest priority task. This list is maintained in a similar way as FreeRTOS ready list, because we do not want to make major changes in the underlying operating system.

A separate ready-task list for each server reduces the server context-switch overhead, since the tasks swapping at every server context-switch is very costly. Further it also keeps our implementation consistent with the FreeRTOS.

The ready task list is an array of circular double linked lists of the tasks. The index of the array presents the priorities of tasks within a subsystem as shown by the gray color in Figure 5. By default, FreeRTOS uses 8 different priority levels for the tasks from lowest priority 1 (only idle task has priority 0) to the highest 7. (User can change it till the maximum 256 different task priorities). The tasks of the same priority are placed as a double linked list at the index of that particular priority. The last node of the double linked list at each index is End pointer that points to the previous (the last node of the list) and to the next (the first node of the list) as shown in the Figure 5. For insertions the efficiency is $O(1)$, and for searching it is $O(n)$ in the worst case, where n is the maximum allowed priority for tasks in the subsystem.

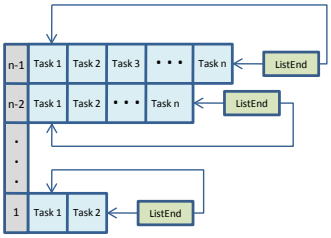

Fig. 5. The structure of ready-task list

Tasks: We added a pointer to the task `TCB`, that points to its parent server control block to which this task belongs. This is the only addition done to the `TCB` of FreeRTOS to adopt it to the two-level hierarchical scheduling framework.
Server context-switch: Since each subsystem has its own ready list for its tasks, the server context switch is very light-

weight. It is only the change of a pointer, i.e. from the task list of the currently executing server to the ready-task list of the newly running server. At this point, the ready-task list of the newly running server is activated and all the tasks of the list become ready for execution.

Task context-switch: We are using the FreeRTOS task context-switch which is very fast and efficient as evaluated in Section VI-B1. At this point, the ready-task list of the newly running server is activated and all the tasks of the list become ready for execution.

B. System functionality

1) The functionality of the tick handler: The tick handler is executed at each system tick (1ms be default). At each tick interrupt:

- The system tick is incremented.
- Check for the server activation events. Here the activation time of (one or more) servers is checked and if it is equal to the system time then the server is replenished with its maximum budget and is moved to the ready-server list.
- The global scheduler is called to incorporate the server events.
- The local scheduler is called to incorporate the task events.

2) The functionality of the global scheduler: In a two-level hierarchical scheduling system, a global scheduler schedules the servers (subsystems) in a similar fashion as the tasks are scheduled by a simple scheduler. The global scheduler is called by the `prvScheduleServers()` kernel function from within the tick-handler. The global scheduler performs the following functionality:

- At each tick interrupt, the global scheduler decrements the remaining budget B_s of the running server by one and handles 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 run and makes a server context-switch if required. `prvChooseNextIdlingServer()` or `prvChooseNextDeferrableServer()` is called to select idling or deferrable server, depending on the `configGLOBAL_SERVER_MODE` macro. All the events that occurred during inactive state of the server (tasks activations) are handled here.
- `prvAdjustServerNextReadyTime(pxServer)` is called to set up the next activation time to activate the server periodically.

In idling server, `prvChooseNextIdlingServer()` simply selects the first node (with highest priority) from the ready-server list and makes it the current running server. While in case of deferrable server, the `prvChooseNextDeferrableServer()` function checks in the ready-server list for the next ready server that has any task ready to execute even if the currently running server has no ready task and its budget has 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.

3) The functionality of the local scheduler: The local scheduler is called from within the tick interrupt using the adopted FreeRTOS kernel function `vTaskSwitchContext()`. The local scheduler is the original FreeRTOS scheduler with the following modifications:

- The round robin scheduling policy among equal priority tasks is changed to FIFO policy.
- 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.

C. Addressing design considerations

Here we address how we achieve the design requirements that are presented in Section IV-F.

1) **The use of HSF and the original FreeRTOS operating system:** We have kept all the original API of FreeRTOS, and the user can choose to run either the original FreeRTOS operating system or the HSF by just setting a macro `configHIERARCHICAL_SCHEDULING` to 0 or 1 respectively in the configuration file.
2) **Consistency with the FreeRTOS kernel and keeping its API intact:** We have kept consistency with the FreeRTOS from the naming conventions to the data structures used in our implementations; for example ready-task list, ready and release server lists. These lists are maintained in a similar way as of FreeRTOS. We have kept the original semantics of the API and the user can run the original FreeRTOS by setting `configHIERARCHICAL_SCHEDULING` macro to 0.
3) **Enforcement:** At each tick interrupt, the remaining budget of the running server is checked and at budget depletion (remaining budget becomes 0), the server is moved from active (ready-server list) to the inactive (release-server list) state. Moreover, release-server list is also checked for the periodic activation of servers at each system tick and at budget replenishment of any server, it is moved from inactive to active state. Preemptive scheduling policy makes it possible.
4) **Monitoring budget consumption:** The remaining budget variable of each server's `subSCB` is used to monitor the consumption. At each system tick, the remaining budget of the running server is decremented by one, and when it exhausts the server is moved from active to the inactive state.
5) **The temporal isolation among servers must be guaranteed:** We tested the system and an idle task runs when there is no task ready to execute. To test the temporal isolation among servers, we use an *Idle server* that runs when no other server is active. It is used in testing the temporal isolation among servers. Section VI-A illustrates the temporal isolation.
6) **Protecting against interference from inactive servers:** The separation of active and inactive servers in separate

server queues prevents the interference from inactive servers and also poses less overhead in handling system tick interrupts.

7) **Minimizing the overhead of server context-switch and tick handler:** A separate ready-task list for each subsystem reduces the task swapping overhead to only the change of a pointer. Therefore, the server context-switch is very light-weight. The access to such a structure of ready list is fast and efficient especially in both inserting and searching for elements. Further the tasks swapping at every server context-switch is very heavy in such a structure.

VI. Experimental Evaluation

In this section, we present the evaluation of behavior and performance of our HSF implementation. All measurements are performed on the target platform EVK1100 [6]. The AVR32UC3A0512 micro-controller runs at the frequency of 12MHz and its tick interrupt handler at 1ms.

A. Behavior testing

In this section we perform two experiments to test the behavior our implementation. Two servers S1, and S2 are used in the system, plus an idle server is created. The servers used to test the system are given in Table I.

Server	S1	S2
Priority	2	1
Period	20	40
Budget	10	15

TABLE I
SERVERS USED TO TEST SYSTEM BEHAVIOR.

Test1: This test is performed to check the behavior of idling periodic and deferrable servers by means of a trace of the execution. Task properties and their assignments to the servers is given in Table II. Note that higher number means higher priority for both servers and tasks. The visualization of the execution for idling and deferrable servers is presented in Figure 6 and Figure 7 respectively.

Tasks	T1	T2	T3
Servers	$S1$	$S1$	$S2$
Priority	1	2	2
Period	20	15	60
Execution Time	4	2	10

TABLE II
TASKS IN BOTH SERVERS.

In the diagram, the horizontal axis represents the execution time starting from 0. In the task's visualization, the arrow represents task arrival, a gray rectangle means task execution, a solid white rectangle represents either local preemption by another task in the server or budget depletion, and a dashed white rectangle means the global preemption. In the server's visualization, the numbers along the vertical axis are the server's capacity, the diagonal line represents the server execution while the horizontal line represents either the waiting time for the next activation (when budget has depleted) or the waiting for its turn to execute (when some other server is executing).

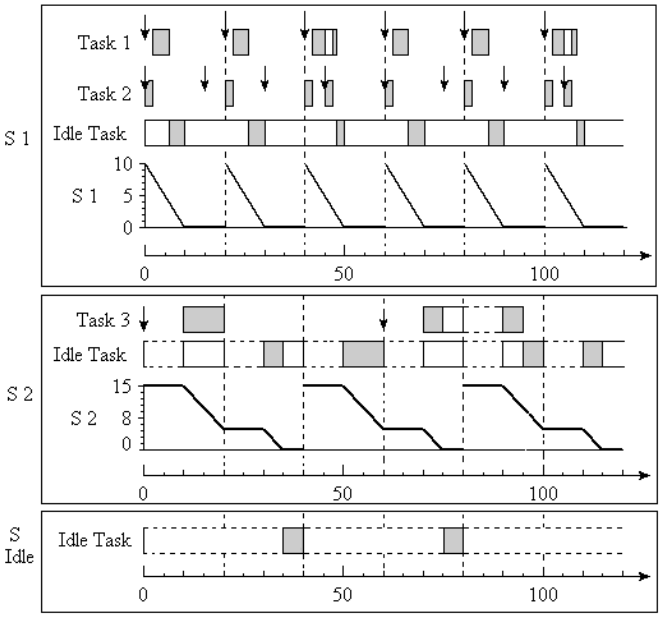

Fig. 6. Trace for idling periodic servers

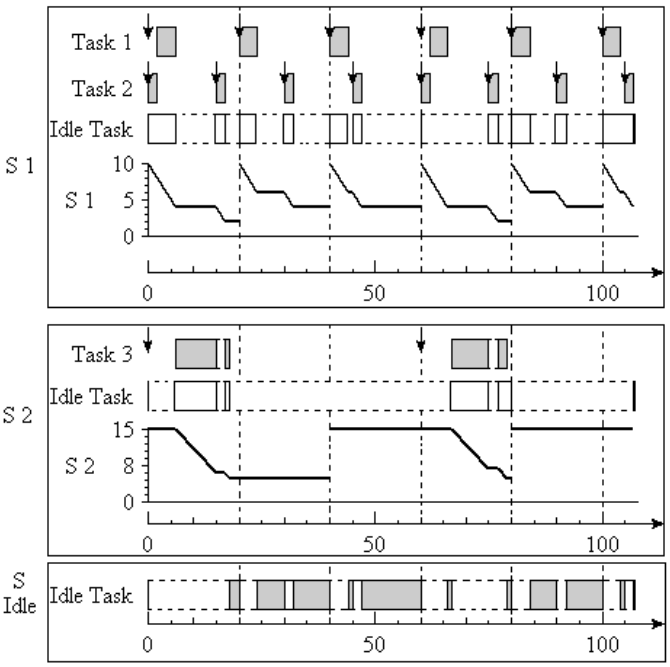

Fig. 7. Trace for deferrable servers

The difference in idling and deferrable servers is clear from these Figures. In idling periodic servers, all the servers in the system executes till budget depletion, if no task is ready then the idle task of that server executes till its budget depletion. While in deferrable servers, when no task is ready in the server

even if it has the capacity, the server will give the chance to another server to execute and preserves its capacity. Thats why there is no idle task (of S1 and S2) execution in deferrable servers as obvious from Figure 7. When no task is ready to execute in the system, then idle task of idle server will execute.

Test2: The purpose of this test is to evaluate the system behavior during the overload situation and to test the temporal isolation among the servers. For example, if one server is overloaded and its tasks miss deadlines, it must not affect the behavior of other servers in the system.

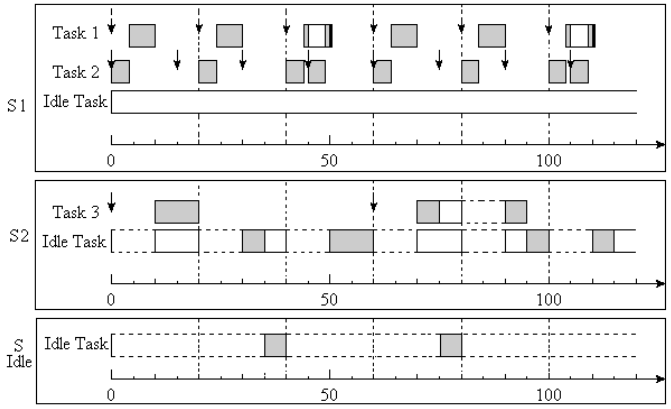

Fig. 8. Trace showing temporal isolation among idling servers

The same example is executed to perform this test but with the increased utilization of S1. The execution times of T1 and T2 are increased to 4 and 6 respectively, hence making the server S1 utilization greater than 1. Therefore the low priority task T1 misses its deadlines as shown by solid black lines in the Figure 8. S1 is never idling because it is overloaded. It is obvious from Figure 8, that the overload of S1 does not effect the behavior of S2 even though it has low priority.

B. Performance assessments

Here we present the results of the overhead measurements for the idling and deferrable servers. The time required to run the global scheduler (to schedule the server) is the first extra functionality needed to be measured; it includes the overhead of server context-switch. The tick interrupt handler is the second function to be measured; it encapsulated the global scheduler within it, hence the overhead measurement for tick interrupt represents the sum of tick-increasing time and global scheduler time. The third overhead needed to be assessed is the task context-switch.

Two test scenarios are performed to evaluate the performance for both idling and deferrable servers. For each measure, a total of 1000 values are computed. The minimum, maximum, average and standard deviation on these values are calculated and presented for both types of servers. All the values are given in micro-seconds (μs).

1) Test Scenario 1: For the first performance test, 3 servers, S1, S2, and S3 are created with a total of 7 tasks. S1 contains 3 tasks while S2 and S3 has 2 tasks each. The measurements are extracted for task and server context-switches, global scheduler and tick interrupt handler and are reported below.

Task context switch: The FreeRTOS context-switch is used for doing task-level switching. We found it very efficient, consistent and light-weight, i.e. 10μs always as obvious from Table III.

Server type	Min.	Max.	Average	St. Deviation
Idling	10	10	10	0
Deferrable	10	10	10	0

TABLE III
THE TASK CONTEXT-SWITCH MEASURES FOR BOTH SERVERS.

Choosing next server: It is *fetching the highest priority server (first node from the server ready queue)*, and it is very fast for both types of servers as given in Table IV. Note that the situations where there is no need to change the server, it becomes 0 and this situation is excluded from these results.

Server type	Min.	Max.	Average	St. Deviation
Idling	10	10	10	0
Deferrable	10	32	14.06593	5.6222458

TABLE IV
THE SERVER CONTEXT-SWITCH MEASURES FOR BOTH SERVERS.

The deferrable overhead is greater than idling server because of the increased functionality, as explained in Section V-B2.

Global scheduler: The WCET of the global scheduler is dependent on the number of events it handles. As explained in Section V-B2, the global scheduler handles the server activation events and the events which has been postponed during inactive time in this server, therefore, its execution time depends on the number of events. The overhead measures for global scheduler function to execute for both types of servers are given in Table V.

Server type	Min.	Max.	Average	St. Deviation
Idling	10	53	12.33666	6.0853549
Deferrable	10	42	13.34865	7.5724052

TABLE V
THE GLOBAL SCHEDULER OVERHEAD MEASURES FOR BOTH SERVERS.

Tick interrupt handler: It includes the functionality of global and local schedulers. The WCET of the tick handler is dependent on the number of servers and tasks in the system. Note that the task context-switch time is excluded from this measurement.

Server type	Min.	Max.	Average	St. Deviation
Idling	32	74	37.96903	7.00257381
Deferrable	32	85	41.17582	10.9624383

TABLE VI
THE TICK INTERRUPT OVERHEAD MEASURES FOR BOTH SERVERS.

Again the deferrable overhead is greater than that of the idling server because of the increased functionality and increased number of server context-switches at run-time.

2) Test Scenario 2: The experiments are run to check heavy system loads. The setup includes $10, 20, 30,$ and 40 servers in the system, each running a single task in it. We cannot create more than 40 idling servers, and more than 30 deferrable servers due to memory limitations on our hardware platform. For this test scenario we only measured the overheads for the global scheduler and the tick interrupt handler, because choosing next server is part of global scheduler and because the time to execute task context-switch is not affected by the increase of number of servers in the system.

Global scheduler: The values for idling and deferrable servers are presented in Table VII and VIII respectively.

Number of servers	Min.	Max.	Average	St. Deviation
10	10	21	10.0439	0.694309682
20	10	32	10.1538	1.467756006
30	10	32	10.3956	2.572807933
40	10	32	10.3186	2.258614766

TABLE VII
THE GLOBAL SCHEDULER OVERHEAD MEASURES FOR IDLING SERVER.

Number of servers	Min.	Max.	Average	St. Deviation
10	10	53	25.84	8.950729331
20	10	53	25.8434	11.90195638
30	10	53	27.15	9.956851354

TABLE VIII
THE GLOBAL SCHEDULER OVERHEAD MEASURES FOR DEFERRABLE SERVER.

The global scheduler's overhead measures are dependent on the number of events it handles as explained in Section V-B2. In this test scenario, there is only one task per server, that reduces the number of events to be handled by the global scheduler, therefore, the maximum overhead values in Table VII are less than from those of Table V. The same reasoning stands for deferrable server too.

Tick interrupt handler: The measured overheads for idling and deferrable servers are reported in Table IX and X respectively. These do not include the task context-switch time.

Number of servers	Min.	Max.	Average	St. Deviation
10	53	96	64.57742	4.656420272
20	96	106	98.35764	4.246876974
30	128	138	132.2058	4.938988398
40	160	181	164.8022	5.986888605

TABLE IX
THE TICK INTERRUPT OVERHEAD MEASURES FOR IDLING SERVERS.

From Tables VI, IX, X it is clear that the tick interrupt overhead increases with the increase in the number of servers in the system.

C. Summary of Evaluation

We have evaluated our implementation on an actual real environment i.e. a 32-bit EVK1100 board hence our results are more valid than simulated results like [17] where the simulation experiments are simulated for OpenRisc 1000 architecture and hence having a very precise environmental behavior. We have evaluated the behavior and performance of our implementation for *resource allocation during heavy load,* and *overload* situations, and found that it behaves correctly and gives very consistent results.

Number of servers	Min.	Max.	Average	St. Deviation
10	106	128	126.2574	4.325860528
20	140	149	144.5446	4.522222357
30	172	181	178.7723	3.903539901

TABLE X
THE TICK INTERRUPT OVERHEAD MEASURES FOR DEFERRABLE SERVERS.

We have also evaluated the efficiency of our implementation, i.e. the efficiency of task context-switch, global scheduler, and tick handler. Searching for the highest priority server and task the efficiencies are $O(1)$ and $O(n)$ respectively, where n is the maximum allowed priority for tasks in the subsystem. For insertions, it is $O(m)$ and $O(1)$ for the server and task respectively, where m is the number of servers in the system in the worst case. Our results for task context-switch, and choosing scheduler conforms this efficiency as compared to [17], where the efficiency is also dependent on dummy events in the RELTEQ queues. These dummy events are not related to the scheduler or tasks, but to the RELTEQ queue management.

VII. CONCLUSIONS

In this paper, we have implemented a two-level hierarchical scheduling support in an open source real-time operating system, FreeRTOS, to support temporal isolation among real-time components. We have implemented idling periodic and deferrable servers using fixed-priority preemptive scheduling at both local and global scheduling levels. We focused on being consistent with the underlying operating system and doing minimal changes to get better utilization of the system. We presented our design details of two-level HSF and kept the original FreeRTOS API semantics.

We have tested our implementations and presented our experimental evaluations performed on EVK1100 AVR32UC3A0512 micro-controller. We have checked it during heavy-load and over-load situations and have reported our results. It is obvious from the results of the overhead measurements (of tick handler, global scheduler, and task context-switch) that the design decisions made and the implementation is very efficient.

In the future we plan to implement support for legacy code in our HSF implementation for the FreeRTOS i.e. to map the FreeRTOS API to the new API, so that the user can run her/his old code in a subsystem within the HSF. We will implement the periodic task model and a lock-based synchronization protocol [25] for global resource sharing among servers. We also want to improve the current Priority Inheritance Protocol for local resource sharing of FreeRTOS by implementing Stack Resource Protocol. And finally we want to integrate this work

within the virtual node concept [5].

REFERENCES

[1] L. Sha, T. Abdelzaher, K-E. rzn, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2/3):101–155, 2004.

[2] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, pages 16–25, June 1995.

[3] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. 18^{th} IEEE Real-Time Systems Symposium (RTSS)*, 1997.

[4] FreeRTOS web-site. http://www.freertos.org/.

[5] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *22nd Euromicro Conference on Real-Time Systems (ECRTS10) WiP Session*, pages 17–20, July 2010.

[6] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4114.

[7] Daeyoung Kim, Yann-Hang Lee, and M. Younis. Spirit-ukernel for strongly partitione real-time systems. In *Proc. of the 7^{th} International conference on Real-Time Computing Systems and Applications (RTCSA'00)*, 2000.

[8] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 2–13, 2003.

[9] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proc. 20^{th} IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[10] G. Lipari and S.Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proc. 6^{th} IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 166–175, 2000.

[11] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *ACM Intl. Conference on Embedded Software(EMSOFT'04)*, pages 95–103, 2004.

[12] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *IEEE Real-Time Systems Symposium (RTSS'02)*, pages 26–35, 2002.

[13] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS'05)*, pages 389–398, 2005.

[14] S. Saewong and R. Rajkumar. Hierarchical reservation support in resource kernels. In *Proc. 22^{th} IEEE Real-Time Systems Symposium (RTSS)*, 2001.

[15] G. Buttazzo and P. Gai. Efficient edf implementation for small embedded systems. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'06)*, 2006.

[16] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards hierarchical scheduling on top of vxworks. In *Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, pages 63–72, July 2008.

[17] Mike Holenderski, Wim Cools, Reinder J. Bril, and J. J. Lukkien. Extending an Open-source Real-time Operating System with Hierarchical Scheduling. Technical Report, Eindhoven University, 2010.

[18] Mike Holenderski, Wim Cools, Reinder J. Bril, and J. J. Lukkien. Multiplexing Real-time Timed Events. In *Work in Progress session of the IEEE International Conference on Emerging Techonologies and Factory Automation (ETFA09)*, 2009.

[19] M.M.H.P. van den Heuvel, Mike Holenderski, Wim Cools, Reinder J. Bril, and Johan J. Lukkien. Virtual Timers in Hierarchical Real-time Systems. In *Work in Progress Session of the IEEE Real-Time Systems Symposium (RTSS09)*, December 2009.

[20] Håkan Sundell and Philippas Tsigas. Simple Wait-Free Snapshots for Real-Time Systems with Sporadic Tasks. In *Proceedings of the (RTCSA 2004)*, pages 325–240.

[21] Microchip web-site.

[22] EE TIMES web-site. http://www.eetimes.com/design/embedded/4008920/The-results-for-2010-are-in-.

[23] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 181–191, 1986.

[24] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments. *IEEE Transactions on Computers*, 44(1), 1995.

[25] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 6(1), February 2010.

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 xServerCreate(xPeriod, xBudget, uxPriority, *pxCreatedServer);`
2) `signed portBASE_TYPE xServerTaskGenericCreate( pxTaskCode, pcName, usStackDepth, *pvParameters, uxPriority, *pxCreatedTask, pxCreatedServer, *puxStackBuffer, xRegions ) PRIVILEGED_FUNCTION;`
3) `#define xServerTaskCreate( pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCreatedTask, pxCreatedServer ) xServerTaskGenericCreate( (pvTaskCode), (pcName), (usStackDepth), (pvParameters), (uxPriority), (pxCreatedTask), (pxCreatedServer), ( NULL ), ( NULL ))`
4) `portTickType xServerGetRemainingBudget( void );`

The newly added private functions and macros are as follows:

1) `#define prvAddServerToReadyQueue( pxSCB )`
2) `#define prvAddServerToReleaseQueue( pxSCB )`
3) `#define prvAddServerToOverflowReleaseQueue( pxSCB )`
4) `#define prvChooseNextDeferrableServer( void )`
5) `#define prvChooseNextIdlingServer( void )`
6) `static inline void prvAdjustServerNextReadyTime( *pxServer );`
7) `static void prvInitialiseServerTaskLists( *pxServer );`
8) `static void prvInitialiseGlobalLists(void);`
9) `static signed portBASE_TYPE prxRegisterTasktoServer(* pxNewTCB, *pxServer);`
10) `static signed portBASE_TYPE prxServerInit(* pxNewSCB);`
11) `static signed portBASE_TYPE xIdleServerCreate(void);`
12) `static void prvScheduleServers(void);`
13) `static void prvSwitchServersOverflowDelayQueue(* pxServerList);`
14) `static void prvCheckServersDelayQueue(* pxServerList);`

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) `signed portBASE_TYPE xTaskGenericCreate( pxTaskCode, pcName, usStackDepth, *pvParameters, uxPriority, *pxCreatedTask, *puxStackBuffer, xRegions );`
2) `void vTaskStartScheduler( void );`
3) `void vTaskStartScheduler (void);`
4) `void vTaskDelay( xTicksToDelay );`
5) `void vTaskDelayUntil( pxPreviousWakeTime, xTimeIncrement);`

and adopted private functions and macros:

1) `#define prvCheckDelayedTasks(pxServer)`
2) `#define prvAddTaskToReadyQueue( pxTCB )`
3) `void vTaskIncrementTick( void );`
4) `void vTaskSwitchContext( void );`