6-2021

Hierarchical Scheduling for Real-Time Periodic Tasks in Symmetric Multiprocessing

Tom Springer

Peiyi Zhao

Hierarchical Scheduling for Real-Time Periodic Tasks in Symmetric Multiprocessing

HIERARCHICAL SCHEDULING FOR REAL-TIME PERIODIC TASKS IN SYMMETRIC MULTIPROCESSING

Tom Springer and Peiyi Zhao

Fowler School of Engineering, Chapman University, Orange, CA, USA

ABSTRACT

In this paper, we present a new hierarchical scheduling framework for periodic tasks in symmetric multiprocessor (SMP) platforms. Partitioned and global scheduling are the two main approaches used by SMP based systems where global scheduling is recommended for overall performance and partitioned scheduling is recommended for hard real-time performance. Our approach combines both the global and partitioned approaches of traditional SMP-based schedulers to provide hard real-time performance guarantees for critical tasks and improved response times for soft real-time tasks. Implemented as part of VxWorks, the results are confirmed using a real-time benchmark application, where response times were improved for soft real-time tasks while still providing hard real-time performance.

KEYWORDS

Real-time systems, hierarchical scheduling, symmetric multiprocessing, operating systems.

1. INTRODUCTION

The next generation embedded systems are working to consolidate large complex workloads onto multi-core platforms with mixed real-time applications. The existing architecture typically uses distributed uniprocessors connected over a common backplane where one processor may be assigned a soft real-time (SRT) task set and another processor a hard real-time (HRT) task set. The problem with this approach is it limits the computational throughput and increases costs as compared to multi-core platforms. It is for these reasons; designers are looking to re-host these new complex workloads onto multi-core platforms to reduce the size, weight and power (SWaP) requirements of traditional distributed systems.

Therefore, in this paper we look into symmetric multiprocessing (SMP) because most multi-core systems use SMP architecture. Briefly, SMP is a computing framework that manages the processing of tasks across multiple homogeneous processors or cores1 that share a common operating system, memory and I/O data path. One major challenge for SMP in mixed real-time scheduling is to effectively balance the competing needs of HRT and SRT tasks, such as temporal isolation, resource allocation or fault mitigation.

There are two main scheduling approaches for a SMP-based system: partitioned and global scheduling. Partitioned scheduling binds a task to a specific processor or core while global scheduling allows a task to migrate across multiple cores. Researchers have studied the

[1] Note that core and processor will be used interchangeably to indicate the basic computation unit of the CPU

David C. Wyld et al. (Eds): BIoT, DKMP, CCSEA, EMSA - 2021
pp. 91-111, 2021. CS & IT - CSCP 2021 DOI: 10.5121/csit.2021.110810

schedulability of both approaches and have concluded that no single method dominates the other for all task sets [1]. Global scheduling provides better average case response times by performing load-balancing across multiple cores. However, the superior average case performance of global scheduling is not easily extended to hard real-time performance guarantees. For example, when performing load-balancing a global scheduler may migrate a task to another core and as a result invalidate the local cache. This invalidation process proves costly and can severely impact the determinism of the affected task.

On the other hand, partitioned scheduling statically assigns tasks to a specific core which can control task migration. Also known as CPU affinity, the idea is the designer can specify which tasks to run on a specific core then the scheduler obeys the order and only runs those tasks on the specified core. It also makes logical sense to bind all the tasks that access the same data to the same core(s) in this way they do not contend over data and ensure the task receives the full attention of the processor. However, when tasks are statically assigned to specific cores an unbalanced load distribution is likely to occur leading to a less than optimal utilization of the overall system.

Another concern involves the diversity and complexity of the various computational workloads in these next generation systems. Processing and criticality requirements may vary significantly where different operating modes could have vastly different workloads. In addition to the computational variations, mission critical type systems must perform continuously in harsh environments where they are expected to perform at least a subset of some critical functions under an overloaded or fault condition. The occurrence of an overload or fault must not hinder the overall survivability of the embedded system. Consequently, what is needed may be a more collective type of resource allocation where tasks are assigned resources according to their functionality requirements. In this way, applications can be grouped by service classes based upon their processing and criticality constraints.

Unfortunately, traditional SMP-based schedulers are not suitable to this type of collective resource allocation because they perform fine-grained scheduling at the task level. Since, these schedulers do not differentiate between tasks of different applications system-wide performance may not be the ideal metric for application specific requirements. Additionally, HRT and SRT tasks have competing objectives. HRT tasks require strict timing constraints where deadline misses are not tolerated. While SRT tasks can accept some deadlines misses but place a greater premium on task response time.

To solve these issues in this paper we present a new multi-core hierarchical scheduling framework (HSP) for periodic tasks in SMP-based systems. Our objective is to provide a hierarchical scheduling mechanism that can more effectively adapt to execution time variations in mixed real-time environments. Traditionally, the approach to scheduling mixed real-time applications has been to provide conservative WCET values to ensure the timing correctness of the HRT tasks. The problem with this approach is it usually leads to underutilized resources and poor response times because the actual WCET value of a task is rarely realized. As a result we look to exploit this underutilization by utilizing both the partitioned and non-partitioned scheduling mechanisms of a SMP-based system.

The benefits of this new scheduler are: (1) Better determinism for hard real-time tasks and improved response times for soft-real time tasks as compared to the global and partitioned scheduling methods of traditional SMP-based schedulers. (2) An application based resource allocation scheme which enhances scalability by reducing excessive interprocessor communication, bus contention and synchronization overhead. (3) A scheduling mechanism which provides for improved resource utilization and task acceptance rates. (4) Temporal

isolation for hard real-time tasks where lower priority tasks cannot affect the timing behavior during overload or fault conditions.

The remainder of this paper is organized as follows. Section 2 provides an overview of the hierarchical scheduling framework used by our scheduling mechanism. Section 3 discusses previous work on hierarchical scheduling and SMP based scheduling mechanisms. Section 4 provides an overview of our hierarchical scheduler (HSP). Section 5 presents the schedulability analysis of our scheduler in a multicore environment. Section 6 utilizes task set simulations to provide comparisons between our hierarchical scheduling approach and the scheduling mechanisms for a traditional SMP-based scheduler. Section 7 describes the implementation of our hierarchical scheduling mechanism as an extension to Wind River's VxWorks RTOS and ported onto a commercially available multi-core processor. In Section 8 we conclude with future work and the research summary

2. PRELIMINARIES

This section provides a discussion of the terminology used in the paper as well as an overview of hierarchical scheduling to provide as a reference for understanding the overall architecture of hierarchical scheduling in a symmetric multiprocessing environment.

2.1. Terminology

We consider a periodic task model defined as $\tau_i\,(T_i, C_i^{Lo}, C_i^{Hi}, D_i)$, where T_i is defined as the task period, C_i^{Lo} and C_i^{Hi} are defined as the average case execution time (ACET) and the worst case execution time (WCET) respectively and finally D_i is defined as the relative deadline. It is assumed that each task τ_i is a constrained task such that $C_i^{Hi} \leq D_i \leq T_i$. Each task τ_i must receive C_i^{Hi} within D_i or it is considered late. It is also assumed that C_i^{Hi} processor units are assigned to a task in a non-concurrent manner.

A subsystem (i.e. application) consists of a task set defined as a collection of periodic tasks $T_s = \{\tau_1, \tau_2, \ldots \tau_n\}$. A system S consists of *n* homogenous processors while a subsystem consists of *m* processors such that $1 \leq m \leq n$. Each subsystem is characterized by a *multiprocessor resource* model [2] which specifies the resource supply provided to the subsystem (also known as a clustering). The *multiprocessor periodic resource* (MPR) model is defined as (P_s, Q_s, m_i), where Q_s provides the resource budget over P_s time units to a subsystem consisting of m_i processors. Therefore, a schedulable subsystem must meet the condition $Q_s \leq mP_s$.

In uniprocessor scheduling the supply bound function *(sbf)* is used to bound the supply required for schedulability of the subsystem. Authors in [2] extended this approach for hierarchical multiprocessor frameworks for deriving schedulability conditions of the subsystem. Therefore, the supply bound function for a multicore subsystem sbf_s is defined as:

$$sbf_s(t) = \begin{cases} kQ_s + max\{0, [I - kP_s]m + Q_s\}, & t \geq P_s - \left\lceil \frac{Q_s}{m} \right\rceil \\ 0, Otherwise \end{cases} \quad (1)$$

where, $k = \left\lfloor \frac{t-(P_s-\lceil Q/m \rceil)}{P_s} \right\rfloor$ and $I = t - 2P_s + \left\lceil \frac{Q_s}{m} \right\rceil$. Additionally, a lower bound of the sbf_s has been derived for improved schedulability. The lower bound supply lsb_s function is defined as:

$$lsb_s(t) = \frac{Q_s}{P_s}\left(t - 2\left(P_s - \frac{Q_s}{m}\right)\right) \quad (2)$$

The schedule for a subsystem that generates the resource supply in a time interval of $[0,t)$ is shown in Figure 1 along with the linear lower bound function. In Figure 2 we define $\alpha = \left\lfloor \frac{Q_s}{m} \right\rfloor$ and $\beta = Q_s - m\alpha$.

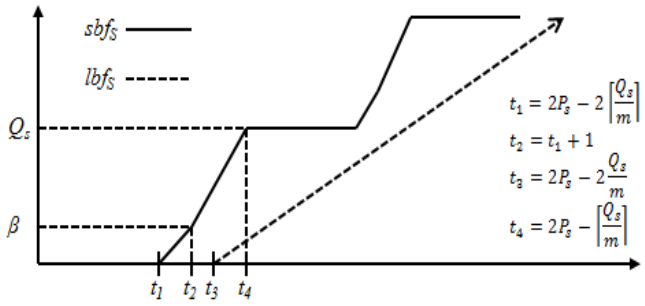

Figure 1: Supply bound and lower supply function for a subsystem

The MPR model presented by authors in [2] presents a framework that allows a subsystem exclusive access over a share of the multi-core platform. This share is then guaranteed by the sbf_s to provide a minimum resource supply to a subsystem. Therefore, HSP can utilize the MPR model to provide temporal isolation and schedulability guarantees between subsystems.

2.2. Hierarchical Scheduling

The basic framework of a hierarchically scheduled system [3] [4] for a uniprocessor platform is composed of multiple applications (subsystems) where each subsystem can be composed of a single or multiple tasks (see Figure 2). A *global* scheduler controls which subsystem is allocated the processor while the *local* scheduler determines which subsystem's task should actually execute

This two-level hierarchical scheduling approach is general enough in that it can be extended to a multiprocessor platform. In this case the scheduling of tasks within a subsystem, across m processors can be performed by the subsystem (local) scheduler while the scheduling of subsystems across the multiprocessor platform is performed by the system (global) scheduler. For example, consider a system where the overall utilization for each subsystem is $\sum_i^n \frac{C_i}{T_i}$ and $S_1 = 1.3,\ S_2 = 0.133\ and\ S_3 = 1.122$ then the overall budget is 2.5 and m = 3, then the global scheduler will provide two units of resource from two processors and the remaining 0.5 units will be provided by the third processor.

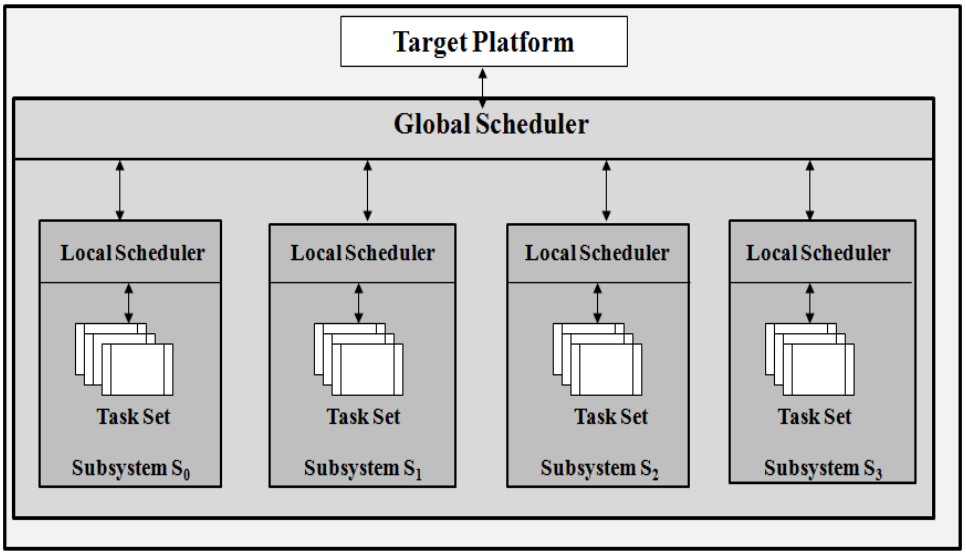

Figure 2: Hierarchical Scheduling Framework example

3. RELATED WORK

Initially an HSF was proposed by authors in [6] [8] as a means to perform composability analysis for open systems development. The motivation being that it can quickly become intractable to accurately verify the timing behavior of the embedded system as the complexity increases. The approach was to verify the timing behavior of each individual subsystem independently then compose each subsystem into the overall system.

A considerable amount of research has also been performed with hierarchical scheduling in a uniprocessor environment [4][7][9]. There has also been a fair amount of work in investigating how resources are shared across subsystems in an HSF [3][5][10]. However, there has not been a lot of work performed in actually applying a hierarchical scheduler to a multi-core environment. This lack of research is due in part to the fact that existing hierarchical scheduling algorithms are not easily extendable to multi-core environments. A couple reasons is that existing algorithms do not incorporate the inherent parallelism of a multi-core system and unfairness or task starvation can result if applied in a naïve manner.

Authors in [11] have presented a hierarchical multiprocessor algorithm known as H-SMP which was designed for a SMP-based platform. Their approach is to take a task set (i.e. an application) and assign it to the various cores in the subsystem based upon the application's level of parallelism and service requirements. Applications with higher service requirements would be allocated a higher bandwidth partition. For example, applications with soft real-time requirements would be receive a higher service level than applications with a best-effort type of service requirement. The primary limitation of this approach is that the CPU partitioning is done statically based upon a priori simulated workloads which may not represent real-world applications. In particular this static bandwidth partitioning may not achieve the best CPU partitioning for a dynamically changing workload. Another drawback is there is no explicit notion of criticality for adaptability to changing computational environments. In other words, tasks are assigned fixed budgets based upon their pre-determined WCET values where overly conservative WCET estimates could lead to system underutilization or higher task rejection rates.

-Additional work was done by authors [12][13] to provide a mixed-criticality scheduling framework for real-time operating systems (RTOS). Their approach was to use hierarchical scheduling to temporally isolate tasks of different criticality levels. A different scheduling algorithm was assigned to each criticality level. For example, tasks with the highest criticality were assigned a cyclic executive scheduler while less critical tasks were assigned other schedulers like earliest deadline first (EDF). Temporal isolation is enforced by a server with a specific budget which is statically assigned to each critically level.

There has also been some work done [14][15][16] in semi-partitioned scheduling in multiprocessors. The idea is that some tasks are assigned according to the partitioned scheduling approach while other tasks are assigned by global scheduling and therefore allowed to migrate. In order to determine how tasks are assigned the authors took a look at the task workload and then tried to assign that tasks to processors accordingly. For example, tasks with a high workload (i.e. high utilization factor $U_i = \frac{C_i}{T_i}$) would be partitioned while tasks with a low workload would be scheduled globally. Other approaches have looked at how to assign tasks to reduce cache misses [17] by using partitioned scheduling for the task most likely to generate a high number of cache invalidations. The main limitation with these approaches are that the processor assignments are done a priori with no real notion of criticality for HRT or SRT tasks to adapt to computational changes, such as task overloads.

In our work we take an adaptive approach where non-critical resources are assigned dynamically based upon environmental changes. Instead of static partitioning tasks are allocated based upon a feedback mechanism that the scheduler uses to adjust resource allocation to more effectively adapt to diverse computational workloads at run time. In order to support a service level requirement approach like H-SMP tasks are guaranteed a certain budget but are allowed to share any unused budget by employing capacity sharing mechanisms. A type of capacity sharing algorithm, known as slack stealing [24] is used which allows a lower-priority task to share the bandwidth of a higher priority task. In this way critical functions can be guaranteed a certain level of service but any unused resource can then be re-allocated to task with a lower service level thereby improving the performance, such as reduced response times, of the lower priority task.

4. HSP ALGORITHM DESCRIPTION

This section provides an overview of the HSP scheduling framework which is used to more effectively manage HRT and SRT tasks on a symmetric multiprocessing platform. Our approach employs a two-level hierarchical scheduled framework (see Figure 3) to provide resource partitioning and temporal isolation between subsystems. Additionally, HSP utilizes elements of both the partitioned and global scheduling approaches to maximize the benefits of both scheduling mechanisms.

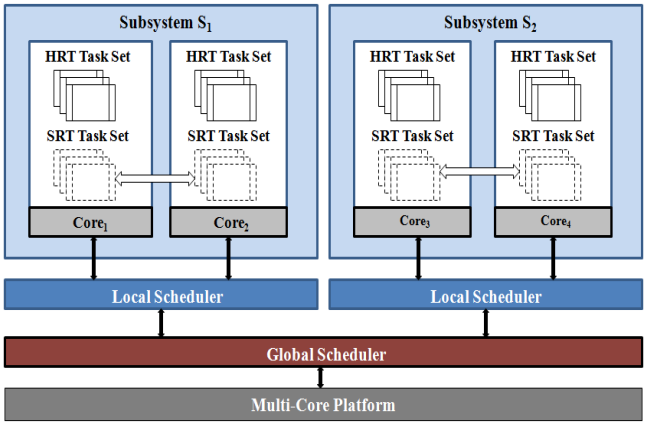

Figure 3: Hierarchical Scheduling for Multicore Processor

However, unlike uniprocessor based hierarchical scheduling SMP-based hierarchical scheduling needs to contend with tasks that can be stationary or migratory. In order to account for this added complication SMP-based hierarchical scheduling requires enhanced functionality which includes: processor assignment, task set schedulability analysis and run-time scheduling. Processor assignment is the algorithm that determines how an application is assigned to the various processors allocated by the subsystem. The tasks that comprise an application are assigned to processors based upon a combination of mixed-criticality scheduling and semi-partitioned scheduling. The schedulability analysis determines whether the HRT/SRT task set is schedulable on a specific processor. Run-time scheduling determines when tasks execute as well as manage when a task should migrate to another idle core in the subsystem.

4.1. Processor Assignment

HSP like other traditional partitioned scheduling approaches assigns each task to a particular processor based upon some type of bin-packing heuristics. HRT tasks with strict timing constraints are assigned to a specific core first according to the chosen heuristic and if the schedulability condition can be satisfied for that core. In this way HRT tasks can get the full attention of the processor and improve the deterministic behavior of the task. Consider Table 1 that defines a task set for the example Subsystem1 depicted in Figure 3. According to Table 1 tasks that are partitioned (p) are considered HRT tasks are statically assigned to a specific core and not allowed to migrate. Tasks that are global (g) are considered SRT tasks and allowed to migrate across cores in the subsystem. This is similar to mixed-criticality scheduling that assigns highly critical tasks to specific cores but allows less critical tasks to migrate.

Table 1: Example subsystem task set

Task	Core	Ti	C_i^{Lo}	C_i^{Hi}	Di
τ1	p	5	1	2	5
τ2	p	10	2	4	10
τ3	p	15	1	3	15
τ4	p	20	2	5	20
τ5	g	15	1	3	15
τ6	g	20	2	4	20
τ7	g	25	2	5	25

For the purpose of schedulability guarantees the HRT tasks are allocated a budget, by the hierarchical scheduler, equal to the task's WCET value (C_i^{Hi}), in this way tasks are guaranteed a fixed processing time by the subsystem's local scheduler. The HRT tasks are assigned to a core based upon the next-fit bin-packing heuristic and since the rate monotonic (RM) algorithm is optimal for fixed priority scheduling it is used as the determination of schedulability for partitioned tasks (see Algorithm 1). Therefore, the maximum utilization U_{si} for a core in a subsystem as defined by RM is:

$$U_{si} = \sum_{i=1}^{n} \frac{C_i^{Hi}}{T_i} \leq n\left(2^{1/n} - 1\right) \quad (3)$$

From the example task set shown in Table 1 and the multi-core system depicted in Figure 3 the HRT tasks would be assigned a particular core as illustrated in Figure 4.

Algorithm 1: HRT Task assignment algorithm

Algorithm 1 HSP HRT task processor assignment algorithm

Input: The HRT/SRT task set T_s and the processors m assigned to the subsystem S_i

Output: On each processor p_i a executable (or not schedulable) task set.

1: FOR each $\tau_i \in T_s$
2: IF τ_i is not a HRT task then
3: continue
5: $u_i = C_i/T_i$
4: Assign τ_i to processor p_i based upon u_i and next-fit bin packing heuristic
5: Let p_i^{hrt} be the set of HRT tasks assigned to processor p_i
6: ENDIF
7: Execute HSP task-splitting algorithm on processors in subsystem S_i

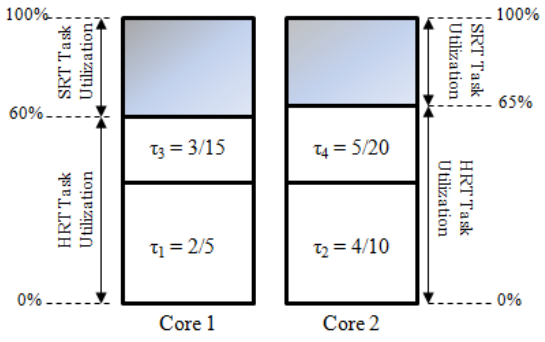

Figure 4: Partitioned task core assignments

After the HRT tasks are assigned to their respective cores the SRT tasks are assigned based upon the remaining resource capacity. If the SRT task does not fit onto a particular core to support the full execution capacity then the task is split across cores in the subsystem. Task splitting is based upon semi-partitioned scheduling which is defined as a task τ_i that is executed on l_i processors

where$l_i \geq 2$. There are l_i subtasks denoted by$\tau_i^1, \tau_i^2, \ldots, \tau_i^{l_i}$, which are synchronized where no subtasks can run in parallel and each subtask τ_i^j has a computation time C_i^j such that$C_i = \sum_{j=1}^{j-1} C_i^j$. The algorithm for splitting a task τ_i is provided in Algorithm 2.

Algorithm 2: Task splitting assignment algorithm

Algorithm 2 HSP task-splitting processor assignment algorithm

Input: The HRT/SRT task set T_s and the processors m assigned to the subsystem S_i

Output: On each processor p_i a executable (or not schedulable) task set.

```
1:  FOR each τ_i ∈ T_s
2:      IF τ_i is not a SRT task then
3:          continue
5:      find processor m* with maximum slack potential
6:      return unschedulable if U_m* ≤ U_si
7:      IF U_m* + C_i/T_i ≤ U_si then
8:          assign subtask τ_i^li to processor m*
9:      ELSE
10:         split task τ_i again where C_i^li+1 ← C_i^li – (U_si – U_m*)T_i
11:         C_i^li ← (U_si – U_m*)T_i
12:         assign subtask τ_i^li to processor m*, where U_m* ≤ U_si
13:     ENDIF
```

Consider the example provided below of how a task may be split across more than one processor. To help identify the core(s) with the maximum slack time potential for SRT task processor assignment.

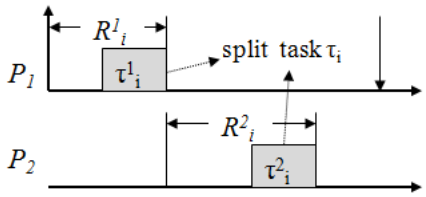

Figure 4: Split task across two processors

While Figure 5 illustrates how a split task could be split it does not describe the criteria used to assign the split tasks to the various processors in the subsystem. Traditional approaches have been to assign each share to processors with subsequent indexes so that τ_i^1 would be assigned to P_1 and τ_i^2 would be assigned to P_2. With semi-partitioned scheduling most tasks are assigned to a particular processor to reduce overhead while the remaining tasks are split to improve schedulability. The problem with this approach is there is no real notion of criticality and tasks are assigned to a processor based upon their respective WCET values which are typically conservative. Our approach with HSP is different in that task criticality is considered by assigning HRT tasks first ensuring that the tasks will be fixed to a particular processor thereby reducing runtime overhead. The schedulability is maintained for the SRT tasks by performing task-splitting and task response times are improved by taking advantage of the potential unused processing capacity, also known as slack. This slack potential is then used by HSP for processor assignment of SRT tasks. SRT tasks whether they requiring splitting or not are then assigned to available cores based upon the maximum slack potential for that core. Note that this slack

potential is determined not by the WCET of a HRT task but rather by their average execution time denoted by C_i^{Lo}. In this way the maximum potential can be identified which represents a much less conservative calculation for improving task response times. The set of algorithms for identifying slack and taking advantage of it is known as slack stealing. A brief overview of slack stealing is provided in the subsection below; for more detail readers are encouraged to review the references.

4.1.1. Slack Stealing

According to Equation (3) a task set that meets the criteria will always make its deadlines. The problem is this criterion is based upon WCET values which are usually conservative calculations and there tends to be a large gap between the WCET value and the actual processing time of a HRT task. This gap, known as slack, presents an opportunity to minimize the response times of a SRT task. Authors in [24][25] describe how the slack is found by mapping out the processor schedule of the HRT tasks over their hyper-period in a task mapping table. The table is then examined to determine the slack present between the deadline and the next invocation of the task. In turn, this table is then examined by HSP to help identify the core(s) with the maximum slack time potential for SRT task processor assignment.

4.2. Task Scheduling

The local scheduler of a subsystem in HSP is responsible for scheduling of tasks on the various cores of the subsystem. Scheduling for the HRT tasks are straightforward in that traditional scheduling mechanisms, such as RM, where the priorities of each task are assigned so that:⟦ τ⟧_4< τ_3< τ_2<τ_1. Similar to HRT tasks priorities are assigned according to the RM except SRT tasks always have a lower priority than HRT tasks, such that SRT < HRT, except during slack stealing periods. During periods of slack stealing the SRT task is temporarily promoted to the same priority level as the HRT task that finished with some available slack time. In this way another HRT task of lower priority cannot preempt a SRT task while it is stealing the slack of another HRT task.

During run-time after a HRT task completes the local scheduler looks to exploit the slack time of an HRT tasks to improve a SRT task's response time. The run-time slack of a HRT task τ_i is based upon the budget (C_i^{Hi}) of task τ_i provided by the subsystem's S_i local scheduler. The task's budget for the subsystem's S_i local scheduler of a HRT task along with the feedback from the task provides the information needed to determine if there is any potential slack available to the SRT tasks. In order to calculate the slack at some arbitrary time t we look at the unused server budget of an HRT task in the interval$[t, t + D_i(t))$. Therefore, the slack is determined by the length of that interval less than the actual unused budget available from all of the HRT tasks that fall into that interval. The slack is defined as s_i(t) = $\sum_{\forall j \in hrt(i)}(Q_j - c_j)$ that is available to any SRT task at some arbitrary time t and c_j is the actual processing time of the HRT task. As an example consider the example task set in Table 1. Figure 6 represents the tasks scheduled on the first core while Figure 7 represents the tasks scheduled on the second core. The up arrow represents task start time and the down arrow represents the task completion time.

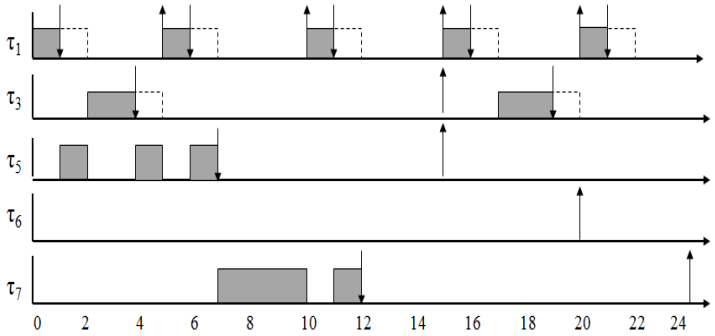

Figure 5: Core 1 task schedule

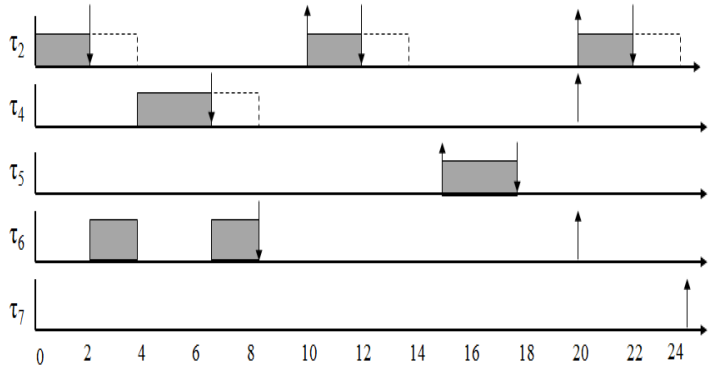

Figure 6: Core 2 task schedule

The HRT task set is statically assigned to a core and based upon the next-fit bin-packing heuristic tasks τ_1 and τ_3 are assigned core 1 while tasks τ_2 and τ_4 are assigned core 2. The highest priority SRT task τ_5 if the first task scheduled to run on either core when there is available processing or slack time. At time t1 task τ_5 is allowed to run by stealing the slack from task τ_1 but at time t2 is preempted by the HRT task τ_3. Task τ_5 is then allowed to steal slack from task τ_3 at time t4 and from task τ_1 then complete execution by time t7.

5. SCHEDULABILITY ANALYSIS

With the HSP all tasks execute up to their worst case execution time C_i^{Hi} but the local scheduler prevents the tasks from executing any further. If a task executes further than C_i^{Hi} it is considered in fault and aborted or considered overloaded and rescheduled until it is safe to be executed again. This section presents the response time analysis for HSP as it relates to partitioned and non-partitioned scheduling.

As mentioned in Section 4.1 the tasks are scheduled by a fixed priority preemptive scheduler and the task priorities are assigned according to the RM algorithm. Priority (p) is derived from the deadlines of the tasks, such that for any two tasks τ_i and τ_j their deadlines $D_i < D_j \Rightarrow p_i > p_j$. To test for schedulability, the standard Response Time Analysis (RTA) [19] [20] for uniprocessor scheduling can be extended to HSP. RTA first computes the worst-case completion time for each task (i.e. response time R_i) and then compares that value to the task deadline, such that $R_i \leq D_i$ for task τ_i. The response time value is calculated using recurrence relations:

$$R_i = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (4)$$

where *hp(i)* defines the set of tasks with a higher priority than the task τ_i. The general response time Equation (4) can then be applied to mixed critically systems [12] where the LO-criticality and HI-criticality mode schedulability can be verified. HSP can then adapt this analysis and apply it to HRT tasks which are considered HI-criticality and SRT tasks which are considered LO-criticality. Standard RTA for a uniprocessor can be applied for SRT tasks as follows:

$$R_i^{Lo} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{Lo}}{T_j} \right\rceil C_j^{Lo} \quad (5)$$

where *hp(i)* is the set of SRT tasks with a higher priority than task τ_i. The same analysis can also be applied to HRT tasks as follows:

$$R_i^{Hi} = C_i + \sum_{j \in hpH(i)} \left\lceil \frac{R_i^{Hi}}{T_j} \right\rceil C_j^{Hi} \quad (6)$$

where *hpH(i)* is the set of HRT tasks with a higher priority than task τ_i. For uniprocessor based systems the schedulability test is determined by calculating the response times of all tasks in an interval starting with a critical instant (case where all tasks experience their WCET) and comparing that to the task deadlines. However it has been shown [20] that it is a NP-hard problem when analyzing globally scheduled periodic tasks. The issue is that it is not easy to find a "representative" interval to represent the start of the critical instant. As a result, in a multicore system only sufficient results can be determined in a reasonable amount of time. Authors in [22] provide a sufficient RTA-based approach for schedulability tests for global scheduled multicore systems. The test is based upon the RTA test of Equation (4) and operates as follows:

$$R_i^{max} \leftarrow C_i + \frac{1}{m} \sum_{\tau_j \in hp(i)} \left(\left\lceil \frac{R_j^{max}}{T_j} \right\rceil C_j + C_j \right) \quad (7)$$

The schedulability analysis for semi-partitioned systems can then be derived by combing equation (4) and equation (7). To determine the schedulability for SRT and HRT tasks using average case execution time:

$$R_i^{Lo} \leftarrow C_i^{Lo} + \frac{1}{m} \left(SRT\left(\tau_i^{Lo}\right) + HRT\left(\tau_i^{Lo}\right) \right) \quad (8)$$

where $SRT\left(\tau_i^{Lo}\right)$ represents the SRT task set average execution times such that:

$$SRT\left(\tau_i^{Lo}\right) = \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{Lo}}{T_j} \right\rceil C_j^{Lo} + C_j^{Lo} \quad (9)$$

And $HRT\left(\tau_i^{Lo}\right)$ represents the HRT task set average execution times such that:

$$HRT\left(\tau_i^{Lo}\right) = \sum_{\tau_j \in hpH(i)} \left\lceil \frac{R_i^{Lo}}{T_j} \right\rceil C_j^{Lo} \quad (10)$$

where *hpH(i)* is the set of HRT tasks that are assigned to processor P_i. Additionally, to determine the schedulability for SRT and HRT tasks using worst case execution time:

$$R_i^{Hi} \leftarrow C_i^{Hi} + \frac{1}{m}\left(SRT\left(\tau_i^{Hi}\right) + HRT\left(\tau_i^{Hi}\right)\right) \quad (11)$$

$$SRT\left(\tau_i^{Hi}\right) = \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{Hi}}{T_j} \right\rceil C_j^{Hi} + C_j^{Hi} \quad (12)$$

$$HRT\left(\tau_i^{Hi}\right) = \sum_{\tau_j \in hpH(i)} \left\lceil \frac{R_i^{Hi}}{T_j} \right\rceil C_j^{Hi} \quad (13)$$

Consider the task set represented by Table 1 in Section 4.1 the schedulability analysis for both SRT and HRT would be as follows.

Table 2: Example Task Set with Response Times

Task	Core	Ti	C_i^{Lo}	C_i^{Hi}	Di	R_i^{Lo}	R_i^{Hi}
τ1	p	5	1	2	5	1	2
τ2	p	10	2	4	10	2	4
τ3	p	15	1	3	15	2	5
τ4	p	20	2	5	20	4	9
τ5	g	15	1	3	15	4	15
τ6	g	20	2	4	20	7	29
τ7	g	20	2	5	25	8	58

6. PERFORMANCE ANALYSIS

For the purpose of comparisons, we used a combined SRT/HRT periodic task set that comprised a single subsystem (i.e. application) and spanned up to *m* cores, where *m* = 2, 4, 8. Task periods (p_i) were chosen using a uniform random distribution from the list {0.25Hz, 0.5Hz, 1Hz, 2 Hz, 4Hz, 5Hz, 8Hz, 10Hz, 20Hz, 25Hz, 50Hz, 100Hz, 200Hz}. The list was created to represent some typical rates of periodic tasks. Overall system utilization (u_{sys}) for each processor ranged from [0.50, 1.00] in increments of 0.05. Individual task utilization (u_i) was randomly generated with an expected value of 0.20 and a standard deviation of 0.15. The number of tasks in the set were determined by the summation of the individual tasks where $\sum_{i=0}^{n} u_i = u_{sys}$. The execution time (c_i) was calculated based upon the task period and task utilization such that $c_i = p_i * u_i$. The HRT/SRT tasks were randomly divided from the generated task set with an expected value of $n/2$ and a standard deviation of *n-2*.

HSP was compared against four other semi-partitioning algorithms used in mixed-criticality systems, DU-RM, DU-Audsley [26], DC-RM and DC-Audsley. Each algorithm, including HSP utilizes the next-fit bin packing heuristic but differ on processor and priority assignment. The DU-RM algorithm decreasingly assigns tasks based upon the task utilization and determines feasibility based upon the RM scheduler. In other words the task with the highest utilization factor is assigned to the first available processor. DU-Audsley is similar to DU-RM except Audsley's priority assignment is optimal for a given processor but the complexity is much higher than RM assignment. The DC-RM algorithm performs processor assignment based upon the decreasing criticality of a task so HRT tasks would be assigned to a processor before a SRT task. DC-Audsley also performs processor assignment based upon the task criticality but its priority assignment is different than DC-RM. Our approach with like DC-RM and DC-Audsley assigns a task based upon criticality but differs in that if there is not enough available utilization HSP will spilt tasks across any available processors. This has the potential to significantly improve schedulability.

For the simulations we generated 10,000 task sets from the parameters described in the previous paragraph. The task sets were determined to be schedulable if every task in the set was successfully assigned to the group of cores defined by the subsystem S_i. The performance criteria for the processor assignment algorithm was determined by the success ratio of the number of tasks scheduled by the number of submitted tasks accepted, defined as follows:

$$\frac{number\ of\ scheduable\ task\ sets}{number\ of\ attempted\ task\ sets}$$

The overall subsystem utilization was determined by $u_{sys} * m$, so that 1.0, 2.0 and 4.0 represents 50% utilization for $m = 2, 4, 8$ respectively. The data in Figures 8, 9 and 10 illustrates the results from u_{sys} = [0.5, 1.0] where HSP clearly provides better schedulability than the other processor assignment algorithms. Note that the other algorithms start to report failure around 0.5 to 0.7 while HSP does not start to report failure until close to 0.7 to 0.8. This coincides with other work [21][23] that states maximum schedulability for RM or DM is about 88% for uniprocessors. Also notice that HSP outperforms the other algorithms as the number of cores increase because this provides HSP the opportunity to share more of the computation across the various cores in the subsystem.

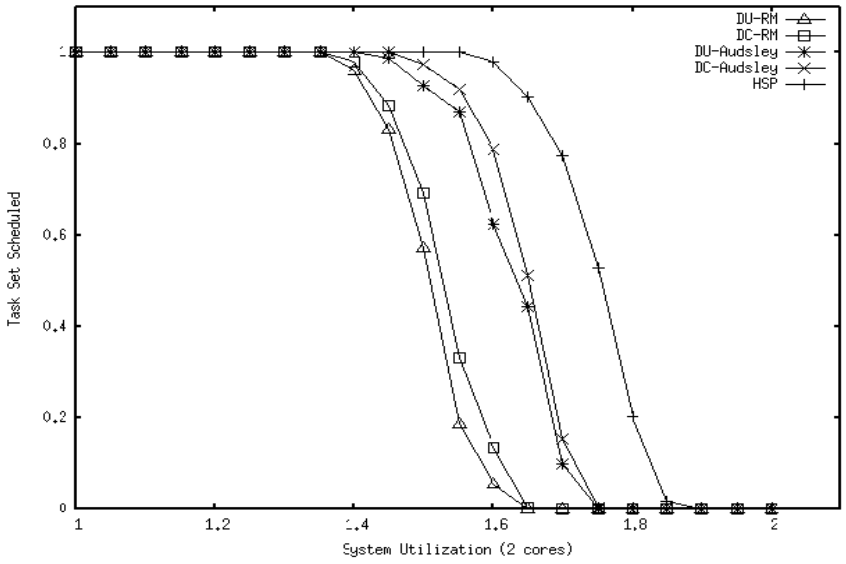

Figure 7: Task Set simulation 2 cores

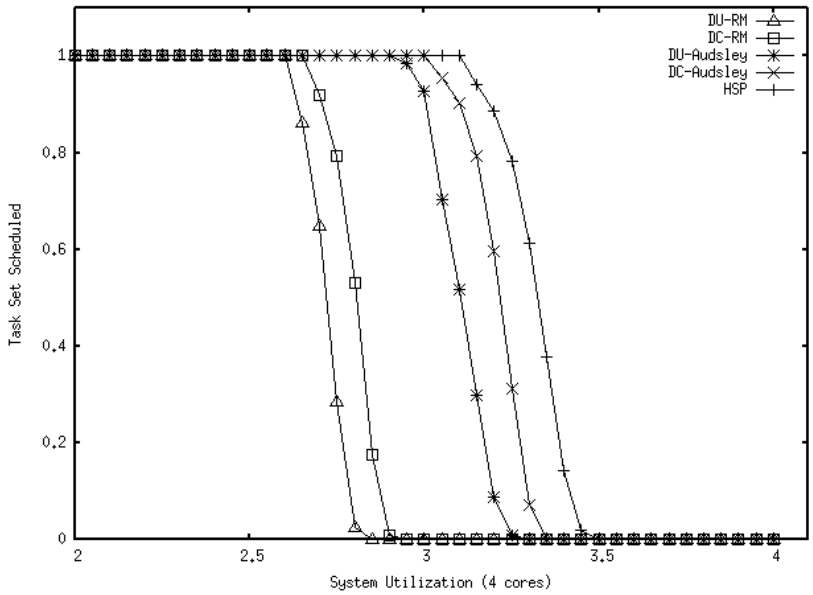

Figure 8: Task Set simulation 4 cores

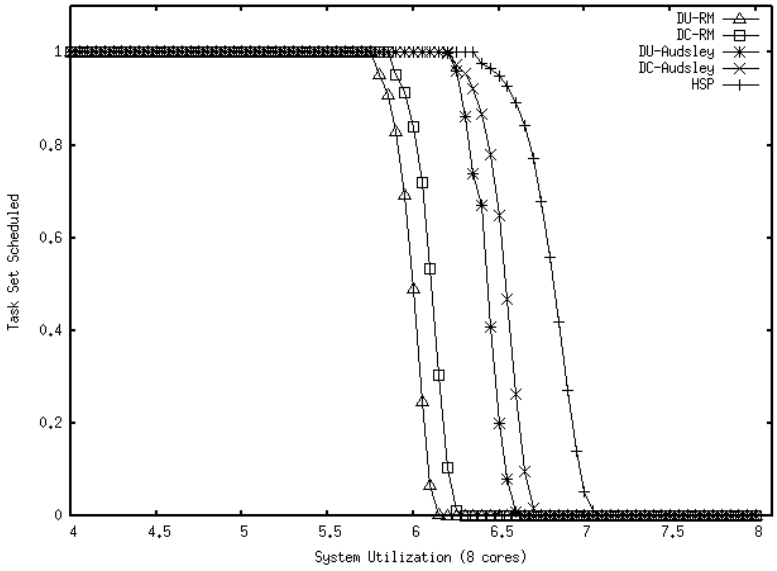

Figure 9: Task Set simulation 8 cores

7. IMPLEMENTATION

This section defines the design and implementation of HSP in the VxWorks real-time operating system (RTOS). The work is based upon the architecture presented in [27] and extended to work in a SMP-based platform.

7.1. Local Scheduler Implementation

The native VxWorks scheduler can schedule tasks using either a preemptive priority based or a round-robin scheduling policy. In VxWorks 6.x and greater Wind River introduced the concept of real-time processes (RTP) which more closely resemble processes in general purpose operating

systems like Linux. Tasks in kernel mode or processes in RTP mode are scheduled in the same way. Processes are created with memory protection so kernel memory space, ISRs and direct hardware access are prohibited. Tasks that operate in kernel mode have full access to kernel resources and are not subject to the same limitations as processes in RTP mode

We choose to implement HSP in kernel mode because the overhead in RTPs are prohibitive and HSP needs access to the kernel resources for task management. HSP was implemented on top of the native VxWorks scheduler as a type of extension or middleware that sits between the hierarchical scheduler and the VxWorks native scheduler. The VxWorks RTOS provides functions to extend the capability so various kernel mechanism can be customized to support HSP. For example, the scheduler can be extended with either a customized ready queue structure or to attach an interrupt handler that is executed at every clock tick.

The native VxWorks scheduler dispatches the highest priority task in the ready queue. Our approach utilizes the system call *tickAnnounceHookAdd( )* that is invoked at every tick interrupt and called before the native scheduler accesses the ready queue to dispatch the highest priority task. The ready queue is then manipulated by resuming a task *taskResume( )*, suspending a task *taskSuspend( )* or setting/changing priorities *taskPrioritySet( )*. The kernel's tick counter is also utilized to read *tickGet( )* and set *tickSet( )* as a means to manage the notion of time when the tick interrupt ISR is invoked.

The primary function of the local scheduler is to arrange tasks in the ready queue at every period start, in effect extend the VxWorks scheduler to support periodic tasks. The local scheduler is implemented as part of a custom ISR that is attached with the *tickAnnounceAdd( )* system call. The system call routines mentioned previously are then called to change the status of the task or to change task priorities. The native VxWorks scheduler is then invoked to perform the necessary functions (i.e. context switching) to dispatch the task on the appropriate processor. The pseudo code listed in Algorithm 3 below provides an overview of the local scheduler.

Algorithm 3: Local scheduler algorithm

Algorithm 2 HSP Local scheduler algorithm

```
1:  FOR each de_i ∈ DE_i
2:      IF DEQ[i]. task = ready THEN
3:          logMsg(deadline_miss, DEQ[i].task)
4:      END IF
5:      updateEventQueue(DEQ[i].task)
6:  ENDFOR
7:  FOR each pe_i ∈ PE_i
8:      insertReadyTask(PEQ[i].task)
9:      updateEventQueue(PEQ[i].task)
10: END FOR
11: event = getNextEvent(DEQ, PEQ)
12: expire = event – systemTime
13: setInterrupt(expire, localSchedIsr)
14: systemTime = event
```

The first step of the algorithm is to check if the task is still in the ready queue (lines 2-4) the then the deadline event queue (DEQ) is updated (line 5) to track the task deadlines. At each period start tasks are inserted into the ready queue (7-8). Tasks deadlines and periods are updated in the

periodic event queue (PEQ). The next event is then updated by extracting the closet deadline/period from event queue (lines 11-12). The interrupt is set at the next event and the local system counter is updated (lines 12-14).

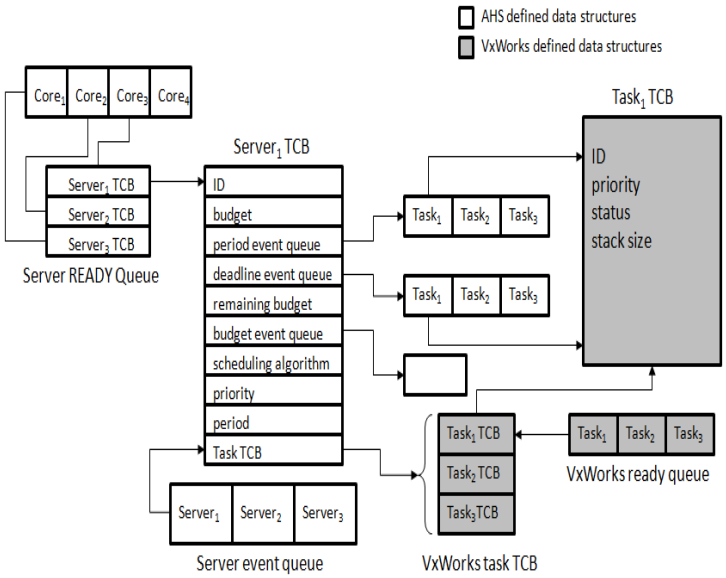

Figure 10: HSP Implementation in VxWorks

7.2. Global Scheduler Implementation

Global scheduling is used to implement the notion of servers in a hierarchical scheduled system. The global scheduler is responsible for managing all the events in the system which can include subsystem events, server events and server budget events. The global scheduler itself is a task in VxWorks with its own task control block (TCB) and task event queue. Figure 11 below illustrates the implementation of the required data structures to support global scheduling in HSP for VxWorks.

The TCBs needed to support global scheduling in VxWorks are described in the list below.
ID is a unique number associated with each server.

period_event_queue is a reference to the server's event queue which contains the task period.
period is the period of the server.

deadline_event_queue is a reference to the server's task queue which holds the task deadline.
budget is the server defined budget.

remaining_budget is the current remaining budget of the server.

priority is the server's priority.

scheduling algorithm is the server's local scheduling algorithm.

Task_TCB is a list to the VxWorks TCB task list. It references those task TCB's that are associated with the server.

7.3. Hardware Platform

HSP was implemented as described in the previous section with VxWorks 6.9 on a Freescale T4240: QorIQ 12 core (24 virtual-core) communications processor.

For evaluation purposes we ported the SNU Real-Time Benchmark Suite [18] and compared response times and overall system utilization using partitioned, non-partitioned (global) and hierarchical scheduling. The SNU real-time benchmark suite contains small C programs used for worst-case execution time analysis. This benchmark was chosen because it is completely structured (no unconditional jumps, no loop body exits,), no switch or do-while statements and no library calls or specific systems calls. The programs are mostly numeric and DSP algorithms.
In order to represent the periodic task model of an embedded system a subset of the programs in the benchmark suite were chosen and assigned arbitrary task rates (see Table 3).

Table 3: Simulated Periodic Task Set

C Program	Task	Rate	C_i^{Lo}	C_i^{Hi}
matmul	τ_1	50Hz	1.7ms	5.1ms
fft1	τ_2	40Hz	2.7ms	5.4ms
fir	τ_3	20Hz	10.4ms	20.8ms
lms	τ_4	10Hz	12.6ms	25.2ms
ludcmp	τ_5	40Hz	6.8ms	13.6ms
minver	τ_6	10Hz	3.5ms	10.5ms
qsort-exam	τ_7	5Hz	2.2ms	11.0ms

The tasks sets were assigned as HRT = $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ and SRT = $\{\tau_5, \tau_6, \tau_7\}$. The HRT/SRT task sets comprised a single subsystem S_1 which was allocated two cores in the hierarchical system. The HRT/SRT task sets were conceived so that if the C_i^{Hi} value for each SRT task was realized then the task set is not schedulable and an overload condition would result. In order to evaluate the effectiveness of HSP the execution times of the overall task sets were increased from [0.00, 1.00], where 0.0 indicates all tasks are executed at their respective C_i^{Lo} levels and 1.0 indicates all tasks are executed at their respective C_i^{Hi} levels. The task response times were measured by the high resolution counter/timer used as part of the timestamp mechanism by WindRiver's System Viewer application. Table 3 was used to represent their respective average case and worst case execution times for each task in the set.

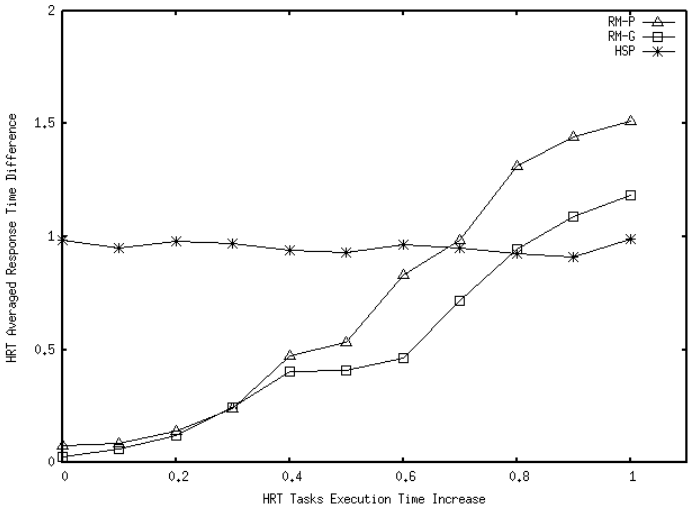

Figure 11: HRT Task Set Response Time Average

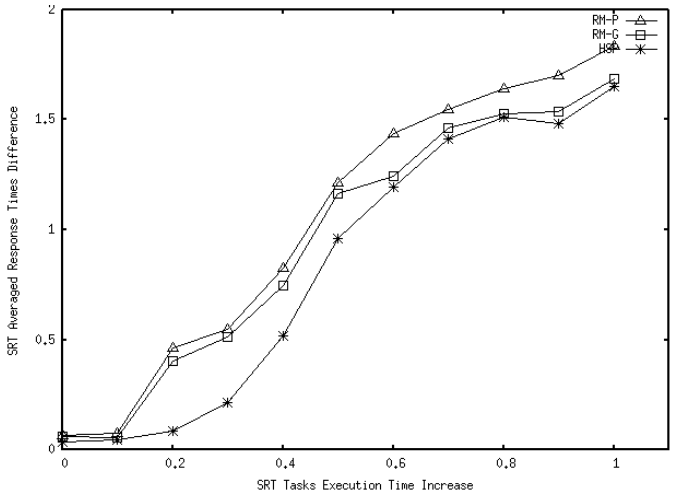

Figure 12: SRT Task Set Response Time Average

Figure 12 represents the measured response times of the HRT task set. To represent each individual task would create an overly crowded graph so the individual task response times were normalized and then averaged over the whole task set. Specifically each task response time was recorded then compared to the respective task's estimated response time. Let the actual task response time be defined as R_i^{Act}, the estimated lower bound response time is R_i^{Lo}, the upper bound response times is R_i^{Hi} so that the averaged response time difference is defined as:

$$\Delta i = \frac{\left(R_i^{Hi} - R_{Hi}^{Lo}\right) - R_i^{Act}}{\left(R_i^{Hi} - R_i^{Lo}\right)}$$

then the total task set response time average is defined as the average of all Δ_i for the HRT task set. What this means is a value of 0.0 indicates the measured task response times were at or near their respective R_i^{Lo} values and a value of 1.0 indicates R_i^{Hi} values. A value greater than 1.0 signifies that one or more tasks exceeded their deadline. Notice that for HSP the response time difference hover around 1.0 this is because the local scheduler does not allow other HRT tasks to execute before a higher priority task C_i^{Hi} execution time. Therefore, before the system starts to

become overloaded around 0.6 the response times for both the partitioned method (RM-P) and the non-partitioned method (RM-G) outperform those of HSP. Recall, this is an acceptable situation because with HRT tasks we are less concerned about response times as we are with HRT timing constraints. Note, that at times 0.6 to 0.7 both RM-P and RM-G methods start to exceed 1.0 which indicates that tasks in the HRT set are beginning to experience deadline misses while with HSP no HRT tasks experience deadline misses.

The SRT task set performance is illustrated in Figure 13. Notice that early on before the system becomes overloaded from 0.0 to 0.4 HSP clearly outperforms both the RM-P and RM-G methods. This is because the HSP is able to take advantage of the slack generated by the HRT task set. Once the system starts to become overloaded at 0.5 HSP starts to converge to RM-G because there is no longer any available slack time. Both the RM-G and the HSP methods outperform RM-P because they are allowed to migrate across the cores in the subsystem.

8. CONCLUSIONS/FUTURE WORK

In this paper we considered the problem of how to assign and schedule HRT and SRT tasks in a symmetric multiprocessor environment to more effectively adapt to environmental changes. Those changes such as unexpected computational workload deviation were managed by hierarchical scheduling to provide the temporal isolation between tasks. The efficient assigning and scheduling of processors was accomplished by combining mixed-criticality and semi-partitioned scheduling. The result was demonstrated improvement of response times for SRT tasks and schedulability guarantees for HRT tasks where no deadlines were missed during periods of overload. As further confirmation for the validity of this approach we also implemented HSP as part of the VxWorks RTOS.

Future work includes evaluating the additional overhead HSP incurs in VxWorks as compared to traditional scheduling. Additionally, tasks as well as task sets are considered to be completely independent with no shared resources. A more practical implementation would include HSP scheduled tasks or subsystems that would have to share a mutual resource such as a semaphore.

REFERENCES

[1] J. Carpenter, S. Frank, P. Holman, A. Srinivasan, J. Anderson and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. Handbook of Scheduling: Algorithms, Models and Performance Analysis. CRC Press LLC, 2003.

[2] I. Shin; A. Easwaran, I. Lee. Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors. Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on, vol., no., pp.181,190, 2-4 July 2008.

[3] R.I. Davis and A. Burns. Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems. In RTSS'06.

[4] P. Goyal, X. Guo and H.M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In OSDI, pp. 107-121, 1996

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

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

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

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

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

[10] M. Behnam, T. Nolte, M Sjodin and I Shin. SIRAP: A synchronization protocol for hierarchical resource sharing real-time open systems. In Proc. 7th ACM and IEEE Int. Conf. Embedded Software (EM-SOFT 07).
[11] A. Chandra, P. Shenoy. Hierarchical Scheduling for Symmetric Multiprocessor. In IEEE Trans. on Parallel and Distributed Systems. 2013.
[12] M. Mollison, J. Erickson, J. Anderson, S. Baruah, J. Scoredos. Mixed-Criticality Real-Time Scheduling for Multicore System. IEEE (CIT 2010).
[13] J. Herman, C. Kenna, M. Mollison, J. Anderson. D. Johnson, RTOS Support for Multicore Mixed-Criticality Systems. (RTAS 2012)
[14] O. Kelly, H. Aydin, B. Zhao, On Partitioned Scheduling of Fixed-Priority Mixed Criticality Task Set. (TrustCom 2011)
[15] S. Kato, N. Yamasaki. Semi-Partitioned Fixed-Priority Scheduling on Multiprocessor. (RTAS 2009).
[16] S. Kato, N. Yamasaki, Y. Ishikawa. Semi-Partitioned Scheduling of Sporadic Task Systems on Mulitprocessor. (ECRTS 2009)
[17] B. Andersson, J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. (RTSA 2000)
[18] SNU Real-Time Benchmark, http://www.cprover.org
[19] M. Joseph and P. Pandya. Finding response times in a real-time systems. BCS Computer Journal pp390-396, 2009
[20] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. Software Engineering Journal, pp284-292, 1993
[21] L. Papalau, P. Samalik. Design of an Efficient Resource Kernel for Consumer Devices, Stan Ackermans Institute, Eindhoven University if Technology, Eindhoven, Holland 2000.
[22] M. Bertogna, M. Cirinei. Response-Time Analysis for globally scheduled Symmetric Multiprocessor Platforms, RTSS 2007.
[23] J. Lehoczky, L. Sha, Y. Ding, The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior, IEEE Real-Time Systems Symp. 1989.
[24] R. Davis, K. Tindell, A. Burns, Scheduling Slack Tine in Fixed-Priority Pre-emptive Systems. In Proc. Real-Time Systems Symp. 1993.
[25] U., José M., J. Orozco, and R. Cayssials. Fast Slack Stealing methods for Embedded Real Time Systems. 26th IEEE International Real-Time Systems Symposium (RTSS 2005)-Work In Progress Session. 2005.
[26] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report, The University of York, 1991.
[27] M. Behnam, T. Nolte, I. Shin, M. Asberg. Towards Hierarchical Schedling in VxWorks. OSPERT 2008, Proc. of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications. 2008