Implementation of the Multi-Level Adaptive Hierarchical Scheduling Framework

Nima Moghaddami Khalilzad, Moris Behnam, Thomas Nolte
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden
nima.m.khalilzad@mdh.se

***Abstract*—We have presented a multi-level adaptive hierarchical scheduling framework in our previous work. The framework targets compositional real-time systems which are composed of both hard and soft real-time systems. While static CPU portions are reserved for hard real-time components, the CPU portions of soft real-time components are adjusted during run-time. In this paper, we present the implementation details of our framework which is implemented as a Linux kernel loadable module. In addition, we present a case-study to evaluate the performance and the overhead of our framework.**

I. Introduction

Hierarchical scheduling and resource reservation techniques have been widely used for composing independent hard real-time systems on a shared underlying hardware [1]. Using such techniques, the timing behavior of the individual systems (components) are studied in isolation, while the correctness of the entire systems is inferred from the correctness of the individual components before the composition. This compositional timing study is especially useful in open systems in which components are added or removed during the system's life time [2].

Hierarchical scheduling is often performed through CPU reservations. When dealing with hard real-time systems, based on the worst case CPU demand of the individual components, a CPU portion is reserved for each component such that the component's inner tasks are guaranteed to receive enough CPU resource time to complete their executions in time.

While there exists a variety of techniques to handle the composition when composing hard real-time systems (e.g., [2], [3], [4], [5]), the problem of composing soft and hard real-time systems together has not been deeply studied. A considerable group of soft real-time systems have the following attributes. First of all, they demonstrate a wide difference between their worst case and average case CPU demands. Secondly, occasional timing violations can be tolerated in these type of systems. Last but not least, resource demand analysis are rarely done for these systems. As a result, designers do not have enough information about the resource demand requirements of such systems. Note that the timing requirements are often known a priori, whereas, the resource requirements are unknown. In dealing with real-time systems that have the aforementioned attributes, reserving the CPU portions based on the worst case CPU demand of the tasks is not an efficient design approach. Because even if the worst case CPU demand is available, it will result in an unnecessary CPU overallocation. Consequently, the CPU resource will be wasted.

To address this problem, we presented an adaptive framework in [6] where for hard real-time systems we reserve the CPU portions based on their worst case resource demand. On the other hand, soft real-time systems receive dynamic CPU portions based on their actual need at each point in time. For acquiring the resource demand of the soft real-time components, we monitor their behavior during run-time and use the gathered information to adjust the component CPU reservations. While the design and evaluation of our framework is presented in [6], in this paper we present the implementation details of our **Ad**aptive **Hier**archical **Sched**uling (`AdHierSched`) framework[1]. In particular we present the following contributions in this paper.

- The data structures and the mechanisms that are used in the implementation of our framework as a Linux kernel loadable module.
- The performance evaluation of `AdHierSched` with respect to timing requirements of the real-time components.
- The overhead evaluation of the scheduler and the CPU reservation adapter component.

The rest of the paper is organized as follows. The related work is reviewed in Section II. In Section III we present our system model. Section IV presents the data structures and implementation techniques used in `AdHierSched`. We evaluate the performance of `AdHierSched` in Section V. The overhead of `AdHierSched` is presented in Section VI. Finally, the paper is concluded in Section VII.

II. Related work

There exists an enormous number of papers that address the implementation of real-time schedulers (e.g, RTLinux [7] and RTAI [8]). However, in this paper we only review a part of

[1]The source code is available at:
http://www.idt.mdh.se/~adhiersched.

them that focus either on hierarchical scheduling or on adaptive scheduling.

In [9] hierarchical scheduling is done on top of the VxWorks operating system. Hierarchical scheduling on top of the FreeRTOS operating system is presented in [10]. ExSched [11] is a platform independent real-time scheduler which has a hierarchical scheduling plug-in. Hierarchical scheduling is also implemented in μC/OS-II [12]. All of these works are two-level hierarchical schedulers and are designed for hard real-time applications, i.e, they are not adaptive frameworks.

HLS [13] is a multi-level hierarchal scheduling implemented in Windows 2000 which targets composing soft real-time systems. In [14], Parmer and West presented a hierarchical scheme for managing CPU, memory and I/O. These frameworks are not adaptive in the sense that the resource demands are not monitored and hence the resource reservations (if used) are fixed during run-time.

Hierarchical scheduling is also used for virtualization purposes. Recursive virtual machines are proposed in [15] where each virtual machine can directly access the microkernel. A two-level hierarchical scheduler using L4/Fiasco as the hypervisor is presented in [16]. Lee *et al.* developed a virtualization platform using the Xen hypervisor [17]. A Virtual CPU scheduling framework in the Quest operating system is developed by Danish *et al.* [18]. In [19], the CPU reservations are used for scheduling virtual machines. The VirtualBox and the KVM hypervisores are scheduled using CPU reservation techniques in [20].

The AQuoSA framework [21] is an adaptive framework implemented in Linux which uses the CPU reservation techniques together with feedback loops to adjust the reservations during run-time. Our work is different than AQuoSA in the following two aspects. First of all, we target multi-level hierarchical systems while AQuoSA only targets flat systems, i.e the systems with one task per CPU reservation and without local schedulers. Secondly, we have implemented `AdHierSched` as a kernel loadable module, whereas, AQuoSA requires kernel patching.

ACTORS [22] is an adaptive framework which targets multicore systems. In this framework, the CPU reservations are used for providing isolation among real-time tasks, while the reservation sizes are being adjusted during run-time. ACTORS uses `SCHED_DEADLINE` [23] for implementing the CPU reservations. Similar to the AQuoSA framework, ACTORS addresses flat systems and not hierarchical systems.

AIRS [24] is a framework designed to provide high quality of service to interactive real-time applications. AIRS uses a new CPU reservation scheme as well as a new multiprocessor scheduling policy. Alike AQuoSA and ACTORS, AIRS targets flat systems.

III. Model

In this section, we explain how we model servers, tasks and systems.

A. Server model

We use the periodic resource model [25] in our framework, and we implement the periodic model using the periodic servers which work as follows. The servers are released periodically, providing their children with a predefined amount of the CPU time in each period. The periodic servers idle their CPU allocation if there is no active task/server inside them. Any server implementation compliant with the periodic resource model can be used in our framework. A periodic server is represented with the following 4-tuple $S_i^j =< P_i^j, B_i^j, Pr_i^j, \zeta_i^j >$, where P_i^j, B_i^j, Pr_i^j and ζ_i^j represent period, budget, scheduling priority and importance of server S_i^j. The importance value represents the relative importance of the servers with respect to their other sibling servers. This parameter is only used in an overload situation where the total CPU demand is more than the available CPU. In the overload situations, `AdHierSched` prioritizes the servers in the order of their importance. Note that the overload situation is only considered to happen in soft real-time servers. The superscript in the server notation, represents the parent server index. Based on the scheduling policy of S_j, a subset of the parameters in the server 4-tuple are used. For instance, when the scheduling is done according to EDF, Pr_i^j is ignored. In the case of soft real-time servers, B_i^j is adapted during run-time. Thus, the budget is a function of time $B_i^j(t)$. Consequently, server's children may receive a different share of the CPU in different server periods. The adaptation is done through the budget controller component based on on-line monitoring of the server's workload. The budget controller adapts the budgets such that each server receives just enough CPU time at each server period. The details of the adaptation mechanism is presented in our previous work [6].

Furthermore, server S_i^j is composed of n_i child servers and m_i child tasks. Server S_i^j schedules its children according to its local scheduling policy. Servers and tasks inherit the type of their parents, e.g., if a server is a soft-real-time server its children will also be treated as soft real-time servers/tasks. At each point in time there is at most one server assigned to the CPU which is called the “active server”.

Since our adaptation mechanism is designed for the periodic servers, we focus on the explanation of the periodic servers in this paper. Nevertheless, the Constant Bandwidth Server (CBS) [26] is implemented in `AdHierSched`, and it can be used inside the periodic servers for providing timing isolation among tasks and servers that reside inside the same periodic server parent.

B. Task model

We assume the periodic task model in which a periodic task τ_i^j, which is a child of server S_j, is represented using the following parameters: task period T_i^j, task deadline D_i^j, worst case execution time C_i^j and task priority π_i^j. Similar to the server model, depending on the parent scheduling policy some task parameters may be ignored. At each point in time, at most one task is assigned to the CPU which is called the “running task”. In the case of soft real-time tasks, we assume that the

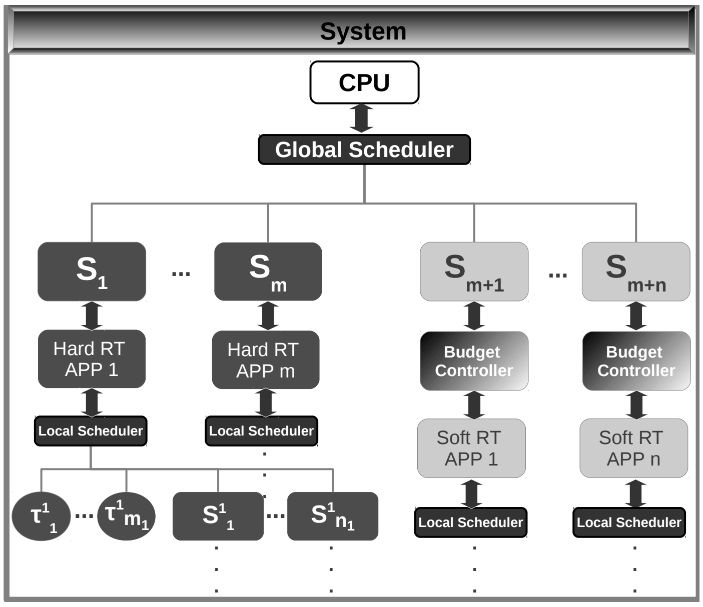

Fig. 1: Visualization of the system model.

tasks are dynamic, i.e, their execution times are changing in a wide range during run-time and their execution time is not known a priori. One instance of the task execution is called a job.

C. System model

We assume a single processor system which consists of n soft real-time servers and m hard real-time servers at the root level. The servers contain applications which consist of tasks and/or sub-servers. Therefore, our system model is a multi-level hierarchical model. A system has a global scheduler which schedules the servers and tasks at the root level of the hierarchy. In addition, there is a local scheduler in each server which is responsible for scheduling the server's inner children (both tasks and servers). Figure 1 illustrates our hierarchical system model. The hard real-time applications are shown using dark gray background in the figure. There is a budget controller component attached to the soft real-time servers. The budget controller component monitors the CPU demand of the applications and assigns a sufficient CPU portion to the servers.

Considering the system hierarchy, we would like to present two definitions that are used in the later sections of the paper for explaining the implementation of the scheduler.

Definition 1: S_i^j is an **ancestor** of S_κ^l if either $i = l$ or by upward traversing the parent of S_l we reach S_i^j. For instance, S_1 is an ancestor of S_5^3 in Figure 2.

Definition 2: S_i^j **outranks** S_κ^l if and only if one of an ancestor of S_l is S_j. For instance, S_2 outranks S_3^1 in Figure 2.

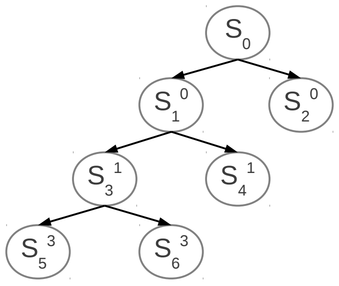

Fig. 2: Example tree structure (S_0 represents the root scheduler).

IV. Framework

In this section, we explain how our assumed model is implemented as a Linux kernel loadable module.

A. Real-time scheduling through a kernel loadable module

We use a similar idea to the work presented in [11]. The idea is to implement a real-time scheduler in Linux without modifying the kernel. To this end, we developed a kernel loadable module that plays a middleware role between real-time tasks and the Linux kernel. The module is responsible to release, run and stop the real-time tasks. When a task has to run, the module inserts it into the Linux run queue and changes its state to running. On the other hand, when the module has to stop a real-time task, it removes the task from the Linux run queue and the task goes to the sleep state. Thus, at each point in time, there is at most one real-time task (priority 0 to 99) in the Linux run queue. Consequently, no matter which Linux real-time scheduling class is used, the `schedule()` system call will always pick the single real-time task that is in the Linux run queue. Figure 3 illustrates the relation between the `AdHierSched` module and the Linux run queue.

B. Managing time

In order to manage the scheduling events, we use the classic Linux timers (low-resolution timers) available in `kernel/timer.c`. As will be explained in the rest of this section, we use one timer per task and two timers per server for managing their corresponding scheduling events. Therefore, `AdHierSched` does not have a release queue and instead it delegates the job of the release queue to the Linux timer list. Since the Linux timer list is implemented using the red-black trees, when the number of timers increases, retrieving and inserting them are still efficient $(O(logn))$. Nevertheless, we assume that systems will not excess a handful of levels, hence n will not be a large number. We insert the timers using the `setup_timer_on_stack` and `mod_timer` system calls, and remove them using the `del_timer` system call.

In order to convert the relative scheduling parameters to absolute parameters, we use the `jiffies` variable available in the kernel which return the current time.

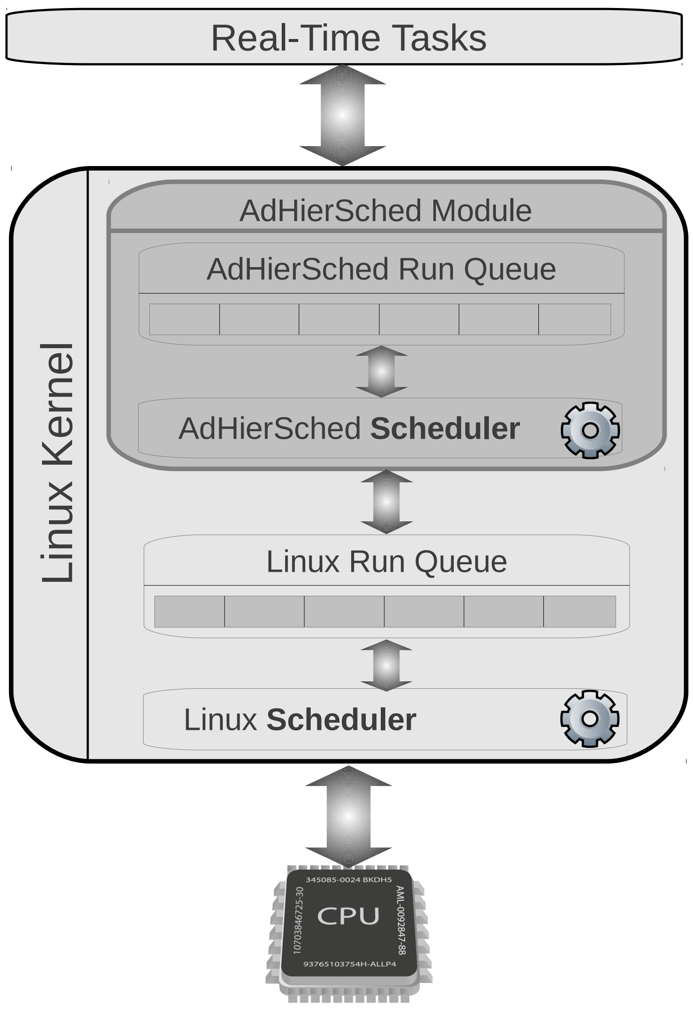

Fig. 3: `AdHierSched` module.

C. Task and server descriptors

`AdHierSched` uses its own task and server descriptors to store their corresponding information. The task descriptor, which is presented in Code Snippet 1, has a `timer_list` member called `period_timer` (line 17) which is used for running tasks periodically. When a task finishes its job, the `period_timer` is set to the next release of the task. Each `AdHierSched` task points to a Linux task (line 16). Tasks may be attached to a periodic server (line 18) and/or a CBS (line 19). The CBSs may be used for providing timing isolation among tasks that reside inside the same parent, i.e periodic server. The `dl_miss` member in line 7 and the `dl_miss_amount` member in line 15 are used to monitor the load situation of the tasks. These fields are used by the budget controller component to adapt the budget of servers. The `timestamp` member is used for monitoring the duration of the scheduling events such as the duration that tasks are assigned to the CPU.

The server descriptor is presented in Code Snippet 2. Our framework supports both constant bandwidth servers

Code Snippet 1: Task descriptor.

```
 1: struct Task {
 2: struct list_head head;
 3: int id;
 4: int priority;
 5: int state;
 6: int cnt; /* job number */
 7: int dl_miss; /* number of deadline misses */
 8: int missing_dl_flag;
 9: unsigned long period;
10: unsigned long release_time;
11: unsigned long exec_time;
12: unsigned long relative_deadline;
13: unsigned long abs_deadline;
14: unsigned long timestamp;
15: unsigned long dl_miss_amount;
16: struct task_struct *linux_task;
17: struct timer_list period_timer;
18: struct Server *parent;
19: struct Server *cbs; };
```

and periodic servers. The `type` member in line 4 indicates whether the server is a constant bandwidth server or a periodic server. The `children` member in line 3 is a pointer to the scheduling entities that are inside the server. The `control_period` field in line 8 stores the budget adaptation frequency. The `current_budget` field stores the remaining budget at each time point. The fields from line 15 to line 17 are used for the budget adaptation purpose. Each server has its own ready queue (line 19) which contains the child tasks and servers that are ready to run. The server structure has two `timer_list` members: `period_timer` and `budget_timer`. The `period_timer` is used for periodically releasing the servers, whereas, the `budget_timer` is used to stop the servers when their budget is depleted.

D. Timer handlers

There are two types of handlers: (i) release handlers (ii) budget depletion handlers. We have implemented each handler in a separate function. The list of timer handlers is as follows. (i) Task release, (ii) Server release, (iii) Periodic server budget depletion, (iv) CBS budget depletion.

E. Queue structure

We define the "scheduling entity" type as a generic type which covers both tasks and servers. Therefore, an entity can be either a task or a server. The ready queues store the scheduling entities in the order of their priority. The ready queue is implemented as a linked list through the `list_head` structure available in the Linux kernel. We have implemented two functions for inserting/deleting an entity to/from queue:

- `insert_queue(queue, entity)`
- `delete_queue(entity)`

Code Snippet 2: Server descriptor.

```
struct Server {
struct list_head head;
Children children;
int type;
int id;
int priority;
int cnt; /* number of jobs */
int control_period;
int importance; /* ζ*/
unsigned long budget;
unsigned long period;
unsigned long relative_deadline;
unsigned long abs_deadline;
unsigned long current_budget;
unsigned long consumed_budget;
unsigned long extra_req_budget;
unsigned long total_budget;
unsigned long timestamp;
struct Queue *ready_queue;
struct timer_list period_timer;
struct timer_list budget_timer;
struct Server *parent; };
```

1) Fixed priority scheduling: When the scheduling policy is fixed priority, the `insert_queue` function inserts the new entities based on their priorities.

2) EDF scheduling: The insertion to the ready queue, when the scheduling policy is EDF is based on the `abs_deadline` of the scheduling entities.

Note that since we use multiple ready queues (one queue per server):

$$q_i^j \leq n_i^j + m_i^j,$$

where q_i^j is the number of elements in the ready queue of S_i^j. Hence, the complexity of the insertion to the queue is $O(q_i^j)$.

F. Communication between tasks and `AdHierSched`

The communication between tasks and `AdHierSched` is done through a device file. The `AdHierSched` library provides a number of API functions. The API functions use the `ioctl()` system call for the communication purpose. When the message is delivered to the `AdHierSched` module, it relays the message to the message's corresponding function. The list of provided API functions is presented in Table I. The names of the functions are self explanatory, however, we explain a few of them here. As soon as `AdHierSched` receives a `run()` message, it releases all of the servers and tasks immediately. So, the release time of all scheduling entities will be equal if this function is used. The `stop()` function first stops inserting new timers to the timer list, i.e, it stops the release events. Secondly, it calls the `wake_up_process()` system call for all of the tasks that are still running. In other words, when the `stop()` function is called, the `AdHierSched` module no longer operates and Linux takes the complete responsibility of scheduling the real-time tasks. The `task_finish_job(task_id)` function should be called at the end of the task jobs. This call indeed changes the task status to sleep until the next release of the task. Note that it is possible to add/remove tasks and servers through the API functions while the module is running.

run()
stop()
create_task()
detach_task(task_id)
release_task(task_id)
task_finish_job(task_id)
detach_server(server_id)
release_server(server_id)
attach_task_to_mod(task_id)
create_server(queue_type, server_type)
attach_server_to_server(server_id, server_id2)
attach_task_to_server(server_id, task_id, server_type)
set_task_param(task_id, period, deadline, exec_time, priority)
set_server_param(server_id, period, deadline, budget, priority, server_type)

TABLE I: List of provided API functions by `AdHierSched` library.

G. Configuration and run

The API functions allow the users to configure their target system, i.e, to create their desirable hierarchy and to set the scheduling parameters. Once the system is set up, the Linux tasks need to be attached to the `AdHierSched` tasks using the `attach_task_to_mod(task_id)` API function. A sample task structure is presented in Code Snippet 3. Finally, the `run()` API function needs to be called to release all servers and tasks. When `AdHierSched` receives a `run()` call, it releases all servers and tasks and then tries to run them. Depending on the global level scheduling policy, among all released scheduling entities at the root level of the hierarchy, the one that has the highest priority or shortest deadline will be assigned to the CPU.

Code Snippet 3: Sample task structure.

```
int main(int argc, char* argv[]){
task_id = atoi(argv[1]);
attach_task_to_mod(task_id);
while $i < job\_no$ do
    /* periodic job */
    task_finish_job(task_id);
end while
detach_task(task_id);
return 0; }
```

If a server is assigned to the CPU, it will try to run its local ready queue. If from the server's ready queue a sub-server receives the CPU, the local ready queue running operation continues until the scheduler decides to run a task. As soon as server S_i^j becomes active, we insert its corresponding budget depletion timer (`budget_timer`) to be invoked at time t_{dep}, where:

$$t_{dep} = \texttt{jiffies} + B_i^j(t).$$

When the `jiffies` is equal to t_{dep}, the budget depletion timer handler is invoked. The handler deactivates its corresponding server (S_i^j) and all of its child servers. If S_i^j is an ancestor (see Definition 1) of the active server, the active server is stopped. If the running task is a child of the server that is getting deactivated, the running task is also stopped. Finally, the timer handler runs the first element that is in the ready queue of S_j (the parent of the server whose budget is depleted). When a server is stopped (either because of its parent budget depletion or because of a preemption), its remaining budget is updated.

Each scheduling entity belongs to a ready queue. The entities at the root level belong to the global ready queue, while the other entities belong to their parent server's ready queue. Therefore, when an entity causes a scheduling event, the event takes place at its corresponding ready queue.

The scheduling decisions are taken only at the scheduling events. We have the following scheduling events in the system.

- task and server release
- server (periodic and constant bandwidth) budget depletion
- task finishing its job
- task and servers leaving the system

When a task is released and the active ready queue is different than the task's ready queue, the task will wait until its ready queue, i.e., its server is activated. Even when the released task's parent is active, it will only be assigned to the CPU if it is able to preempt the running task or the active server. Note that the preemption rules depend on the parent's scheduling policy. When a server is released, it should wait unless one of the following conditions hold in which the released server is allowed to preempt the active server or the running task.

- The server's parent is active and the released server can preempt the running/active scheduling entity.
- The released server outranks (see Definition 2) the active server.

H. Budget adaptation

The budget adaptation is done periodically. The adaptation period is proportional to the server periods. The budget adaptation is done through a function which is called at certain server release events. When calling the budget adapter function, the pointer to the caller server structure is passed to the function. This function uses the `consumed_budget` and the `extra_req_budget` fields in the server data structure to derive the new `budget` field. The `extra_req_budget` field is updated by the server's child tasks and sub-servers that are violating their timing requirements. We also have a mechanism to guarantee that adapting the soft real-time server budgets does not influence the amount of provided budget to the hard real-time servers. For more details about the budget adaptation mechanism refer to [6].

V. Evaluation

In this section, we first design a case-study to study the performance of our framework. Thereafter, we present the results.

A. Tasks

As we mentioned earlier, `AdHierSched` mainly targets systems containing dynamic soft real-time applications. To this end, in our evaluations we use two types of dynamic real-time tasks. Moreover, we use tasks with fixed execution times. In general the following three types of tasks are used in the case-study.

1) Fixed execution time tasks (static tasks). These tasks are indeed a simple `C` program that contain a loop with a constant number of instructions.
2) Mplayer media player[2]. We have modified the source code of the Mplayer media player such that it registers itself to the `AdHierSched` module before starting the playback. Thus, the `AdHierSched` module schedules the player task. In addition, after decoding and playing frames, Mplayer uses the `task_finish_job(task_id)` API function to inform the `AdHierSched` module that a job execution is finished.
3) Image processing program. This program is developed using the OpenCV library and its objective is to filter a color range of its input frame. The input is a movie file to this application in our case-study.

B. Setup

We use an Intel Core i5-2540M processor clocked at 2.60 GHz in which only CPU 0 is active. Our hardware is equipped with 4 GB of memory. In addition, Ubuntu 12.04.2 with Linux kernel version 3.8.2 is used in the evaluations. The scheduler resolution is set to one millisecond.

C. Case-study

The case-study that we investigate in this paper is a system composed of five applications of which one is a hard real-time application and the rest are soft real-time applications. Figure 4 illustrates the structure of the case-study system that we are using in this section. Note that τ_1^1, τ_1^2 and S_1 are hard real-time tasks and a server respectively. The hard real-time server uses a fixed priority scheduler, while the rest of the servers use EDF schedulers for scheduling their children. The tasks are assumed to be ordered based on their priority, i.e, $\pi_1^1 > \pi_2^1$. We use different inputs for the same type tasks. The specifications of the tasks and servers used in the case-study are presented in Table II. All scheduling parameters presented in the table are in milliseconds. We assume that the servers are

[2] http://www.mplayerhq.hu

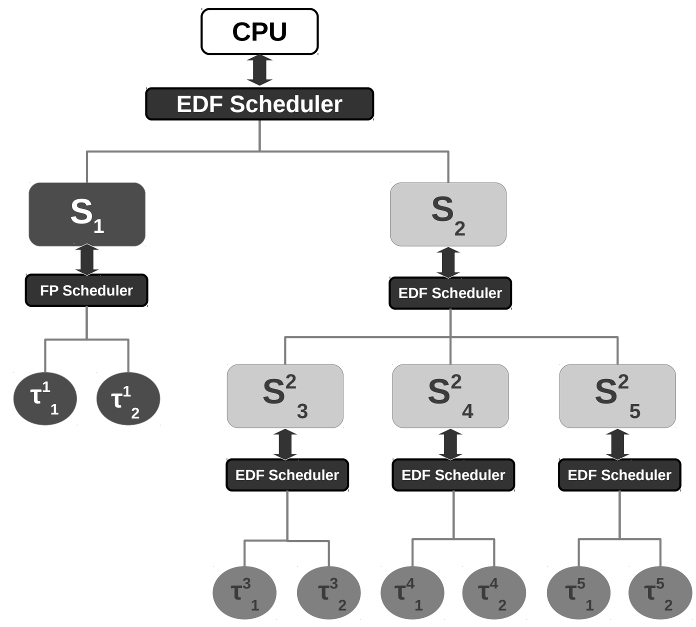

Fig. 4: The sample system investigated in the case-study.

	Hard-Soft	Task type	$P_j - T_i^j$
S_1	HRT server	-	100
τ_1^1	HRT task	1	200
τ_2^1	HRT task	1	400
S_2	SRT server	-	10
S_3^2	SRT server	-	20
τ_1^3	SRT task	2	40
τ_2^3	SRT task	3	200
S_4^2	SRT server	-	100
τ_1^4	SRT task	3	350
τ_2^4	SRT task	3	200
S_5^2	SRT server	-	75
τ_1^5	SRT task	3	350
τ_2^5	SRT task	3	150

TABLE II: Servers and tasks specification in the case-study.

ordered based on their importance meaning that $\zeta_3^2 > \zeta_4^2 > \zeta_5^2$.

D. Workload

In order to observe the workload of the applications, we ran each server separately while assigning 100 % of the CPU to them. The average and the maximum CPU demand of the tasks in the servers are reported in Table III. To illustrate the workload variations of the soft real-time serves we present the CPU demand percentage of S_3^2 in Figure 5.

Server	AVG	MAX
S_3^2	13.35	60.00
S_4^2	12.36	54.00
S_5^2	11.98	44.00
total	37.43	158

TABLE III: The CPU demand percentage of the servers.

Moreover, in our experiments we observe that $C_1^1 = C_2^1 =$ 31. Therefore, based on the suggestion presented in [25] we choose the following period for S_1: $P_1 = 100$. In addition, we derive the minimum budget that guarantees the schedulability of τ_1^1 and τ_1^2 using the analysis presented in [25] which is $B_1 = 39$. Therefore, 39 % of the total bandwidth will be assigned to S_1 and the rest of the bandwidth (61 %) may be utilized by S_2.

Server	fixed	adaptive
S_3^2	3.36	1.11
S_4^2	27.28	6.49
S_5^2	2.19	4.69
total	32.83	12.29

TABLE IV: The deadline miss ratio percentage of the servers.

E. Adaptive budgets

Recall that `AdHierSched` targets soft real-time applications for which their run-time behavior is not known a priori. Therefore, assuming that we have no information about the task CPU demands, we assign an initial budget to the servers and then we let the budget controller to adapt the budgets. We choose the deadline miss ratio as our performance metric that is the number of jobs that finish their execution after their deadline points, divided by the total number of finished jobs. The number of jobs for a server is equal to the sum of its tasks' jobs. As a result of assigning adaptive budgets, the soft real-time servers experience an average of 4.09 % deadline miss ratio. While the most important server S_3^2 experiences only 1.1 % deadline miss ratio. As we show in the rest of this section, the system is overloaded. Thus, missing deadlines is inevitable. However, adapting the bandwidth of the servers, we are serving the real-time tasks in such a way that the available CPU bandwidth is efficiently utilized.

F. Static budgets

Allocating the soft real-time server budgets based on the maximum demand of their tasks is impossible because the sum of the bandwidth (158 %) is more than the available bandwidth (61 %). Therefore, in another experiment we assign the server budgets based on their average CPU demand. Table IV summarizes the results of the case-study for both adaptive and static budget allocation experiments. The adaptive CPU allocation technique results in a total of 20 % less deadline misses than the static technique. In addition, since S_3^2 is the most important application in the system, the deadline misses avoided for this server is of more importance than the other deadline misses potentially avoided. Figure 6 illustrates the budget adaptations of the soft real-time servers used in the case-study.

VI. OVERHEAD

In this section, we report the overhead imposed by the `AdHierSched` module in the case-study presented in Section V. Note that our measurements exclude the Linux scheduler overhead that is responsible to assign the `AdHierSched`

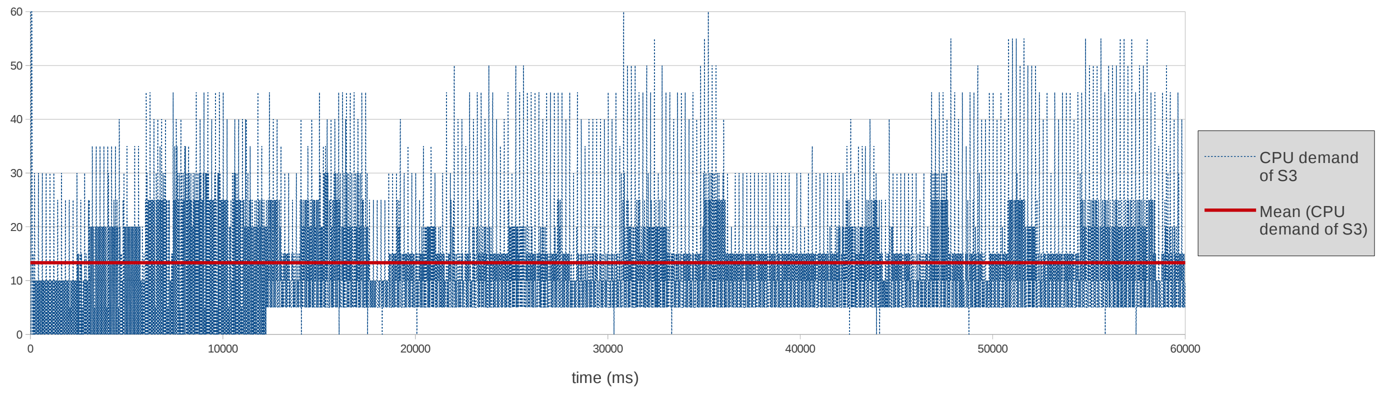

Fig. 5: The CPU demand variation of S_3^2.

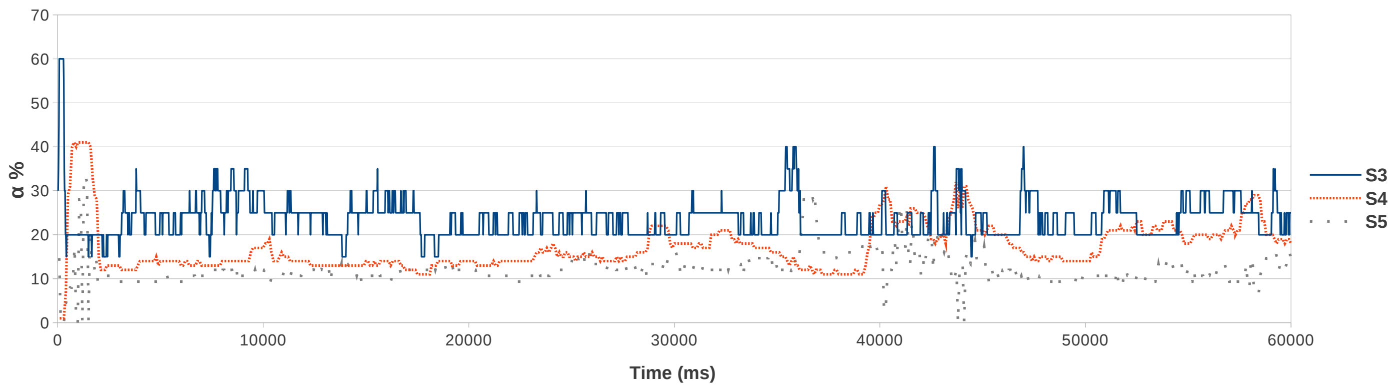

Fig. 6: The budget adaptations over the course of one minute experiment.

real-time tasks to the CPU. Therefore, we do not include the context switch overhead.

There are two sources of overhead: (i) the multi-level hierarchical scheduling overhead, i.e., the amount of extra calculation that is done just for scheduling the real-time time tasks in a hierarchical manner. (ii) the budget adaptation overhead, i.e., the amount of extra calculations that are done because of adapting the server budgets.

We measured the two types of overhead for the case-study. The total overhead is less than 0.2 % ($\simeq$ 122 milliseconds). Figure 7 shows that the budget adaptation overhead ($\simeq$ 8 milliseconds) has a small share of the total overhead. The figure represents the overhead present in the case-study explained in Section V. The overhead has been measured using time stamps that are monitoring the execution length of the timer handlers and the `task_finish_job(task_id)` API function. Then, the total value is divided by the total time that the experiment ran.

VII. Conclusion

In this paper, we presented the implementation details of our adaptive hierarchal scheduling framework which is called `AdHierSched`. We showed how the framework is

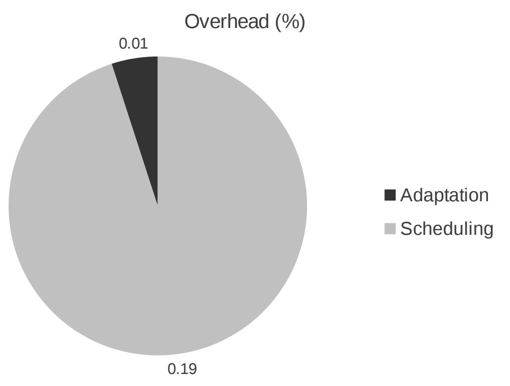

Fig. 7: The overhead of the `AdHierSched` module.

implemented in the Linux kernel as a kernel loadable module. We demonstrated that our framework can efficiently deal with unknown workloads. Finally, we reported the overhead of our framework.

Our implementation can be improved in a number of ways. For instance, we can use a more efficient queue structure to reduce the overhead of the `AdHierSched` scheduler. As a next step we are contemplating extending our framework to multiprocessors. Although we are not currently considering I/O operations, we would like to investigate the implications of modeling them in our adaptive framework. For instance, we can model the I/O requests as critical sections and we can use available semaphore based protocols such as SIRAP [27] and HSRP [28].

Acknowledgment

The authors wish to thank Shinpei Kato for his help with adopting the Mplayer code, and acknowledge the constructive comments of anonymous reviewers.

References

[1] G. Lipari and S. Baruah, "A hierarchical extension to the constant bandwidth server framework," in *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, May 2001, pp. 26–35.

[2] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in an open environment," in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, December 1997, pp. 308–319.

[3] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'01)*, May 2001, pp. 75–84.

[4] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: response-time analysis and server design," in *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT'04)*, September 2004, pp. 95–103.

[5] F. Zhang and A. Burns, "Analysis of hierarchical EDF pre-emptive scheduling," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*, December 2007, pp. 423–434.

[6] N. M. Khalilzad, M. Behnam, and T. Nolte, "Multi-level adaptive hierarchical scheduling framework for composing real-time systems," in *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13), to appear*, August 2013.

[7] M. Barabanov and V. Yodaiken, "Real-time linux," *Linux journal*, vol. 23, March 1996.

[8] P. Mantegazza, E. L. Dozio, and S. Papacharalambous, "RTAI: Real Time Application Interface," *Linux Journal*, vol. 2000, no. 72es, April 2000.

[9] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, July 2008, pp. 63–72.

[10] R. Inam, J. Maki-Turja, M. Sjodin, S. M. H. Ashjaei, and S. Afshar, "Support for hierarchical scheduling in FreeRTOS," in *Proceedings of the 16th IEEE International Conference on Emerging Technologies Factory Automation (ETFA'11)*, September 2011, pp. 1–10.

[11] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar, "ExSched: An external CPU scheduler framework for real-time systems," in *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)*, August 2012, pp. 240–249.

[12] M. van den Heuvel, R. J. Bril, J. J. Lukkien, and M. Behnam, "Extending a HSF-enabled open-source real-time operating system with resource sharing," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'10)*, July 2010, pp. 71–81.

[13] J. Regehr and J. Stankovic, "HLS: a framework for composing soft real-time schedulers," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, December 2001, pp. 3–14.

[14] G. Parmer and R. West, "HIRES: A system for predictable hierarchical resource management," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, April 2011, pp. 180–190.

[15] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson, "Microkernels meet recursive virtual machines," in *Proceedings of the 2nd USENIX symposium on Operating systems design and implementation (OSDI'96)*, 1996, pp. 137–151.

[16] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin, "Implementation of compositional scheduling framework on virtualization," *SIGBED Rev*, vol. 8, pp. 30–37, 2011.

[17] J. Lee, S. Xi, S. Chen, L. T. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky, "Realizing compositional scheduling through virtualization," in *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*, April 2012, pp. 13–22.

[18] M. Danish, Y. Li, and R. West, "Virtual-cpu scheduling in the quest operating system," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, April 2011, pp. 169–179.

[19] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constraints in virtualised services," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, vol. 2, July 2009, pp. 73–78.

[20] M. Åsberg, N. Forsberg, T. Nolte, and S. Kato, "Towards real-time scheduling of virtual machines without kernel modifications," in *Proceedings of the 16th IEEE International Conference on Emerging Technology and Factory Automation (ETFA'11), Work-in-Progress (WiP) session*, September 2011, pp. 1–4.

[21] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA-adaptive quality of service architecture," *Softw. Pract. Exper.*, vol. 39, no. 1, pp. 1–31, January 2009.

[22] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzen, V. Romero, and C. Scordino, "Resource management on multicore systems: The ACTORS approach," *Micro, IEEE*, vol. 31, no. 3, pp. 72–81, May-June 2011.

[23] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina, "An implementation of the earliest deadline first algorithm in Linux," in *Proceedings of the ACM symposium on Applied Computing (SAC'09)*, March 2009, pp. 1984–1989.

[24] S. Kato, R. Rajkumar, and Y. Ishikawa, "AIRS: Supporting interactive real-time applications on multicore platforms," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS'10)*, July 2010, pp. 47–56.

[25] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the 24th IEEE Real-Time Systems Symposium, (RTSS'03)*, December 2003, pp. 2–13.

[26] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998, pp. 4–13.

[27] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Sirap: a synchronization protocol for hierarchical resource sharing in real-time open systems," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT'07)*, 2007, pp. 279–288.

[28] R. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, December 2006, pp. 257–270.