A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux

Mikael Åsberg and Thomas Nolte
MRTC/Mälardalen University
P.O. Box 883, SE-721 23,
Västerås, Sweden
{mikael.asberg,thomas.nolte}@mdh.se

Shinpei Kato
Carnegie Mellon University
Department of Electrical and Computer Engineering
Shinpei@ece.cmu.edu

***Abstract*—This paper presents a Hierarchical Scheduling Framework (HSF) recorder for Linux-based operating systems. The HSF recorder is a loadable kernel module that is capable of recording tasks and servers without requiring any kernel modifications. Hence, it complies with the reliability and stability requirements in the area of embedded systems where proven versions of Linux are preferred. The recorder is built upon the loadable real-time scheduler framework RESCH (REal-time SCHeduler). We evaluate our recorder by comparing the overhead of this solution against another (patched) recorder. Also, the tracing accuracy of the HSF recorder is tested by running a media-processing task together with periodic real-time Linux tasks in combination with servers. The tests are recorded with the HSF recorder, and the Ftrace recorder, in order to show the correctness of the experiments and the HSF recorder itself.**

***Index Terms*—real-time systems, hierarchical scheduling, replay debugging, execution visualization**

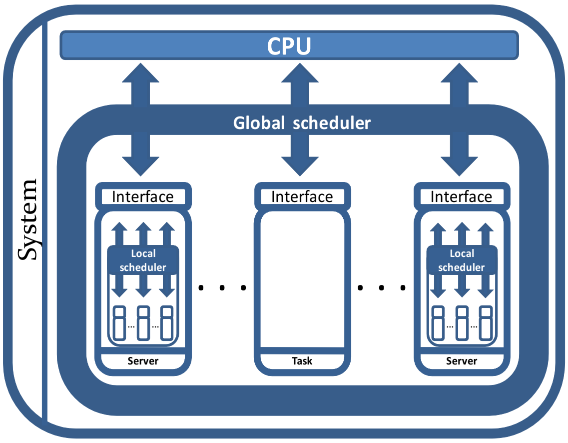

Fig. 1. Hierarchical Scheduling Framework

I. Introduction

Introduction The research that we conduct is primarily focused on the development of hierarchical scheduling [1], [2], [3]. Our previous and ongoing work within hierarchical scheduling includes practical (implementation) aspects of this kind of scheduling [4], [5], the applicability/usage [6], [7] of it, as well as applying formal methods [8] on it. In server-based scheduling (the predecessor of hierarchical scheduling), tasks (a sequence of instructions) are only allowed to execute whenever their server (the virtual task which they belong to) runs. The server itself executes according to some scheduling scheme (global scheduling) which is independent of the tasks. The advantage is that it can improve the response time (the time length between task activation and completion) of event triggered tasks, and still keep the scheduling deterministic since the server scheduling parameters are known and included in the schedulability analysis. Further, introducing a scheduler within each server (local scheduling) makes it more general since it supports time triggered tasks as well. This can be generalized even further by representing a task as a set of tasks together with a scheduler. When we have separate scheduling inside a server, i.e. both global and local scheduling, then we refer to hierarchical scheduling or a Hierarchical Scheduling Framework (HSF), this is illustrated in Figure 1.

Hierarchical scheduling has several advantages, besides improving response time of event triggered tasks. It enables parallel development of system parts (subsystems), simplifies integration of subsystems (analysis), supports runtime temporal partitioning and safe execution of tasks etc. However, except for ARINC653 [9], [10] compliant operating systems that are commonly found in avionics applications, hierarchical scheduling is rarely an integrated part of an operating system (OS). Indeed, there is a need to develop/implement new scheduling algorithms, such as hierarchical scheduling, in the area of embedded and/or real-time systems [6]. A motivation of this can be found in our scheduling example in the evaluation (Section IV), where we let a media-processing task (which does a movie playback) execute within a server (server-based scheduling). The server executes with a certain frequency, giving (guaranteeing) the media task an even amount of CPU power which improves the playback quality of the movie, even though it executes among other time triggered tasks. The media task has an unknown execution pattern, i.e., the releases are undefined. Still, we get predictability (since we can analyze the behavior) from both the media tasks point of view, and the time triggered tasks. Also, we avoid (temporal) interference at runtime, meaning that we get a safe execution environment for the tasks because temporal errors do not propagate between the media task and the time triggered tasks.

From a practical point of view, it is an advantage if hierarchical scheduling can be implemented easily/efficiently and without modifying the kernel. The latter makes it easier for both developers and users since there is no need to maintain/apply kernel modifications every time the kernel is replaced or updated. Moreover, keeping the scheduler isolated in a kernel module, without modifying the kernel, simplifies debugging and potential certification of its correctness (component-based development advantages). We see that the RESCH scheduling framework [11] is useful because it has the advantages mentioned, since it does not need any kernel modifications. Also, it makes scheduler development easier because it simplifies the scheduling interface to the user and it supports the development of schedulers (plugins) which run as independent kernel modules. However, while the development of schedulers are simplified with this framework, it lacks support for debugging the schedulers. That is why we have developed a HSF recorder, which can easily be plugged in to a server-based/hierarchical scheduler, developed in RESCH. The recorder does not require kernel modifications and it is of course also suitable for analyzing the runtime behavior of tasks/servers since the recorded trace can be visualized graphically with the Tracealyzer [12] or Grasp [13] visualization tools. In turn, these tools can present valueable trace data such as execution- and response-time.

The HSF recorder is able to record the following scheduling events during run-time:

1) The time instance when a task/server is released (even though it might not start to execute).
2) The time instance when a task/server starts to execute.
3) When there is a task/server context switch, the recorder distinguishes between preemption and non-preemption.
4) The time instance when a task/server finishes its execution.

Contribution The main contributions of this paper are:

1) We have implemented a task/server recorder with the use of RESCH, i.e., it does not require any kernel modifications. The recorder enables debugging at task and server level, in Linux based real-time/general-purpose OSs.
2) We have evaluated our HSF recorder by implementing yet another recorder (Section II-C), using the technique presented in [14], and compared the overhead of this recorder, with the HSF recorder.
3) We have tested our recorder by running a media-processing task together with time triggered tasks and servers. The example shows how the playback quality gets improved by putting the media-processing task in a server. The HSF recorder is used in this example to debug and display the runtime behavior.

Outline The outline of this paper is as follows: Section II presents preliminary background, in Section III we describe the HSF-recorder implementation. Section IV evaluates the overhead and tracing accuracy of the HSF recorder. Section V presents related work, and finally, Section VI concludes.

II. Preliminaries

A. System model

We assume fixed-priority, preemptive, scheduling of periodic tasks, according to the periodic task model [15]. A task i is presumed to have the following parameters, $\langle T_i, WCET_i, D_i, pr_i \rangle$, where the period T_i represents the frequency in which the task is released for execution, $WCET_i$ is the worst case execution time of the task, the relative deadline D_i (within the period) is when the task must complete its execution (RESCH monitors this) and pr_i is the task priority (lower value represents higher priority). Also, all tasks are assumed to execute independently of eachother and on the same core, i.e., single core.

The servers are also assumed to have fixed priority and they are scheduled preemptively and periodic. A server j has similar parameters as tasks, i.e. $\langle P_j, Q_j, pr_j \rangle$, where P_j is the server period, Q_j is defined as a budget (which is the time given at each period P_j to the tasks within the server) and pr_j is the server priority (lower value represents higher priority).

B. RESCH

We have been developing a loadable real-time scheduler framework, RESCH [11], designed to work with the POSIX-compliant SCHED_FIFO scheduling policy implementation. RESCH has previously been used as the basis for another scheduler called AIRS [16] - a multi-core CPU scheduler for interactive real-time applications. As mentioned previously, RESCH is a modification-free scheduling framework for Linux. It supports periodic tasks which can be scheduled in a fixed-priority preemptive manner. RESCH is simply composed of external kernel modules and user-space libraries for easy installation. It gives both an interface to the users in user space (e.g. a task specific interface like `rt_wait_for_period()`) as well as in the kernel space. The kernel space API (Application Programming Interface) has the interface shown below:

1) **task_run_plugin( )**
2) **task_exit_plugin( )**
3) **job_release_plugin( )**
4) **job_complete_plugin( )**

These functions can be implemented by a **RESCH plugin** (Figure 3), i.e., a kernel module that has access to the RESCH kernel API. These functions are called in the **RESCH core** at certain events which are illustrated in Figure 2. Functions 1) and 2) are executed every time a task registers/unregisters to RESCH. With register we mean that the task does a RESCH API call, transforming it to a **RESCH task**, which creates a RESCH TCB (Task Control Block) and puts it in the RESCH ready-queue etc. A RESCH TCB has, among other real-time specific data, a reference to its corresponding Linux task TCB (`task_struct`). Once the task is registered in RESCH, it will be scheduled periodically (and preemptive) according to its real-time priority. The primitives 3) and 4) are called whenever a RESCH task is released for execution or when it has finished its execution. The plugins get these scheduling notifications and can thereby affect scheduling, trace tasks etc. The plugin notifications are shown in Figure 2. When a task notifies RESCH that it has finished its execution in its current period, the RESCH core will inform any plugin about this event and set a timer for the release of the tasks next period.

As a last step, it will call the Linux kernel to re-schedule another task. The next running task might be a RESCH task or any other Linux process.

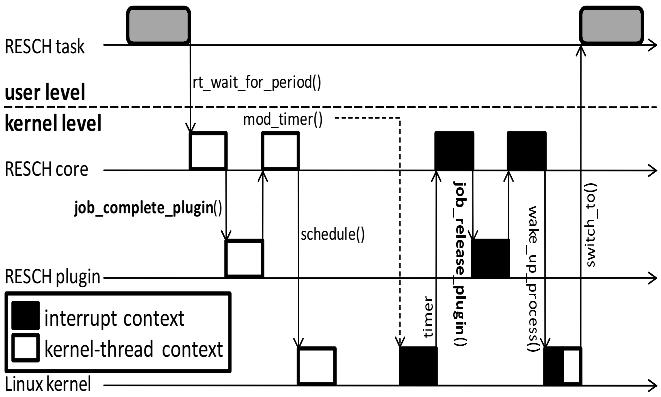

Fig. 2. RESCH control flow

When the kernel responds to the corresponding timeout (task release), a handler in the RESCH core will get notified about this event. The handler will notify any plugin about the task release and then call the kernel to wake up the task.

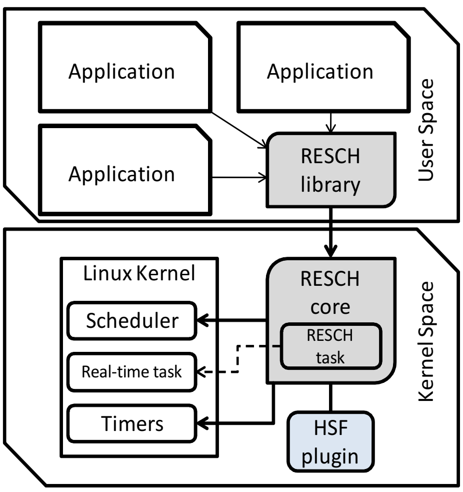

Fig. 3. RESCH framework

In Linux, since kernel version 2.6.23 (October of 2007), tasks can be either a $fair$ or a $real\text{-}time$ task. The latter group has higher priority (0-99 where 0 is highest) than fair tasks (100-140). A task that registers to RESCH is automatically transformed to a real-time task. RESCH is responsible for releasing tasks, and tasks registered to RESCH must notify when they have finished their execution in the current period. In this way, RESCH can control the scheduling. RESCH uses an absolute-time clock, i.e., it does not wrap around. Also, release times are stored as absolute values, so release patterns are exact.

The cost of having a modification-free solution is that RESCH can only see scheduling events related to its registered tasks. Real-time tasks with higher priority than RESCH tasks (i.e. tasks that are not registered in RESCH) can thereby interfere with RESCH tasks without the RESCH core being able to detect it. A simple solution to this problem is to schedule all real-time tasks with the RESCH framework.

C. *Task-switch hook patch*

Our previous work [14] includes an implementation of a `task_switch_hook` function (Figure 4), residing in a kernel module, which is called by the Linux scheduler at every scheduler tick. In this way, it is possible to record task scheduling events. This solution requires modification of two code lines in two separate kernel source files (`sched_rt.c` and `sched_fair.c`). The modification of file `sched_rt.c` is illustrated in Figure 4 (a similar change is done in `sched_fair.c`). Linux has (since kernel version 2.6.23) two scheduling classes, namely the $fair$ and the $real\text{-}time$ scheduling classes. When a new task should be released, the Linux scheduler iterates through its scheduling classes (first the $real\text{-}time$ class, secondly the $fair$ class) in order to find the next task to release.

The modification (Figure 4) makes it possible to re-direct a scheduling class' function pointer `.pick_next_task` to point to a user defined function (i.e., our function `task_switch_hook`), instead of the original function `pick_next_task_rt`. Our function will instead point to `pick_next_task_rt`, in this way, we do not alter the kernel functionality other than executing our function `task_switch_hook` (which contains user defined code) just before `pick_next_task_rt` starts to execute. Our function (hook) can be inserted and removed during runtime. A task recorder can easily be implemented (as a kernel module) and use the `task_switch_hook` function to register task context switches, however, the kernel must be modified.

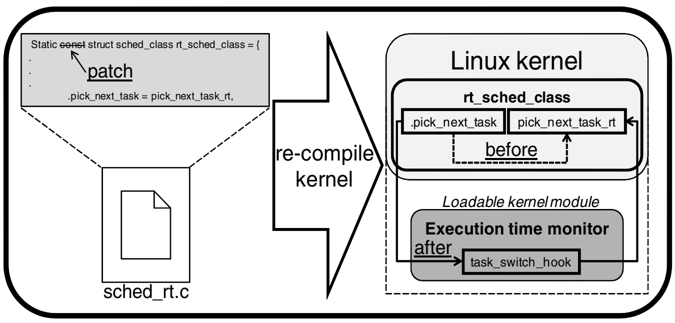

Fig. 4. Hook patch

III. IMPLEMENTATION

The implementation of the HSF recorder is based on the scheduler plugin HSF which in turn is based on the scheduling framework RESCH. Figure 5 shows that the HSF scheduler uses primitives exported by RESCH and exports these, as well as server specific primitives, to the recorder. These primitives are used to register server and task context switches. Note that the flexible structure allows for new scheduler plugins to reuse the recorder as long as they export the same primitives.

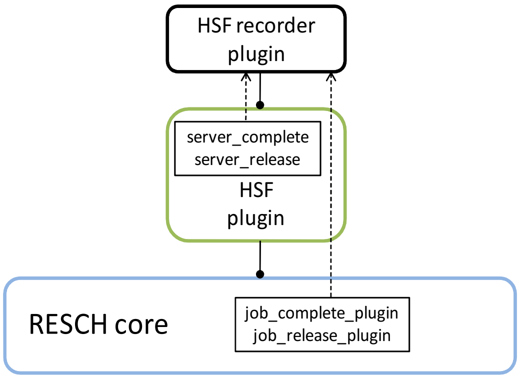

Fig. 5. HSF-recorder plugin

For the recording to work correctly, it is assumed that no higher priority *real-time* Linux tasks, which are not registered by RESCH, are executed.

The current implementation does not support *load balancing* (a function in Linux that migrates tasks to other CPUs based on load). This is because the RESCH scheduler cannot detect task migrations made by the Linux scheduler.

Each recorded event has 2 records:

- ID of the next task/server to execute.
- Timestamp of the event.

The ID of the next task/server is used to calculate the previous task/server. The 4 hook functions (Figure 5) are used by the recorder to save scheduling records in memory (this is a circular implementation). The recorder flushes the recorded data to disk when it gets unloaded by the user. The recording format can easily be converted to match any visualization tool. We have successfully converted the format to fit with the Tracealyzer [12] and the Grasp [13] visualization tools. We use Grasp in the evaluation (Section IV) in order to visualize the trace of the HSF recorder since it also supports hierarchical scheduling in addition to regular (flat) scheduling.

Figure 6 illustrates how the HSF recorder gets triggered. As can be seen, the HSF scheduler gets triggered by its own timers as well as by the RESCH core. The HSF scheduler relays task releases and completions to the HSF recorder when the HSF scheduler itself is triggered by the RESCH core. Whenever the HSF scheduler gets triggered by a timer, it automatically calls its server release/completion plugin, which in turn starts the recorder. The figure also shows that the HSF recorder executes mostly in interrupt context. This makes it less expensive in terms of context-switch overhead.

IV. Evaluation

We have evaluated our HSF recorder by recording a set of tasks and servers (Table I and II). In our example, task `rt_task1` belongs to server `Server0`, `rt_task2` and `rt_task3` does not belong to any server while `rt_task4` belong to server `Server1` and `rt_task5` to `Server2`.

The evaluation shows two aspects: the measured overhead (section IV-A) of the HSF recorder compared to the patched recorder [14], and an example of how the Quality of Service (QoS) of multimedia tasks can be improved with hierarchical scheduling as well as how our HSF recorder can assist in this work (section IV-B). In the multimedia example we used our HSF recorder and the Ftrace [17] recorder.

During our experiments, the two recorders were recording the tasks and servers simultaneously.

Task-name	T	$WCET$	D	$Prio$	$Server$
rt_task1	80	9	80	0	Server0
rt_task2	200	75	200	1	-
rt_task3	105	9	105	2	-
rt_task4	500	100	500	3	Server1
rt_task5	-	-	-	4	Server2

TABLE I
TASKS USED IN THE EVALUATION

Server-name	P	Q	$Prio$
Server0	40	6	1
Server1	90	23	2
Server2	25	8	0

TABLE II
SERVERS USED IN THE EVALUATION

A. *Overhead measurements*

In order to estimate the overhead impact, we measured the execution time of the patched and the HSF recorder, running simultaneously and recording the same trace. We also noted the amount of data (in kilo bytes) that the two recorders produced (out of curiosity we also measured Ftrace). We implemented an optimized version of the patched recorder, **Patch** (Table III) so that it only saved recorded data of the tasks that we were interested in recording. In this way, the comparison to the HSF recorder became fair since it is only triggered at task/server events related to the tasks/servers we are interested in recording (RESCH related task and servers).

Recorder	Exec. time (μs)	Rec. data (KB)
HSF	45	10.5
Patch	1246	17.4
Ftrace	-	888.6

TABLE III
MEASUREMENTS OF THE RECORDERS

The values listed in Table III are the average measured values of 10 runs and the recorders recorded about 4 seconds at each run. We see that the HSF recorder has a ratio of 4.3 μs/KB while Patch has 71.6 μs/KB. The conclusion is that the HSF recorder produces less overhead than the patched recorder, comparing the execution-time/data ratio. The small amount of recorded data compared to Ftrace suggests that our recorder might be a better option if the user is only interested in a subset of tasks. Having a small amount of overhead is attractive for recorders since they can remain active in shipped

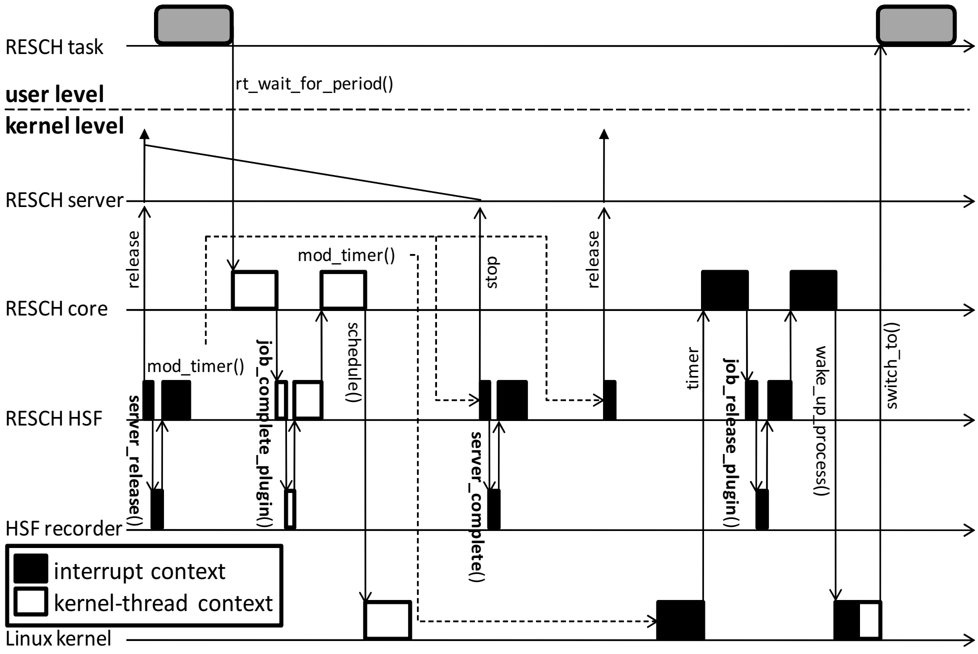

Fig. 6. HSF recorder control flow

products (without wasting too much resources), and thereby eliminating the probe effect.

B. Multimedia example

The purpose of this example is to show how a multimedia task (processing a movie) can benefit from hierarchical scheduling in such a way that the movie playback runs more smoothly. The HSF scheduler has never been evaluated (and debugged) as properly as the example we are about to show, so this is a good case study for the HSF recorder. We run the multimedia task in different setups (with and without hierarchical scheduling), and measure its performance. The hierarchical scheduling gives the multimedia task an even amount of CPU power, and thereby improves the movie playback. Note that all of this is done, including the recording, without modifying the kernel. The HSF recorder plays a key role since knowledge of the scheduling behavior is important in order for the result of this evaluation to be correct. For example, the recorder shows that the tasks and servers get the amount of CPU that we specify (i.e., that tasks run within their servers) and that the tasks/servers run according to the specified frequency and $WCET/Q$. During our experiments, the recording showed that the HSF cannot keep tasks within their server if they do a lot of blocking (e.g. multimedia tasks). Therefore, we set lowest priority to the multimedia task and add idle tasks with higher priority than the multimedia task. This will keep the multimedia task within its server, thereby guaranteeing the upper limit on its resource supply. This was confirmed by the recording of our HSF recorder. A second recorder (Ftrace) was also used in order to show that the HSF recorder recorded correctly. We used the Grasp tool [13] to visualize our recordings (for both the HSF recorder and Ftrace), since it can display both tasks and servers.

In this example, we have 5 tasks, i.e., `rt_task1` to `rt_task5` (Table I). Tasks `rt_task1` to `rt_task4` are dummy tasks, i.e., they just loop (`rt_task1` in Figure 7). `rt_task5` does a movie playback, its task body is shown in Figure 7.

```
// rt_task1
int main(int argc, char *argv[ ])
{
        .
        .
        for (i = 0; i < NR_OF_JOBS; i++) {
               for (j = 0; j < USEC_UNIT; j++) {
               }
               if (!rt_wait_for_period()) {
                  printf("deadline is missed!\n");
               }
        }
        .
        .
}
// rt_task5
int main (int argc, char *argv[ ])
{
        .
        .
        libvlc_media_player_play(player);
        .
        .
}
```

Fig. 7. Task bodies

`rt_task5` used the libVLC[1] for movie playback and the library itself has the nice property that the movie processing can be executed by a task running in *real-time* mode. We executed `rt_task5` in 4 different setups:

[1] libVLC http://wiki.videolan.org/Libvlc

1) rt_task5 with lowest priority and tasks rt_task1 to rt_task4 with priority order as in Table I.

2) rt_task5 with medium priority (in between rt_task2 and rt_task3) and tasks rt_task1 to rt_task4 with priority order as in Table I.

3) rt_task5 with highest priority and tasks rt_task1 to rt_task4 with priority order as in Table I.

4) rt_task5 executed in server Server2, and rt_task1 and rt_task4 in server Server0 and Server1 respectively (rt_task2 and rt_task3 was not included in this setup).

Given these 4 setups, task rt_task5 will get different amount/distribution of CPU power and the processing of movie images (frames) will therefore also be affected. The movie processing is measured in amount of produced frames per second (FPS). The CPU utilization (percantage of CPU time) of task rt_task5 is shown in Table IV as well as the frame rate of which rt_task5 is processing a movie. We measured the FPS by timestamping the beginning and end of the movie playback system call and dividing the amount of frames of the movie with the measured time. The amount of frames is 91 and this value was generated by Mplayer[2] (using the benchmark flag). It is important to note that the CPU utilization given in Table IV is the *available* CPU time, it does not mean that task rt_task5 uses this CPU time. The FPS values may not considered to be 100% accurate, but it shows the approximate efficiency. For example, running rt_task5 with 100% CPU should of course not give worse FPS value than running it with 32% CPU. These values are of course affected by overhead from the Linux kernel etc. We ran the the experiments on an Intel Pentium Dual-Core (E5300 2,6GHz) platform, equipped with a Linux kernel version 2.6.31.9, running with *load balancing* disabled. The recorded tasks (and servers) ran on the same core, i.e., all tasks were migrated to CPU #0 at initialization phase.

Setup	CPU utilization (%)	FPS
Lowest prio	22.65	22.55
Medium prio	51.25	23.57
Highest prio	100	25.48
HSF	32	25.66

TABLE IV
FPS OF TASK rt_task5

The conclusion based on Table IV is that the distribution of CPU power influences the frame frequency a lot and that utilization alone is not sufficient for determining this. For example, giving task rt_task5 51.25% of the CPU produces less FPS than giving it 32%. The 32% CPU is guaranteed (no more no less) and it is distributed evenly as can be seen by the recording of HSF recorder in Figure 8 (visualized with the Grasp tool [13]).

[2]Mplayer http://www.mplayerhq.hu/design7/news.html

Apparently, (during our experiments) task rt_task5 must have been active when other higher priority tasks were occupying the CPU, thereby temporarely getting less than 51.25% CPU. This is not the case when running the multimedia task in its server, since it is always supplied 32%.

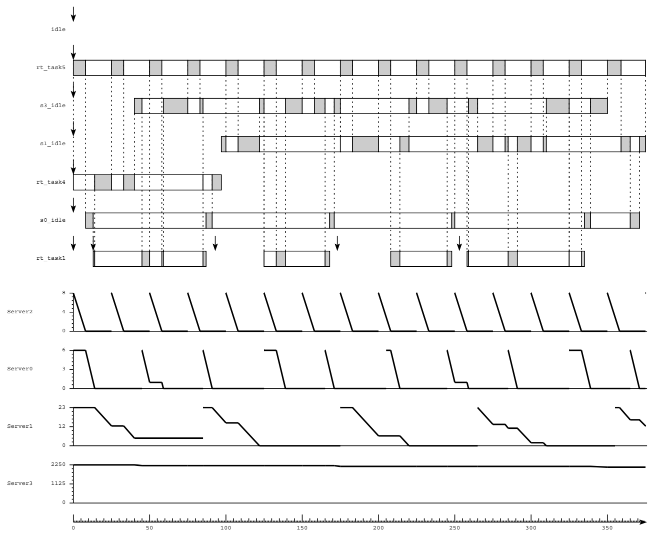

Fig. 8. Tasks and servers recorded with the HSF recorder

Figure 9 shows the same trace as in Figure 8, but recorded with the Ftrace recorder. As can be seen, the HSF recorder records correctly, also, it shows that task rt_task5 does not consume CPU continuously (i.e., it blocks often).

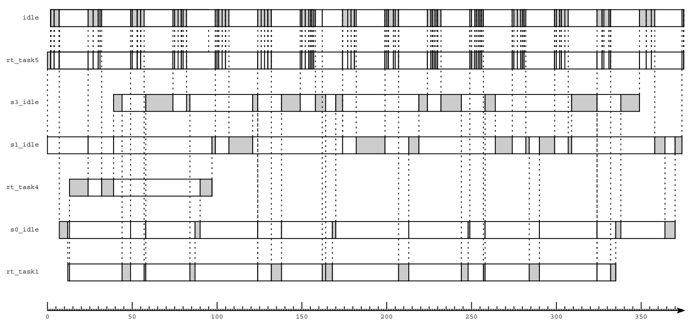

Fig. 9. Tasks recorded with the Ftrace recorder

Figure 10 shows a trace by our HSF recorder when task rt_task5 was running with lowest priority, without HSF. As can be seen, the CPU availability for task rt_task5 is highly dependant on when higher priority tasks execute.

Our example shows that it is difficult to fine tune the CPU supply for a multimedia task, i.e., we can only do it by changing the priority of the task since it is not periodic. However, it is possible to do tuning by setting server period, budget and priority, when using HSF. The main contribution of this example is the trace (Figure 8) made by the HSF recorder which shows the correctness of the CPU distribution, made by HSF, to real-time tasks (with media processing). We have also tested the correctness of the HSF recorder by comparing

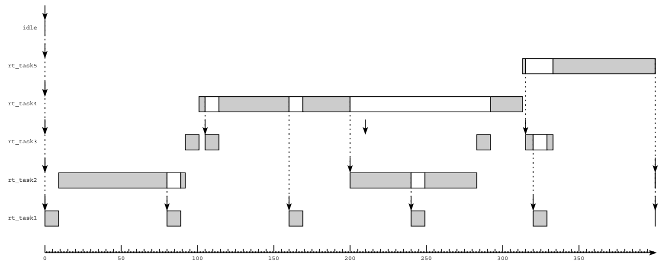

Fig. 10. Tasks recorded with the HSF recorder

its trace results with the Ftrace recorder, i.e., the trace in Figure 8 is identical with the trace in Figure 9, which shows that it records correctly. Also, the trace in Figure 8 shows the amount of unused CPU time (slack time) at both server level and within each server, since the different idle tasks represent this. For example, server `Server3` (which has lowest priority) and its task `s3_idle` represent slack time at server level, while `s0_idle` represent unused time in `Server0`. The conclusion is that the HSF recorder can be a good tool for debugging hierarchical schedulers in RESCH, since it records accurately and with low overhead. Further, this example shows that our (HSF) recorder and scheduler records (and schedules) correctly, even though we do not modify the kernel.

V. Related work

The idea of our solution is based on the replay debugging approach [18], which records system events online and replays them offline. In later work [19], the replay debugging has been extended to be compiler- and OS-independent. While the replay debugging works with off-the-shelf compilers for application-level debugging, our solution is self-contained software using Grasp [13] for OS-level debugging, and it is primarily focused on real-time scheduler debugging.

The SCHED_DEADLINE project [20], which is in charge of the EDF scheduler implementation for Linux, has used the sched_switch tracer provided by the Ftrace toolkit [17] to output the recordings of context switches. The output logs are later converted to the Value Change Dump (VCD) format so that GtkWave can visualize the task execution traces. The trace can of course be converted to other trace formats, such as the Tracealyzer [12] or the Grasp [13] format. Given that Ftrace is supported by the Linux community, it is reasonable to use this toolkit to trace task executions for kernel debugging, but it is dedicated to the Linux kernel, so it is not necessarily suitable for real-time scheduler debugging in general. For instance, sched_switch does not catch job releases, however, context switches are precisely traced, and it can distinguish between task completions and task preemptions. Our solution is more flexible and integrated in that it is available not only for the Linux kernel, but also for other OSs, once the RESCH framework is ported to other platforms.

Our previous work [21] includes a simple task recorder in Linux (based on RESCH) which supports the Tracealyzer [12] and the Grasp [13] format. Further, we have also implemented a task recorder [14] (in Linux) which is able to record all task scheduling events, but it requires modifications to the kernel.

DTrace [22], SystemTrap [23], LTT [24], and LTTng [25] are advanced tools for OS debugging. They are oriented for tracing entire kernel events, so it is required that the developers understand how to use them. Meanwhile, our solution is more simplified by focusing on real-time scheduler debugging, and it is very easy to use in practice.

Real-Time Application Interface for Linux (RTAI) [26] is a collection of loadable kernel modules and a kernel patch which together provides a rich real-time API to the user. It gives the possibility to add/delete hooks for every task-start, task-switch and task-delete. These hooks give the possibility to monitor task execution in a detailed level.

Tracealyzer [12] is a visualization and analysis tool for embedded systems. It can visualize task traces as well as task communication. Recorders implemented in the OSs VxWorks, OSE, Rubus and RTXC support the Tracealyzer format.

VI. Conclusion

We have presented the implementation and evaluation of a task/server recorder based on the RESCH (REal-time SCHeduler) framework in Linux. RESCH is a scheduling framework for Linux which support scheduler plugins, i.e., multi-, uni-core, flat-, server-based-scheduling etc. Our recorder implementation is a plugin on top of an already existing hierarchical scheduler plugin called HSF (Hierarchical Scheduling Framework). This framework supports fixed-priority preemptive scheduling of servers as well as tasks. The HSF recorder uses scheduling primitives supported by RESCH itself, and HSF, in order to record scheduling events. The RESCH framework, the HSF scheduler plugin as well as our HSF recorder require no modification of the kernel and this is the main contribution of this approach. To the best of our knowledge, this is the first attempt to perform task tracing (within hierarchical scheduling) in Linux, without kernel modifications.

The evaluation of the HSF recorder includes two parts:

- Overhead comparison against an optimized version of our previously implemented task-switch patch [14].
- The correctness of the HSF recorder (as well as the HSF scheduler) is tested with a media processing example. The tracing capability and accuracy of the HSF recorder is compared against the main-line Linux recorder Ftrace [17].

Our HSF recorder produces very low overhead, in terms of CPU consumption, compared to the task-switch patch. The amount of recorded data is also much smaller than Ftrace, suggesting that the HSF recorder could be a better choice if only a subset of Linux tasks is of interest to monitor.

The media-processing example shows 5 real-time tasks running with, and without servers, i.e., with the HSF scheduler activated and with only RESCH. In the example, we show that one of the tasks (which is processing a movie) produces higher frame rate with theoretically lower CPU utilization (using the HSF scheduler) than with higher CPU utilization (using only RESCH). The reason for this is that HSF gives the media-processing task better CPU resource distribution. In this example, the HSF recorder contributes by showing that the media task uses only its allocated CPU resource, thereby showing that the example is correct. It also shows a weakness with the HSF scheduler in that it has problems with keeping media tasks (and similar tasks which blocks often) within its server. However, non-blocking real-time tasks are shown to be properly contained inside their servers. All traces from the HSF recorder, in this example, are done in parallel with the Ftrace recorder, thereby showing the accuracy (and correctness) of our HSF recorder.

The conclusion is that the HSF recorder could be a good tool for debugging hierarchical schedulers in RESCH. The recorder can, together with a visualization tool, such as Grasp [13], visualize the execution of tasks and servers as well as display worst-case, best-case and average value of both execution- and response-time of tasks. In case that the Linux kernel is configured with Ftrace, then it could be useful to use also, since it complements our recorder well. Our recorder can record server events and task releases, while Ftrace can record the context switches between the RESCH real-time tasks and other Linux tasks.

Future work includes merging Ftrace and the HSF recorder to get more detailed and complete traces. We will also continue with improving the HSF scheduler plugin as well as developing new server-based schedulers (Bandwidth Sharing Server, Constant Bandwidth Server, Sporadic Server etc.) and support for multi-core scheduling (and tracing).

References

[1] P. Goyal, X. Guo, and H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[2] Z. Deng and J. W.-S. Liu, "Scheduling Real-time Applications in an Open Environment," in *Proc. of the 18th IEEE International Real-Time Systems Symposium*, 1997.

[3] J. Regehr and J. A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *Proc. of the 22nd IEEE International Real-Time Systems Symposium*, 2001.

[4] M. Åsberg, M. Behnam, T. Nolte, and R. J. Bril, "Implementation of Overrun and Skipping in VxWorks," in *Proc. of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2010.

[5] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards Hierarchical Scheduling on top of VxWorks," in *Proc. of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2008.

[6] M. Åsberg, M. Behnam, F. Nemati, and T. Nolte, "Towards Hierarchical Scheduling in AUTOSAR," in *Proc. of the 14th International Conference on Emerging Technologies and Factory Automation*, 2009.

[7] M. Åsberg, T. Nolte, and P. Pettersson, "Prototyping and Code Synthesis of Hierarchically Scheduled Systems using TIMES," *Journal of Convergence (Consumer Electronics)*, vol. 1, no. 1, pp. 77–86, 2010.

[8] M. Åsberg, P. Pettersson, and T. Nolte, "Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling," in *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, 2011.

[9] ARINC, *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), 1996.

[10] ARINC/RTCA-SC-182/EUROCAE-WG-48, "Minimal Operational Performance Standard for Avionics Computer Resources." RTCA, Incorporated, 1828 L Street, NW, Suite 805, Washington D.C. 20036, 1999.

[11] S. Kato, R. Rajkumar, and Y. Ishikawa, "A Loadable Real-Time Scheduler Suite for Multicore Platforms," Technical Report CMU-ECE-TR09-12, 2009. [Online]. Available: http://www.contrib.andrew.cmu.edu/~shinpei/papers/techrep09.pdf

[12] Editors: T. Maragria and B. Steffen, "Leveraging Applications of Formal Methods," *Proc. of the 1st International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Springer*, pp. 140–141, 2004.

[13] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems," in *Proc. of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2010.

[14] M. Åsberg, T. Nolte, C. M. O. Perez, and S. Kato, "Execution Time Monitoring in Linux," in *Proc. of the W.I.P. session in the 14th International Conference on Emerging Technologies and Factory Automation*, 2009.

[15] C. Liu and J. Layland, "Scheduling Algorithms for Multi-Programming in a Hard-Real-Time Environment," *ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[16] S. Kato, R. Rajkumar, and Y. Ishikawa, "AIRS: Supporting Interactive Real-Time Applications on Multicore Platforms," in *Proc. of the 22nd Euromicro Conference on Real-Time Systems*, 2010.

[17] T. Bird, "Measuring Function Duration with Ftrace," in *Proc. of the Japan Linux Symposium*, 2009.

[18] H. Thane and H. Hansson, "Using Deterministic Replay for Debugging of Distributed Real Time Systems," in *Proc. of the 12th Euromicro Conference on Real-Time Systems*, 2000.

[19] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson, "Replay Debugging of Real-Time Systems Using Time Machines," in *Proc. of the 17th International Parallel & Distributed Processing Symposium*, 2003.

[20] D. Faggioli and F. Checconi, "An EDF Scheduling Class for the Linux Kernel," in *Proc. of the 11th Real-Time Linux Workshop*, 2009.

[21] M. Åsberg, J. Kraft, T. Nolte, and S. Kato, "A Loadable Task Execution Recorder for Linux," in *Proc. of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2010.

[22] B. Cantrill, M. Shapiro, and A. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proc. of the USENIX conference*, 2004.

[23] V. Prasad, W. Colhen, F. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating System Problems Using Dynamic Instrumentation," in *Proc. of the Ottawa Linux Symposium*, 2005.

[24] K. Yaghmour and M. Dagenais, "Measuring and Characterizing System Behavior Using Kernel-Level Event Logging," in *Proc. of the USENIX conference*, 2000.

[25] M. Desnoyers and M. Dagenais, "The LTTng Tracer: A Low Impact Performance and Behavior Monitor of GNU/Linux," in *Proc. of the Ottawa Linux Symposium*, 2006.

[26] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza and S. Papacharalambous, "RTAI: Real Time Application Interface," *Linux Journal*, vol. 29, no. 10, 2000.