Towards Partitioned Hierarchical Real-Time Scheduling on Multi-core Processors*

Mikael Åsberg
Mälardalen Real-Time Research Center
Mälardalen University
Västerås, Sweden
mikael.asberg@mdh.se

Thomas Nolte
Mälardalen Real-Time Research Center
Mälardalen University
Västerås, Sweden
thomas.nolte@mdh.se

Shinpei Kato
Graduate School of Information Science
Nagoya University
Nagoya, Japan
shinpei@is.nagoya-u.ac.jp

ABSTRACT

This paper extends previous work on hierarchical scheduling to multi-core systems. We have implemented partitioned multi-core scheduling of servers in the Linux kernel, using the scheduling framework ExSched. Neither ExSched nor the presented scheduler require any modifications to the Linux kernel. Hence, this makes the installation and kernel-version updates easier. We also present a user-space simulator which can be used when developing new multi-core hierarchical schedulers (plug-ins) for ExSched.

We evaluate the overhead of our new multi-core hierarchical scheduler and compare it to a single-core hierarchical scheduler. Our results can be useful for developers that want to minimize the scheduler overhead when using partitioned hierarchical multi-core scheduling.

Keywords

real-time systems, partitioned multi-core scheduling, hierarchical scheduling, implementation

1. INTRODUCTION

Introduction Hierarchical scheduling is a general term for composing applications as well defined components, possibly in a hierarchical manner with one or more levels. Software which is structured in such a way is more robust than flat system since defects will only affect a delimited part of the system. The isolation that comes with hierarchical scheduling will also make software reuse more simple. This powerful mechanism has been adopted by the avionics industry in form of the ARINC653 [3] software specification. ARINC653 isolates applications in terms of both the CPU and the memory. Hence, hierarchical scheduling can be used in safety-critical systems to make them more safe. However, hierarchical scheduling can also be used in soft real-time systems as well. For example, it can be used to guarantee CPU-cycle reservations for time-sensitive software such as video or music applications on smartphones and tablets [10].

The advent of multi-core processors will guarantee a steady improvement of the CPU performance over time. However, multi-core processors also make application development more difficult if the goal is to make use of all the extra CPU performance that multi-core brings. For example, shared resources [13] and cache memory [2] are two challenges that the real-time community are faced with. However, adapting to multi-core is the only option if the objective is to maximize the CPU performance of todays processors.

Problem statement Streaming and decoding are common CPU-intensive tasks in Linux-based systems such as Android devices, and they affect the user perception of performance to a high degree. Hierarchical scheduling is useful for controlling CPU reservations for these kind of applications and hence helps in improving the user experience. Android devices are usually equipped with multi-core processors which demand adaptions to hierarchical scheduling.

The solution must be as non-intrusive as possible since our target product constantly needs software updates. Avoiding kernel modifications simplifies porting efforts when switching kernels.

Performance overhead of the solution must of course be as small as possible since embedded systems such as Android devices have very limited resources.

Contribution In this paper we present an adaptation of the hierarchical scheduling technique in Linux to multi-core processors. The work is based on the scheduler framework ExSched [5]. We present a new Fixed-Priority Preemptive Scheduling (FPPS) scheduler plug-in called `partitioned-hsf-fp` which is the 6th scheduler plug-in available for ExSched. It comes with a simulator that can simulate hierarchical multi-core scheduling. `partitioned-hsf-fp` can be executed in both the Linux kernel as well as in our new simulation tool. We also present an experimental evaluation of the `partitioned-hsf-fp` scheduler.

Outline The outline of this paper is as follows: In Section 2 we present the preliminaries and Section 3 lists the related work in this area of research. Further, Section 4 describes the scheduler implementation and the simulation tool. In Section 5 we evaluate the overhead of our new sched-

*The work in this paper is supported by the Swedish Research Council, and Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VtRes '13 Taipei, Taiwan
Copyright 2014 ACM Copyright is held by the authors ...$15.00.

uler. Finally, Section 6 concludes our work.

2. PRELIMINARIES

2.1 Hierarchical scheduling

The Hierarchical Scheduling Framework (HSF) is illustrated in Figure 1. Observe that there are two types of schedulers; one global and one or several local schedulers. The global scheduler is in charge of scheduling servers and the local scheduler handles the scheduling of tasks. The scheduling algorithm at any level can be arbitrary. However, in this paper we assume FPPS at the global level.

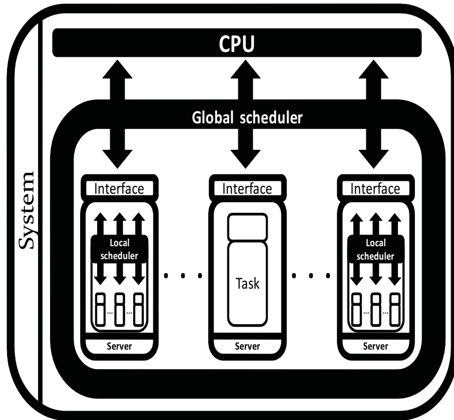

Figure 1: Hierarchical scheduling framework.

The server interface defines the amount of CPU that will be reserved for the particular server. It is usually a time window (referred to as *budget*) that re-occurs at a specific interval of time called *period*, i.e., similar to the the periodic task model [11]. Figure 2 illustrates 3 running servers (A, B and C) which are scheduled using FPPS (without local schedulers). The servers are re-started periodically (illustrated with the arrows). The budget is illustrated with a dotted line that surrounds the task. Server A has the highest priority, B has the middle priority and C has the lowest priority. The order of the execution of the servers is affected by the priorities. Each server could potentially host more than one task each.

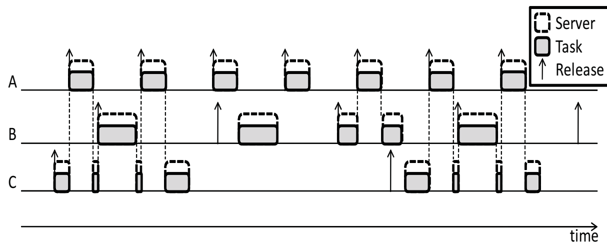

Figure 2: Example trace of a hierarchically scheduled system.

2.2 Multi-core scheduling

Multi-core scheduling can essentially be divided into two categories: *global scheduling* and *partitioned scheduling*.

Global scheduling implies that a scheduling object (task, server etc.) can be scheduled on any CPU core at any time. The term *migration* is used when a scheduling object moves from one CPU core to another during its execution.

Partitioned scheduling defines that every scheduling object is allocated to one specific CPU core and never executes on any other core. This type of scheduling resembles uni-core scheduling the most since it can be seen as multiple instances of uni-core scheduling. It is easier to implement partitioned scheduling (compared to any other multi-core scheduling technique) but it is also inferior in performance compared to the other multi-core scheduling techniques. Partitioned scheduling cannot utilize all the CPU cores as effective as for example global scheduling [5, 6]. Our extension to hierarchical scheduling uses partitioned multi-core scheduling.

2.3 ExSched

The ExSched [5] framework is based on a user-space library and a kernel module as illustrated in Figure 3. The framework does not require any modifications to the kernel. The porting effort to new kernels is (usually) at most a couple of small source-code modifications in ExSched.

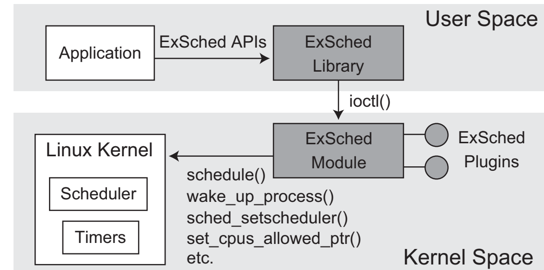

Figure 3: The ExSched framework.

User space applications (tasks) can be scheduled by the scheduler plug-ins that are available in ExSched. Scheduler plug-ins use basic scheduler primitives exported from the ExSched kernel module (such as timers etc.). The ExSched kernel module uses native Linux kernel primitives and exports a simplified interface to scheduler plug-in developers. The ExSched module implements FPPS and Earliest Deadline First (EDF) as base (task) scheduler policies. Plug-ins can extend these base scheduling policies with new policies (for example hierarchical or multi-core scheduling) or simply disable them if they are not useful. ExSched is available for both Linux and VxWorks and it has 3 multi-core scheduler plug-ins and 2 hierarchical scheduler plug-ins available (and the new `partitioned-hsf-fp` scheduler plug-in presented in this paper).

Figure 4 illustrates the flow of function calls when ExSched suspends and releases a task. A task can suspend itself by calling the ExSched API function `rt_wait_for_period()` which executes in the ExSched kernel module via ioctl. Any active scheduler plug-in will get notified by such an event through the callback function `job_complete_plugin()`. The ExSched module then calls the Linux function `mod_timer()` to setup an interrupt that will call an ExSched interrupt handler called `job_release()` (the time until the interrupt occurs will be equal to the tasks period). After that, ExSched calls the Linux primitive `schedule()` which will schedule

another task (if any active task exists). The ExSched interrupt handler `job_release()` will notify any scheduler plug-in that a task has been released via the callback function `job_release_plugin()`. Finally, ExSched will call the Linux function `wake_up_process()` in order to activate the task. The Linux scheduler will decide if a context switch should occur or not. This depends on the status of the Linux task ready-queue.

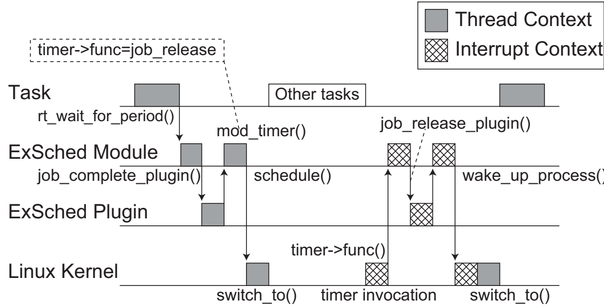

Figure 4: The flow of function calls in ExSched.

3. RELATED WORK

To the best of our knowledge there exists only two papers that present hierarchical scheduler implementations with multi-core support ([7] and [8]). Our work differs in that we use fixed priorities and our solution does not modify the Linux kernel.

Checconi et al. [7] presented a scheduler (which modified the Linux kernel) that included Hard Constant Bandwidth Servers (H-CBS). These servers are scheduled on each core and do not migrate. However, the tasks inside a server can migrate to another server that resides on another core. Hence, a server has a fraction of each core at its disposal. The SCHED_DEADLINE scheduler project [8] is responsible of the EDF scheduler implementation in Linux. The scheduler modifies the Linux kernel and supports the scheduling of servers. SCHED_DEADLINE has the highest priority among the scheduling classes in Linux and it supports partitioned multi-core scheduling (one runqueue per CPU).

$Litmus^{RT}$ [6] is an experimental platform which modifies the Linux kernel (latest version is 3.0). $Litmus^{RT}$ provides a simplified scheduling interface for multi-core scheduler development.

Recently, Mollison et al. presented a multi-core scheduling framework (without the support of hierarchical scheduling) in Linux which only executes in user space [12].

The AQuoSA framework [14] is a hierarchical scheduler based on CBS [1]. It also supports adaptive resource reservations. The framework modifies the Linux kernel and exports scheduling hooks to the user.

4. IMPLEMENTATION

This section will give a short description of the `partitioned-hsf-fp` scheduler implementation and our new simulation tool. Both ExSched (together with `partitioned-hsf-fp` and the other 5 plug-ins) and the simulator are available for free as an open source project.

4.1 Partitioned multi-core hierarchical scheduler

Figure 5 shows the main functions of the uni-core HSF scheduler. This scheduler has been extended for multi-core platforms by implementing partitioned scheduling. This means that servers (and all of its tasks) run on one specific core and do not migrate to other cores. The ExSched kernel module handles task scheduling inside servers.

```
1: void job_release_plugin(struct exsched_task_struct *rt) {
2:     .
3: }
4: void job_complete_plugin(struct exsched_task_struct *rt) {
5:     .
6:}
7: void job_init_plugin(struct exsched_task_struct *rt) {
8:     .
9:}
10: void server_release_handler(unsigned long _data) {
11:    .
12:}
13: void server_complete_handler(unsigned long _data) {
14:    .
15:}
```

Figure 5: Skeleton code of the `hsf-fp` scheduler.

The `job_init_plugin` function (Figure 5) executes once for every task before it starts to execute. The function registers a task to a specific server (which is chosen at line (3) in Figure 6) and migrates the task to core number 0. We extend this function by migrating the task to the same core that its server is assigned to.

The `job_release_plugin` and `job_complete_plugin` functions are callback functions that relate to the ExSched kernel module. These functions are executed whenever a task gets released (by the ExSched kernel module) or finishes (by calling `rt_wait_for_period` at line (7) in Figure 6). These two functions originally used one server ready-queue. The `job_complete_plugin` used the ready-queue mainly for error checking. The `job_release_plugin` used the server ready-queue in order for it to prevent task releases whenever they occured outside of its server budget (hence the task release would be postponed until its server started to execute). However, both `job_complete_plugin` and `job_release_plugin` has been modified since there are multiple server ready-queues when partitioned scheduling is used.

```
1: main(timeval C, timeval T, timeval D, int prio, int nr_jobs) {
2:         .
3:         rt_set_server(1);
4:         rt_run(0);
5:         for (i = 0; i < nr_jobs; i++) {
6:                 /* User's code. */
7:                 rt_wait_for_period();
8:         }
9:         rt_exit();
10:}
```

Figure 6: Example of a task using the ExSched API.

The `server_release_handler` and `server_complete_handler` functions are interrupt handlers that are responsible for releasing servers (and their tasks), depleting servers (and suspending their tasks) and performing server context switches. These two server handlers (lines (10) and (13) in Figure 5)

are duplicated to the same amount of CPU cores, i.e., if the platform has two cores then `partitioned-hsf-fp` uses two release and two complete handlers. This means that server release and ready queues are also duplicated. Hence, the amount of server related interrupts will increase as the number of cores increase. However, it is possible to only use two interrupt handlers (sharing one interrupt signal) but we defer such an implementation for future work. The implementation of the two handlers would be more complex but the overhead would most likely decrease since the interrupt context-switch and handler overhead would decrease.

The ExSched kernel module does not need any extensions since it already supports multi-core scheduling. However, our hierarchical scheduler recorder [4] was modified in order to support tasks running on different cores.

4.2 Multi-core scheduler simulator

Our simulator is useful when developing new hierarchical multi-core scheduler plug-ins for ExSched. The simulator executes as a Linux process and supports both uni-core and multi-core since we map each interrupt signal to one process signal, i.e., similar to how WindRiver (VxWorks) simulators work. The current version of our simulator can simulate server-scheduling related interrupts and hence it can also record the execution of servers running in parallel. Such a recording can later be viewed graphically using tools such as Grasp [9] for example. Having the possibility to view an execution trace makes scheduler debugging easier. The `partitioned-hsf-fp` scheduler was developed rapidly using the simulator. The simulator can be downloaded together with the ExSched package.

5. EVALUATION

This section presents our experiments with the `partitioned-hsf-fp` implementation. We will present the overhead measurements of the `partitioned-hsf-fp` and the `hsf-fp` scheduler and compare them. We also elaborate on the factors that affect the scheduling overhead in partitioned multi-core scheduling. Further, we also provide execution traces of tasks and servers scheduled by the `partitioned-hsf-fp` and `hsf-fp` schedulers and we visualize them using the Grasp tool [9].

5.1 Hardware and software setup

We conducted the experiments on an Intel Pentium Dual-Core (E5300 2,6GHz) platform. The platform was equipped with a Linux kernel (version 2.6.31.9) configured with *load balancing* disabled (since we use partitioned multi-core scheduling). We assume Rate Monotonic (RM) priority assignment in all of the experiments.

5.2 Overhead measurements

In the first experiment we executed the `partitioned-hsf-fp` and `hsf-fp` scheduler with 2-10 servers (one task per server). The total server CPU utilization in all server configurations was below 50% in order for it to be schedulable with the `hsf-fp` scheduler (which had 100% CPU utilization while `partitioned-hsf-fp` had 200%). We did an equal allocation (50/50) of servers to core 0 and 1 in the case of `partitioned-hsf-fp`. The server-to-CPU allocation strategy was random, i.e., we did not follow best-fit, worse-fit etc. Figure 7 shows the measured overhead results. The overhead only includes the execution time of the server interrupt-handlers (lines (10) and (13) in Figure 5) and the task callback functions (lines (1) and (4) in Figure 5). Hence, we do not include the overhead of the ExSched kernel module nor Linux-related system overheads such as interrupt context-switch overhead etc. Each presented measurement value is the average of 10 sampled measurement values. We ran the server systems for 500 jiffies (in our platform this corresponds to 2 seconds). We can observe that the `partitioned-hsf-fp` scheduler generates more overhead than `hsf-fp` in most server configurations (except for 3 and 5 servers). We would expect that the overhead of `partitioned-hsf-fp` and `hsf-fp` should be equal since we do not include interrupt context-switch overhead in our measurements.

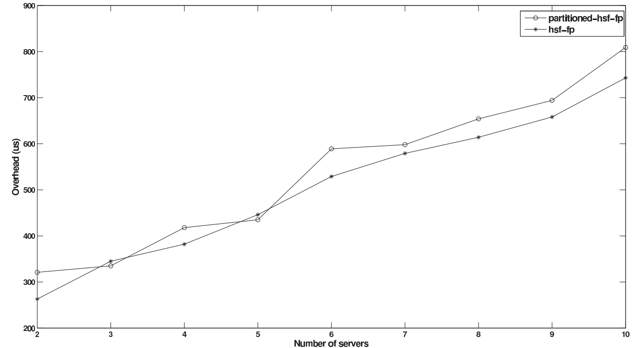

Figure 7: Average overhead measurements of the `partitioned-hsf-fp` and `hsf-fp` schedulers.

The conclusion is that the overhead is not the same in all server configurations, hence, the next question arises: how come that the overhead differentiates between `partitioned-hsf-fp` and `hsf-fp`? In order to answer this question we ran a set of tests and the results are presented in Table 1. The two most interesting tests included a server release and preemption test. Each presented measurement value is the average of 10 sampled measurement values.

We ran 10 servers for 4 seconds, configured with the same period (**Release test**, Table 1). The `partitioned-hsf-fp` scheduler had a simple 50/50 allocation strategy. The results show a difference of 78 us. The reason is because two separate interrupt handlers execute at every server release in case of `partitioned-hsf-fp`. Using `hsf-fp` results in half as many executions of the server interrupt handler. However, the amount of executed instructions in the server interrupt handler should almost be the same in both `partitioned-hsf-fp` and `hsf-fp`. This is an extreme case but we want to demonstrate that scheduling overhead can be reduced when scheduling events coincide together in a single execution of an interrupt handler.

In the second test (**Preemption test**, Table 1) we ran two servers (for 4 seconds) with one task each. The servers had a large difference in period and budget values in order to increase the amount of server context-switches when scheduled with `hsf-fp`. The difference in overhead was 169 us. This extreme case also shows that the difference in the amount of server context-switches affects the difference in overhead. The execution trace of the two servers is shown in Figure 8 and 9. Figure 8 shows that server $S1$ gets preempted frequently by server $S0$ (since server $S0$ has a higher priority

than server $S1$) while the 2 core setup (Figure 9) avoids this since the two servers are allocated onto different cores. Task $S0_task$ was allocated to server $S0$ and task $S1_task$ was allocated to server $S1$. Each core had one idle task and one idle server running in the background (they had the lowest priority).

Experiment type	partitioned-hsf-fp (us)	hsf-fp (us)
Release test	839	761
Preemption test	1134	1303

Table 1: Overhead comparisons.

The conclusion is that the user has the possibility to minimize the scheduler overhead by selecting an optimal server allocation when using partitioned hierarchical scheduling. The two strategies that we can recommend is to coincide scheduling events (server releases and depletions) and minimize server context-switches. The latter can be done by categorizing the servers by periods and then group together servers with equal or almost equal period values on each core. It is most likely that these kind of events affect the overhead differences the most between `partitioned-hsf-fp` and `hsf-fp` scheduling as shown in Figure 7. The overhead of `hsf-fp` is probably lower due to the coinciding scheduling events. However, Figure 7 shows that `partitioned-hsf-fp` has lower overhead in case of 3 and 5 servers. It is most likely due to a lower amount of server context-switches, i.e., similar to the scenario presented in Figure 8 and 9.

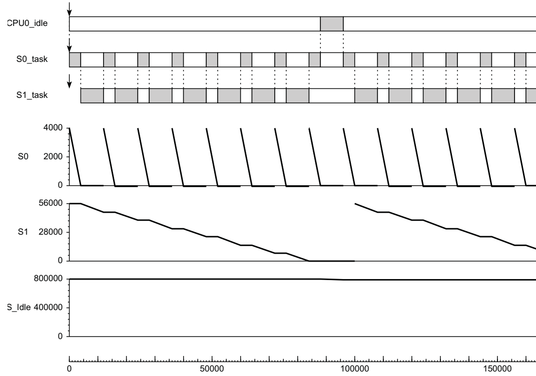

Figure 8: Execution trace of the `hsf-fp` scheduler.

We ran 16 servers ($S0$-$S15$, Table 2) in our last experiment using the `partitioned-hsf-fp` scheduler and two different CPU allocation strategies. Each server had one task each and we ran the experiments for 4 seconds. A lower priority value implies a higher priority. The motivation with this experiment was to show how much the overhead differed depending on how the servers were allocated to the CPUs. The first allocation strategy (**Categorized by period**, Table 3) partitioned servers $S0$-$S7$ on core 0 and the rest on core 1. The second allocation method (**Mixed**, Table 3) simply mixed servers $S0$, $S2$, $S4$, $S6$, $S8$, $S10$, $S12$ and $S14$ together and allocated them on core 0, and the rest ($S1$, $S3$, $S5$, $S7$, $S9$, $S11$, $S13$ and $S15$) on core 1.

Table 3 shows that the **Mixed** allocation generated about 40% more overhead using the `partitioned-hsf-fp` scheduler compared to the other strategy (**Categorized by period**) which allocated the servers based on period values (small period values on core 0 and large period values on core 1). This example shows how much the overhead can differ depending on how the servers are allocated onto the CPU cores. Note also that we do not include Linux system overhead (task context switches etc.) in our measurements. Hence, the actual overhead might be significantly larger than the presented results.

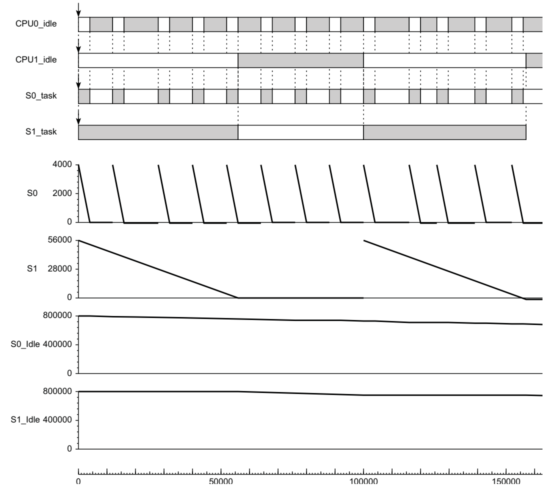

Figure 9: Execution trace of the `partitioned-hsf-fp` scheduler.

Server	Period (jiffies)	Budget (jiffies)	Priority
S0	6	1	0
S1	7	1	1
S2	8	1	2
S3	9	1	3
S4	10	1	4
S5	11	1	5
S6	14	1	6
S7	18	1	7
S8	125	15	8
S9	125	15	9
S10	125	15	10
S11	125	15	11
S12	125	15	12
S13	125	15	13
S14	125	15	14
S15	125	15	15

Table 2: Server parameters.

6. CONCLUSION

We have presented an implementation of a multi-core scheduler in Linux called `partitioned-hsf-fp` and a user-space simulator which can be used for developing multi-core schedulers. `partitioned-hsf-fp` is based on an already existing hierarchical fixed-priority preemptive scheduler (`hsf-fp`) in the scheduler framework ExSched. `partitioned-hsf-fp` is a hierarchical scheduler which can run multiple servers in

Allocation type	Overhead (us)
Categorized by period	1502
Mixed	2129

Table 3: Measured overhead using different CPU allocation methods.

parallel using partitioned multi-core scheduling. The servers and their corresponding tasks run on one core only and do not migrate to other cores. We have shown with experiments that the overhead of `partitioned-hsf-fp` is slightly higher than `hsf-fp`. We have identified the main sources of overhead (server releases and context-switches) through experiments and presented recommendations on how to decrease the overhead when using partitioned hierarchical multi-core scheduling. We have also shown example execution traces when running the `partitioned-hsf-fp` scheduler in a Linux kernel. Our last experiment showed that the scheduler overhead of `partitioned-hsf-fp` differed approximately 40% when we used different CPU-core allocation methods for servers.

The overhead of `partitioned-hsf-fp` is similar to the single core version (with the assumption that the interrupt context-switches do not affect the multi-core version significantly). We believe that allocation strategies could decrease the scheduler overhead. However, core allocation is also affected by other factors like shared resources between tasks. Hence, there are many different factors to consider when allocating tasks and servers to cores. Another way to decrease the overhead is to implement `partitioned-hsf-fp` with one interrupt signal instead of using one per core. We defer such implementation to future work. We will also look into using global, clustered and semi-partitioned scheduling combined with hierarchical scheduling.

ExSched and the `partitioned-hsf-fp` scheduler has a high potential when it comes to resource reservation problems when running for example media applications in Android systems (smartphones and tablets). The user experience of media streaming could be improved by using `partitioned-hsf-fp`. Especially since todays media devices are equipped with more than one core. The advantage with using ExSched is that there is no need to modify the Linux kernel in Android systems. Hence, it is easy to keep up with the newest kernel release without having to do tedious modifications to the kernel every time it is updated. Keeping up with the newest kernel release is important since new Linux kernel versions are released several times per year.

7. REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium (RTSS)*, 1998.

[2] B. Åkesson, A. Molnos, A. Hansson, J. Ambrose Angelo, and K. Goossens. Composability and Predictability for Independent Application Development, Verification, and Execution. In *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*. Springer, Dec 2010.

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

[4] M. Åsberg, T. Nolte, and S. Kato. A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux. In *Proc. of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.

[5] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar. ExSched: An External CPU Scheduler Framework for Real-Time Systems. In *Proc. of the 18th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[6] B. Brandenburg, J. Calandrino, and J. Anderson. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In *Proc. of the 29th Real-Time Systems Symposium (RTSS)*, 2008.

[7] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari. Hierarchical Multiprocessor CPU Reservations for the Linux Kernel. In *Proc. of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2009.

[8] D. Faggioli, M. Trimarchi, and F. Checconi. An Implementation of the Earliest Deadline First Algorithm in Linux. In *Proc. of the 24th Annual ACM Symposium on Applied Computing (SAC)*, 2009.

[9] 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 (WATERS)*, 2010.

[10] A. Kandhalu and R. Rajkumar. Energy-Aware Resource Kernel for Smartphones and Tablets. In *Proc. of the 33rd Real-Time Systems Symposium (RTSS), Demo Track*, 2012.

[11] C. Liu and J. Layland. Scheduling Algorithms for Multi-Programming in a Hard-Real-Time Environment. *ACM*, 20(1):46–61, 1973.

[12] M. Mollison and J. Anderson. Bringing Theory into Practice: A Userspace Library for Multicore Real-Time Scheduling. In *Proc. of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[13] F. Nemati. *Resource Sharing in Real-Time Systems on Multiprocessors*. PhD thesis, Mälardalen University, May 2012.

[14] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA—adaptive quality of service architecture. *Softw. Pract. Exper.*, 39(1):1–31, 2009.