Towards Hierarchical Scheduling in Linux/Multi-core Platform

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

Shinpei Kato
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku,
Tokyo 113-8656, Japan
shinpei@il.is.s.u-tokyo.ac.jp

***Abstract*—This paper proposes the implementation of 4 different scheduling strategies for combining multi-core scheduling with hierarchical scheduling. Three of the scheduling schemes are analyzable with state-of-the-art schedulability analysis theory, available in the real-time systems community. Our idea is to implement these hierarchical multi-core scheduling strategies in a Linux based operating system, *without modifying the kernel*, and evaluate them. As of now, we have developed/implemented a prototype two-level hierarchical scheduling framework (HSF) in Linux (uni-core), which supports fixed priority preemptive scheduling (FPPS) of periodic servers at the top level, and FPPS of periodic tasks at the second level. The HSF is based on the REal-time SCHeduler (RESCH) framework.**[1]

I. Introduction

Our ongoing research is highly focused on the implementation of hierarchical scheduling [1] and execution time monitoring (in Linux) for hierarchical/flat scheduler debugging [2], [3]. Although hierarchical scheduling has major strengths which have been shown in industry, for example the ARINC653 standard inherent in the integrated modular avionics architecture, and considering it is a relatively old technique, it is rarely found integrated in General Purpose Operating Systems (GPOSs) intended for the embedded systems market. Linux has recently included real-time group scheduling (server-based scheduling), i.e., *control groups* and *sched-rt-group* which makes it possible to schedule groups of tasks. The interface given to each group is a period and runtime, specifying that each group may run a certain amount of time (runtime) each period. One motivation for introducing server-based (hierarchical) scheduling in a GPOS, such as Linux, is because of the possibility to allocate a specific amount of CPU power to an application that may consist of many Linux processes. Multimedia intensive applications will gain performance due to this kind of scheduling (since the CPU availability for video/audio processing increases) which directly affects the users media quality experience. An interesting example is when you execute a process which processes a movie, using, e.g., the VLC library or similar. Running it (as a real-time task) with high priority will affect all lower priority real-time tasks and non real-time tasks. Lowering its priority will result in an unexpected degradation of CPU time (depending on the priority, type and amount of running real-time tasks). The problem is that it is not possible to fine-tune its performance by setting a frequency at which it will get a specified amount of CPU time, since it is a aperiodic task.

We have successfully implemented a prototype hierarchical scheduler with the use of RESCH [4]. This framework supports fixed priority preemptive scheduling (FPPS) of periodic (real-time) Linux tasks. The RESCH scheduler is a loadable kernel module which only uses exported Linux kernel primitives, as a means to affect scheduling. RESCH itself, and our HSF, is completely free of kernel modifications (no patches). This complies with the requirements in the area of embedded systems where reliability and stability of software are important properties, hence, proven versions of Linux are therefore preferred. In this way, our scheduling framework adapts a high degree of usability (since it does not require patches), and it is more flexible than the built-in group scheduling of Linux, such as *control groups* and *sched-rt-group*, since our scheduler is highly adaptable. For example, our current server scheduler supports periodic server scheduling, but could easily be extended to support Constant Bandwidth Server (CBS), Sporadic Server (SS) etc.

A major drawback with the current version of our HSF is that it only supports uni-core scheduling. This makes it inefficient since it cannot utilize the entire CPU capacity of the multi-core platforms. Hence, we want to extend our hierarchical scheduler combining hierarchical scheduling with different types of multi-core scheduling schemes.

A. Preliminaries

a) RESCH: As mentioned previously, RESCH is a patch-free scheduling framework for Linux. It supports periodic tasks which can be scheduled in a fixed-priority preemptive manner. RESCH runs as a kernel module (in kernel space), giving both an interface to users in user space (e.g. a task specific interface like rt_wait_for_period()), as well as in kernel space. The kernel space primitives (hooks) can be implemented by a **RESCH plugin**, i.e., a kernel module that has access to the RESCH kernel API. Examples of such hook functions are job_release_plugin and job_complete_plugin, which

[1]The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

are executed when a RESCH task is released for execution respectively when it has completed its current job.

In Linux, since kernel version 2.6.23 (October of 2007), tasks can be either a **fair** or a **real-time**. 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. The cost of having a patch-free solution is that it can only see scheduling events related to its registered tasks, i.e., higher priority real-time tasks, which are not registered in RESCH, can thereby interfere with RESCH tasks without the RESCH core detecting it. A simple solution to this problem is to schedule all real-time tasks with RESCH.

b) Multi-core scheduling: There exits three types multi-core scheduling schemes [5], [6], [7], [8]. **Global scheduling** is referred to when a single scheduler (with a single task ready queue) schedules tasks on all cores on the platform. In this type of scheduling, tasks may switch (migrate) to other cores. **Partitioned scheduling** is defined as multiple schedulers (one per core) scheduling tasks on only one core. Tasks are statically assigned to a CPU and they do not migrate. The third approach, **Cluster scheduling**, is a combination of the two mentioned. It may have at minimum one scheduler (and maximum equal to the amount of cores) and each scheduler is assigned a subset of cores (minimum 1 and maximum the number of cores on the platform). Each scheduler schedules its tasks on its assigned cores, note that there may exist global and partitioned scheduling in this scheme.

c) HSF: As mentioned previously, we have implemented a hierarchical scheduler (HSF) plugin in RESCH. HSF schedules RESCH tasks in groups, where a group (server) is scheduled in FPPS, with respect to its interface: $period$, $budget$ and $priority$. A server runs $budget$ time units every $period$, and in the order of their $priority$, i.e., FPPS. Inside each server, RESCH schedules the group of tasks according to FPPS. An example execution trace of HSF/RESCH is shown in Fig. 1. We ran a movie processing (aperiodic) task (rt_vlc) in server $Server0$, and 2 periodic real-time tasks (rt_task1 and rt_task2) in server $Server1$. $Server0$ had highest server priority (and high frequency), leading to that rt_vlc interfering with task rt_task1 and rt_task2 very frequent. We ran these tasks/servers on a desktop computer (Intel Pentium Dual-Core, E5300 2,6GHz) equipped with Ubuntu Linux (kernel version 2.6.31.9). The arrows represent task releases and the trace itself is visualized with the Grasp tool [9].

B. Outline

The outline of this paper is as follows: Section II presents related work. In Section III we propose the implementation of 4 different hierarchical multi-core scheduling schemes, and finally in Section IV, we discuss the complexity of these schemes.

II. Related work

To the best of our knowledge, related to hierarchical scheduling implementation in combination with multi-core and Linux for real-time systems, there is only one paper presenting such work [10]. The scheduler (patch) is composed of one Hard Constant Bandwidth Server (H-CBS) per core, and global multi-core scheduling within each partition, which has a fraction of each core at its disposal. Further, theoretical work in this area [11] has been done by Shin et al. The difference from [10] is that there is only one hierarchical global scheduler and a partition might not have access to all cores. The hierarchical global scheduler is defined to be Global Earliest Deadline First (G-EDF) as well as the local ones.

Hierarchical scheduler implementations in Linux are, to the best of our knowledge, all based on modifications of the Linux kernel. Here are some examples.
The SCHED_DEADLINE project [12], is in charge of the Earliest Deadline First (EDF) scheduler implementation for Linux. The scheduler (patch) supports scheduling of servers which has a period, budget and a task associated with it. SCHED_DEADLINE is the highest priority scheduling class, i.e., higher than SCHED_FAIR and SCHED_RT and it supports multi-core (one runqueue per CPU).

Real-Time Application Interface for Linux (RTAI) [13] is a collection of loadable kernel modules and a kernel patch which together provides a rich real-time scheduling API to the user. It does not implement a hierarchical scheduler but it exports the sufficient primitives to implement one. Similar to RTAI, RT-Linux [14] is a patch to the Linux kernel and introduces a layer between the OS and the hardware. Linux is scheduled as low priority task while real-time tasks have higher priority.

The AQuoSA framework [15] is a hierarchical scheduler based on the Constant Bandwidth Server (CBS) with a advanced adaptive resource reservation. The framework is built on a patch which exports appropriate scheduling hooks.

A POSIX compliant (minimally invasive) implementation of the sporadic server (SCHED_SPORADIC) [16] is presented in [17]. The sporadic server improves the response time of aperiodic tasks within fixed priority scheduling.

Related to multi-core scheduling in Linux, $Litmus^{RT}$ [18] is an experimental platform which extends (modifies) the Linux kernel (latest version is 2.6.32). It provides a simplified scheduling interface for multi-core scheduling development.

III. Proposed implementations

There exists (to the best of our knowledge) three known hierarchical multi-core scheduling schemes for real-time systems, where one of these has been implemented in Linux. Our aim is to implement all 3 of these in Linux, based on our HSF/RESCH scheduler implementation, i.e., without kernel patches. The current HSF implementation could also be evaluated by comparing it with existing Linux hierarchical schedulers such as the AQuoSA framework [15]. These are the 4 hierarchical multi-core scheduling schemes that we intend to

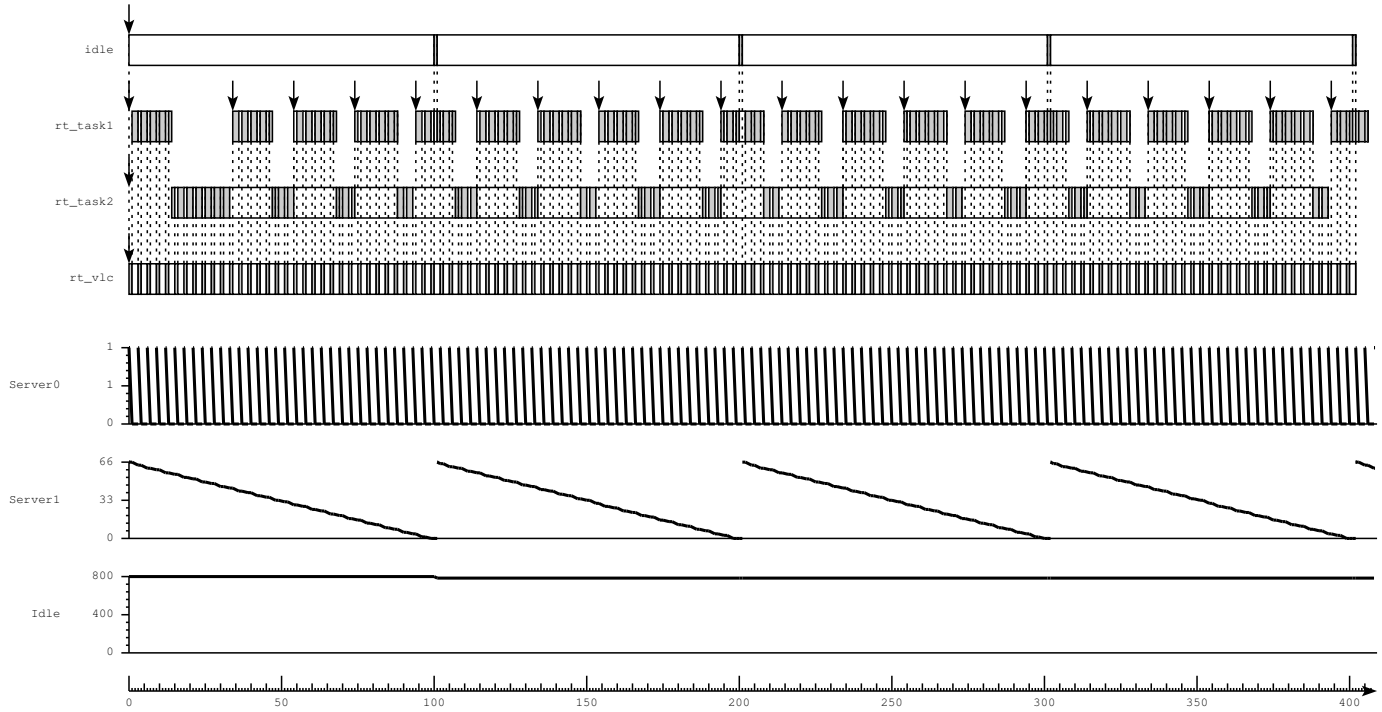

Fig. 1. Example trace visualized with the Grasp tool

implement: Shin et al. [11] has proposed analysis for cluster-based scheduling. In this setup, tasks are scheduled with G-EDF at the local level, i.e., the ready queue is ordered by shortest deadline and tasks are transferred to the different cores until they are all occupied. At the global level, each subsystem, i.e., group of tasks and servers are assigned a subset of the cores. The global scheduler schedules the subsystems servers based on G-EDF and the assigned cores. The subsystem has the same amount of servers as the amount of assigned cores, and all servers have the same period, but the budget is distributed among them. This distribution is not fair, e.g., 3 servers with budget 2.5 will result in budgets 1, 1 and 0.5. In Checconi et al. [10] (has an implementation in Linux), each subsystem will have access to all cores, and the number of servers (in each subsystem) are the same as the number of cores. Locally, tasks are scheduled with global multi-core fixed priority, globally, each core has its own H-CBS scheduler, scheduling one server from each subsystem. Note that the global scheduling is done in parallel (several H-CBS schedulers). Nemati et al. [19] has a scheme where the servers are scheduled with a global multi-core scheduling scheme (fixed or dynamic priority), and locally, each subsystem is scheduled with partitioned multi-core (fixed or dynamic priority) scheduling, i.e., each subsystem has maximum one server (that may run on any core), so tasks always execute on the same core. Yet a fourth scheme could be to schedule servers in sequence, thereby scheduling tasks, within each subsystem, with global multi-core on all cores. The difference from [10] is that there is only one global scheduler. All 4 schemes are summarized in Table I.

Strategy	Multi-core scheduling	Server parallelism
Shin et al. [11]	Cluster	Yes
Checconi et al. [10]	Global	Yes
Nemati et al. [19]	Partitioned	Yes
Sequential	Global	No

TABLE I
HIERARCHICAL MULTI-CORE STRATEGIES

Fig. 2 illustrates how the servers would be executed in the 4 different schemes. In Shin et al. [11], the servers of a subsystem typically occupies a subset of all cores at a time (depending on the priority of other subsystems and its assigned number of cores). All subsystem server periods are the same, but the budget may differ, which is illustrated in the figure. Checconi et al. [10] differs in that there is one scheduler per core, scheduling independently of one another. A subsystem has one server per core with the same parameters. Fig. 2 illustrates that the schedule will look similar on all cores, but asynchronous, if we assume different scheduler drift or similar interference. In Nemati et al. [19], there is maximum one subsystem server running at once, but it may migrate, as illustrated. Our proposed scheme, the sequential approach, will execute the subsystems in sequential order, occupying each core with one server.

IV. DISCUSSION

The implementation complexity of the four schemes differ in both the local (the scheduling of tasks inside subsystems), and the global (the scheduling of servers) level. Shin et al. [11] has the most complex server scheduler (global level), since the maximum number of cores that may be utilized, and the occupied cores must be checked when scheduling a server. Checconi et al. [10] is more simple since all servers

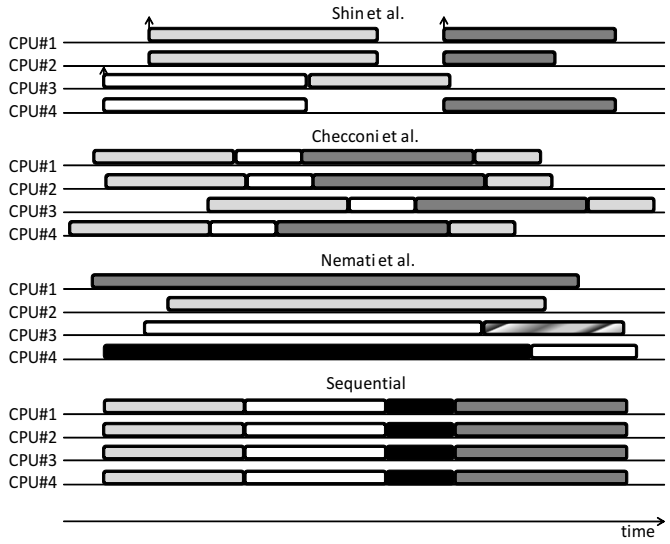

Fig. 2. Example server execution with all 4 schemes

are assigned to a core statically offline. In fact, one (H-CBS) global scheduler could be sufficient to handle all servers (which would make it similar to our sequential approach). Nemati et al. [19] is quite simple since there is maximum one server per subsystem, but the availability of cores need to always be checked since servers are not statically assigned to cores. Our proposed solution is simple since servers are statically assigned to cores.

Looking at the local scheduling, Shin et al. [11] and Checconi et al. [10] have similar local scheduling schemes. Both of them schedules global multi-core scheduling, on a subset of cores. Nemati et al. [19] uses simple partitioned scheduling (uni-core scheduling), while our approach schedules global multi-core scheduling on all cores. To summarize, [11] and [10] have slightly more complex scheduling than the sequential approach, while [19] is the most simple.

Looking at shared resources, due to that Nemati et al. [19] uses partitioned scheduling, shared resources within a subsystem becomes less complex with this approach, compared to the other three schemes.

Our aim is to implement all four approaches and compare their runtime scheduling complexity, their implementation complexity and their utilization of the cores with/without shared resources. Also, the current implementation could be extended with CBS, SS etc. and compared to other hierarchical schedulers, such as AQuoSA [15]. The main difference between these two approaches is that our scheduler does not require any patches, which may decrease the performance. It would be interesting to measure the performance loss due to the higher degree of usability.

REFERENCES

[1] 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.

[2] 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*, July 2010.

[3] 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.

[4] 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

[5] P. Gai, M. D. Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform," in *Proc. of the 9th Real-Time and Embedded Technology and Applications Symposium*, May.

[6] U. C. Devi, "Soft real-time scheduling on multiprocessors," Ph.D. dissertation, 2006.

[7] S. Baruah and N. Fisher, "The Partitioned Multiprocessor Scheduling of Sporadic Task Systems," in *Proc. of the 26th International Real-Time Systems Symposium*, December.

[8] T. P. Baker, "A comparison of global and partitioned edf schedulability tests for multiprocessors," In International Conf. on Real-Time and Network Systems, Tech. Rep., 2005.

[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*, July 2010.

[10] 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*, June 2009.

[11] I. Shin, A. Easwaran, and I. Lee, "Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors," in *Proc. of the 20th Euromicro Conference on Real-Time Systems*, July 2008.

[12] D. Faggioli and F. Checconi, "An EDF scheduling class for the Linux kernel," in *Proc. of the Real-Time Linux Workshop*, 2009.

[13] 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.

[14] V. Yodaiken, "The RTLinux Manifesto," in *Proc. of the 5th Linux Expo*, 1999.

[15] 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, 2009.

[16] IEEE Portable Application Standards Committee (PASC), "Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifations," vol. 7. IEEE, December 2008.

[17] D. Faggioli, A. Mancina, F. Checconi, and G. Lipari, "Design and Implementation of a POSIX compliant Sporadic Server for the Linux Kernel," in *Proc. of the 10th Real-Time Linux Workshop*, October 2008.

[18] 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*, November.

[19] F. Nemati, M. Behnam, and T. Nolte, "Multiprocessor Synchronization and Hierarchical Scheduling," in *Proc. of the 1st International Workshop on Real-time Systems on Multicore Platforms: Theory and Practice, in conjunction with ICPP'09*, September 2009.