A Loadable Real-Time Scheduler Suite for Multicore Platforms [*]

Shinpei Kato[†‡], Ragunathan (Raj) Rajkumar[†], and Yutaka Ishikawa[‡]
[†]Department of Electrical and Computer Engineering, Carnegie Mellon University
[‡]Department of Computer Science, The University of Tokyo

Abstract

Linux has received considerable attention in the embedded real-time systems domain recently, given that rich software libraries and device drivers are available. An issue of concern is, however, that the Linux kernel is not very capable of satisfying real-time systems requirements, particularly in the multicore context. In this paper, we present a loadable ***RE****al-time* ***SCH****eduler suite for Linux, called* ***RESCH****. RESCH supports easy installation of variant scheduling algorithms as plugins into Linux without any kernel patches, regardless of kernel configurations. We design and implement four scheduler plugins with different scheduling algorithms, corresponding to the state-of-the-art partitioned, semi-partitioned, and global scheduling techniques for multicore platforms. We also develop a* ***Sched****ulability* ***Bench****marking tool, called* ***SchedBench****, to evaluate the actual runtime schedulability of these scheduler plugins. Experiments demonstrate that the scheduler plugins boost schedulability for both arbitrary and harmonic periods, as compared to the native Linux scheduler. Particularly, the scheduler plugin based on semi-partitioned scheduling achieves better performance in most cases, but the one based on partitioned scheduling is also competitive in the harmonic cases. Meanwhile, the ones based on global scheduling are often inferior to them, and do not receive benefit from the harmonicity very much.*

1 Introduction

Today, real-time systems fulfill a significant role as a part of our societal infrastructure. Robots, cars, multimedia systems, and most other embedded systems are essentially built on real-time systems, in which the computation correctness depends not only on functional correctness but also on timing correctness.

In the early days, real-time systems were often closed systems. Thus, they used light-weight real-time operating systems (RTOSs) as their platforms. By contrast, modern real-time systems are likely to be open, given that systems are connected by networks and running on more powerful platforms. To run multiple applications in these kinds of open environments with graphics, I/O devices, virtual machines, etc., classical light-weight RTOSs are not sufficient from the viewpoints of functionality, tool set, and utility.

Because of the factors raised above, Linux has received considerable attention in the embedded real-time systems domain recently. A rich set of software libraries and device drivers are available for Linux. In addition, many vendors provide preview kits for various hardware platforms that can be downloaded for free. For instance, advanced humanoid robots [1, 14] have been already developed upon Linux. An issue of concern is, however, that the Linux kernel is not very capable of satisfying real-time systems requirements, as compared to classical RTOSs, though its mainline development has started actively supporting real-time tasks. Therefore, the Linux kernel needs to be patched and extended so that it can provide sufficient real-time performance when used in real-time systems.

Most traditional Linux-based RTOSs [8, 9, 10, 29, 31, 35, 36, 38, 39, 40] apply kernel patches for the purpose of installing real-time performance extensions. The requirement of kernel patches is, however, a complex step for both developers and users. Developers are obligated to constantly maintain their patches to keep up with the latest kernel releases, while users have to apply the patches and rebuild the kernel every time they adopt new patch releases. Besides, the kernel versions available for those Linux-based RTOSs tend to be limited, given that the Linux kernel may upgrade to the next version before developers port their kernel patches to the current version.

This paper explores how we can improve real-time performance in Linux *without any kernel patches*. To this end, we present a loadable **RE**al-time **SCHE**duler suite, called **RESCH**, which works together with the Linux kernel. We particularly focus on (i) runtime schedulability, (ii) fixed-priority preemptive scheduling policies, and (iii) multicore platforms. The primary objective of RESCH is to support easy installation of variant scheduling algorithms as plugins into Linux without any kernel patches, *regardless of kernel configurations*. RESCH is also aimed at evaluating the actual runtime schedulability of scheduling algorithms based on the state-of-the-art partitioned, semi-partitioned, and global scheduling techniques for multicore platforms. The following are the major components of RESCH:

- **R-Core**: The RESCH core provides a basic real-time scheduler and the interfaces for scheduler plugins.
- **R-Plugin(s)**: The RESCH plugin(s)implement variant scheduling algorithms.
- **R-Lib**: The RESCH library provides application programming interfaces (APIs) to user applications.
- **SchedBench**: A benchmarking tool measures the actual runtime schedulability of the scheduling algorithms supported by RESCH.

Unlike prior Linux-based RTOSs, RESCH does not need any kernel patches, and is therefore compatible with the mainline of the Linux kernel and some prior Linux-based RTOSs, since all native features from the Linux kernel remain. As a result, it brings benefits to both developers and users from the viewpoints of expandability and availability. Meanwhile, RESCH is not capable of improving timing latency and temporal granularity, since these kinds of basic real-time performance are dependent on the Linux kernel.

[*]This work is supported by the fund of Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists.

However, it may be possible to use RESCH together with the well-known RT-Preempt patch [26] that aims to improve basic real-time performance in Linux, given that RESCH does not modify the kernel source code.

To the best of our knowledge, this is the first piece of work that (i) achieves completely-loadable real-time schedulers without kernel patches, (ii) implements the state-of-the-art semi-partitioned scheduling technique, (iii) compares fixed-priority scheduling algorithms on real-world multicore platforms, and (iv) develops an open-source schedulability benchmarking tool. RESCH is open source, and can be downloaded from our website[1].

The rest of this paper is organized as follows. The next section defines our system model considered in this paper. Section 3 presents the framework and the basic functions of RESCH. Section 4 designs and implements four scheduler plugins for multicore platforms. Section 5 develops a schedulability benchmarking tool, and Section 6 evaluates the actual runtime schedulability of RESCH with different scheduler plugins, as compared to the native Linux scheduler. This paper is concluded in Section 7.

2 System Model

The system contains m CPU cores, and a set of n sporadic tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ_i is characterized by a tuple (C_i, D_i, T_i), where C_i is a worst-case computation time, D_i is a relative deadline, and T_i is a minimum inter-arrival time that is also referred to as a period. The utilization of τ_i is also denoted by $U_i = C_i/T_i$. RESCH particularly assumes such constrained-deadline systems that satisfy $C_i \leq D_i \leq T_i$ for any τ_i. When a task τ_i has $D_i > T_i$, RESCH internally transforms D_i to $D_i = T_i$.

Each task τ_i generates a sequence of jobs, each of which has a computation time less than or equal to C_i. A job of τ_i released at time t has a deadline at time $t + D_i$. Arrivals of successive jobs of τ_i are separated by at least T_i. Tasks are independent and preemptive: there are no shared resources and critical sections. Classical resource-sharing techniques like priority inheritance will be supported later.

3 RESCH Framework

In this section, we present a framework that provides basic functions for fixed-priority scheduling of recurrent real-time tasks in Linux, and enables new scheduling algorithms to be installed as plugins. The RESCH framework consists of a loadable kernel module and a user library. While the RESCH library provides API functions to user applications, the RESCH module manipulates the Linux scheduler by calling the exported kernel functions, such as `schedule()` and `sched_setscheduler()`. The RESCH module also has interfaces to plugins that implement new scheduling algorithms.

The conceptual framework is depicted in Figure 1. The RESCH core is built as a character device module and is accessed through a device file `/dev/resch`. When user applications call the RESCH APIs, the RESCH library notifies the RESCH core of the API type and its arguments, using an `ioctl()` system call. The RESCH core and the RESCH plugins, if any, execute corresponding procedures with the exported kernel functions, according to the implemented scheduling algorithm. In order to activate RESCH,

[1]http://www.ece.cmu.edu/~shinpei/resch/.

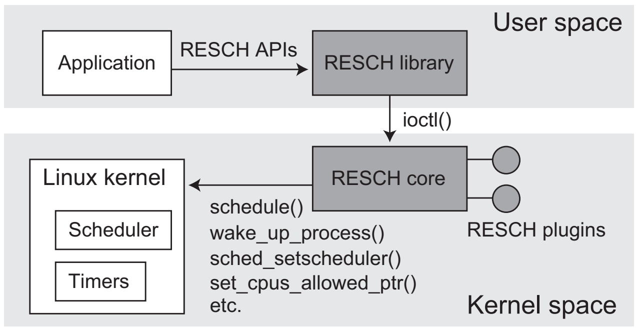


Figure 1. Conceptual framework of RESCH.

all users have to do is to install the RESCH core and the RESCH plugins, using the standard `insmod` command, and link the RESCH library to applications when compiled.

Note that the conceptual framework of RESCH is similar to that of KURT Linux [3] in that `ioctl()` is used to communicate with the kernel, though RESCH differs in that kernel patches are not required.

RESCH makes use of the Completely Fair Scheduler (CFS) implemented in the Linux kernel, to isolate real-time tasks and normal tasks. CFS provides scheduling classes `rt_sched_class` and `fair_sched_class` for real-time tasks and normal tasks respectively. Tasks in `rt_sched_class` are always scheduled ahead of ones in `fair_sched_class`. In other words, normal tasks are never executed when some real-time tasks are ready. RESCH adds its tasks to `rt_sched_class` so that they are not affected by normal tasks.

3.1 API Specification

The basic APIs supported by RESCH are listed in Table 1. Some APIs have arguments and others do not. Figure 2 shows a sample C program, using these APIs. The program enters the real-time mode, using the `rt_enter()` API. Next, a worst-case execution time, a period, a deadline, and a priority are set. Then, this sample starts recurrent real-time execution immediately with no acceptance test. The task submits `nr_jobs` numbers of jobs, each of which executes the user's code in `for loop`, and returns to the normal mode, using the `rt_exit()` API.

We believe that the API specification of RESCH is reasonable, given that existing Linux-based RTOSs, such as RTLinux [39], RTAI [8], Linux-RK [31], and KURT Linux [35], also use similar API specifications.

3.2 Management of Timing Properties

In order to realize real-time scheduling in Linux, we need to attach timing properties, such as a release time, a deadline, etc., to each task. However, the task descriptor of the Linux kernel does not contain members related to those timing properties. Though they may be possibly supported in future kernel versions, the kernel versions available for RESCH are strictly limited in this case.

For version-independent use, RESCH holds its own task descriptor regarding timing properties in the kernel module space. This task descriptor is shown in Figure 3. The `task` field contains a pointer to the Linux task descriptor associated with each `resch_task[k]`. Other members

Table 1. Basic RESCH APIs.

rt_enter()	Changes a caller to a real-time task. Returns failure if (i) the caller has no root permission or (ii)the # of tasks managed by RESCH exceeds NR_RESCH_TASKS.
rt_exit()	Changes a caller back to a normal task.
rt_run(timeout, test)	Starts recurrent real-time execution when an interval of @timeout elapses. The acceptance test is conduced if @test flag is set.
rt_wait_for_period()	Waits for the next period by sleeping with TASK_UNINTERRUPTIBLE status. Returns failure if a caller has missed its deadline.
rt_set_wcet(wcet)	Sets @wcet to the worst-case execution time of a caller by the timeval type.
rt_set_period(period)	Sets @period to the minimum inter-arrival time of a caller by the timeval type.
rt_set_deadline(deadline)	Sets @deadline to the relative deadline of a caller by the timeval type.
rt_set_priority(priority)	Sets @priority to the priority of a caller within the range of 1 to 99.

```
main(timeval C, timeval T, timeval D)
     int prio, int nr_jobs,
{
  rt_enter();
  rt_set_wcet(C);
  rt_set_period(T);
  rt_set_deadline(D);
  rt_set_priority(prio);
  rt_run(0,false);
  for (i = 0; i < nr_jobs; i++) {
      /* User's code. */
      ...
      rt_wait_for_period();
  }
  rt_exit();
}
```

Figure 2. Sample code using RESCH APIs.

are regarding timing properties. NR_RESCH_TASKS indicates the maximum number of tasks that can be managed by RESCH. The kernel module accesses the current task only through the `current` macro, which points to the Linux task descriptor of the current task. However, the Linux task descriptor does not know which `resch_task[k]` is attached, since it has no member that points to the RESCH task descriptor. So, we need a way to obtain the RESCH task descriptor from the Linux task descriptor to access its timing properties.

To associate the RESCH task descriptor with the Linux task descriptor, the RESCH core reassigns the process ID (PID) of each submitted task so that the new PID can imply the element `k` of `resch_task[k]`. We know that the Linux kernel assigns PIDs in the range of [1, `pid_max`] where `pid_max` is defined in the `/proc/sys/kernel/pid_max` file. Let us define `pid_offset` = `pid_max` + 1. Then, the PIDs are reassigned in the range of [`pid_offset`, `pid_offset` + `NR_RESCH_TASKS - 1`] so that we can obtain `resch_task[k]` from the current task, using the simple calculation `k` = `current->pid` - `pid_offset`. This procedure is done when the `rt_enter()` API is called by user applications, and the RESCH core finds and attaches such `resch_task[k]` that is not used by any prior tasks.

3.3 Basic Real-Time Scheduler

The RESCH core supports the fixed-priority preemptive scheduling of periodic tasks as a basic scheduler. This section is focused on the basic scheduler design and implementation, and scheduler plugins are presented in Section 4.

```
struct resch_task_struct {
  struct task_struct *task;
  unsigned long wcet;
  unsigned long period;
  unsigned long deadline;
  unsigned long exec_time;
  unsigned long release_time;
} resch_task[NR_RESCH_TASKS];
```

Figure 3. RESCH task descriptor.

The RESCH core mainly uses the following kernel functions exported from the Linux kernel to implement the basic real-time scheduler. Some functions have (slightly) different names, depending on kernel versions.

- `schedule()` switches the current execution context to the highest-priority task ready on the local CPU. Note that the Linux kernel makes context switches only in this function.
- `sched_setscheduler(task,policy,prio)` sets the scheduling policy and the priority of the task.
- `setup_timer(timer,func,arg)` associates the timer object with the given function and its argument.
- `mod_timer(timer,timeout)` activates (or reactivates) the timer object so that it invokes when the timeout expires.
- `set_cpus_allowed_ptr(task,cpumask)` specifies the CPUs upon which the task is allowed to execute, and is used to make migrations.

The Linux kernel offers two posix-compliant scheduling policies for real-time tasks, `SCHED_RR` and `SCHED_FIFO`; `SCHED_RR` executes tasks of the same priority level in round-robin fashion, while `SCHED_FIFO` does first-in-first-out. Either way, the scheduler dispatches the highest-priority task ready on the local CPU, and therefore they do not differ in theoretical schedulability. In consideration of fairness, we use `SCHED_RR` by default. Thus, the Linux kernel supports *fixed-priority scheduling*.

Given that most real-time systems contain periodic tasks, the RESCH core supports periodic execution in addition to fixed-priority scheduling. Since we aim at applying no kernel patches, we need to explicitly call the exported kernel functions raised above to affect the Linux scheduler.

In the design and implementation of the RESCH core, it is important to understand that context switches occur only (i) when jobs with higher priorities than the current

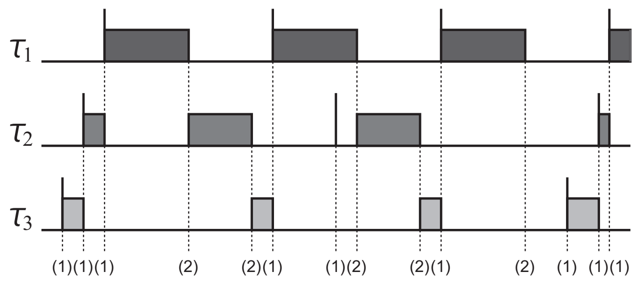


Figure 4. Example of fixed-priority scheduling of three periodic tasks.

job are released or (ii) when jobs complete, under fixed-priority preemptive scheduling disciplines. Figure 4 depicts an example of the fixed-priority scheduling of three periodic tasks τ_1, τ_2, and τ_3, where tasks with lower indices have higher priorities. It is easily seen that context switches occur only for job releases and job completions, marked by "(1)" and "(2)" respectively.

The above discussion suggests that the `schedule()` function be called when jobs are released or complete, given that the Linux kernel originally offers a fixed-priority scheduler. Figure 5 shows how and when the RESCH core invokes the `schedule()` function. It also shows how and when the plugin interfaces are called.

Let us first focus on the `schedule()` function calls. Every time a job completes, a user task calls the `rt_wait_for_period()` API (see Figure 2). Then, the RESCH core invokes a corresponding internal function, `rt_wait_for_period_internal()`, which calls another internal function `job_complete()`. This internal function also calls an internal function `sleep_in_period()`, which activates a timer and goes to sleep. The timer is associated with an internal function, `job_release()`, and it is set to invoke at the next release time. On invocation, it wakes up the task given by its argument, and a new job is released. The `sleep_in_period()` function is also called by an internal function `rt_run_internal()`, which is corresponding to the `rt_run()` API.

The `rt_run_internal()`, `job_release()`, and `job_complete()` internal functions contain the `task_run_plugin`, `job_release_plugin`, and `job_complete_plugin` interfaces respectively. These plugin interfaces are function pointers, which are to point to functions implemented in scheduler plugins when they are installed. The RESCH core does not offer any other functions beyond the fixed-priority preemptive scheduling of periodic tasks, but functional extensions can be supported by scheduler plugins. The details of scheduler plugins are described in Section 4.

Figure 6 illustrates a time-line flow from a job completion to a job release, including plugin interface calls. A user task can be switched to other tasks when it calls the `rt_wait_for_period()`, while it can be resumed at the next release time. The figure illustrates an example sequence in which the priority of the task is the highest in the ready tasks when it is released, and therefore it can preempt the preceding tasks at the release time. Scheduler plugins can hook the RESCH plugin execution parts. In this way, we realize the fixed-priority preemptive scheduling of periodic tasks, without applying any kernel patches.

```
job_release(resch_task_struct *p)
{
  p->deadline =
          p->release_time + p->deadline;
  job_release_plugin(p);
  wake_up_process(p->task);
}
sleep_in_period(resch_task_struct *p)
{
  setup_timer(timer, job_release, p);
  mod_timer(timer, p->release_time);
  p->task->state = TASK_UNINTERRUPTIBLE;
  schedule();
  del_timer(timer);
}
job_complete(resch_task_struct *p)
{
  p->release_time += p->period;
  job_complete_plugin(p);
  if (p->deadline < jiffies)
    sleep_in_period(p);
}
rt_run_internal(int timeout)
{
  k = current->pid - pid_offset;
  resch_task[k].release_time =
                    jiffies + timeout;
  task_run_plugin(&task_task[k]);
  sleep_in_period(&resch_task[k]);
}
rt_wait_for_period_internal()
{
  k = current->pid - pid_offset;
  job_complete(&resch_task[k]);
}
```

Figure 5. Internal functions of the RESCH core for job release and job completion.

3.4 Task Migration

Before moving on the design and implementation of the scheduler plugins, we present the task migration mechanism in RESCH, since they likely require task migration.

Traditionally, the `set_cpus_allowed_ptr()` function (this function is referred to as `set_cpus_allowed` in earlier versions) is used to perform task migration in the Linux kernel. The usage of this function to migrate a task to a certain CPU is illustrated in Figure 7. We use the `migrate_task()` function for migration, but there are two scenarios that we need to take into account. The simple scenario is that the program runs in the thread context. In this scenario, we can directly call the `__migrate_task()` function. However, in the other scenario wherein the the program is executed in the interrupt context, we should not directly call this function. This is because the `set_cpus_allowed_ptr()` function is to invoke the `schedule()` function, which is not allowed during interrupts, unless the kernel is built with the `CONFIG_PREEMPT` option. If the `CONFIG_PREEMPT` option is activated by default, the RESCH core just uses the `__migrate_task()` function for both scenarios.

Since we also aim at removing time and effort for rebuilding the kernel, we provide an alternative approach. To enable task migration in the interrupt context, the RESCH core creates a kernel thread called *migration thread*, whose objective is to service task migration, when it is installed.

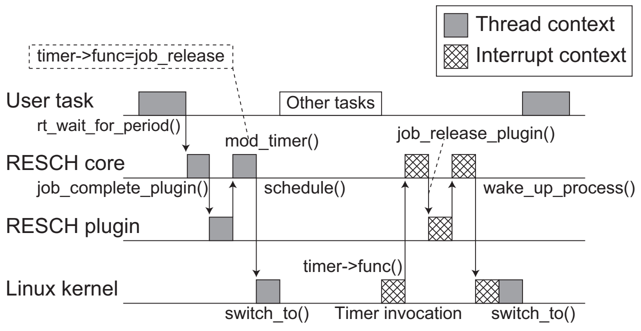


Figure 6. Control flow of RESCH.

Since a kernel thread runs in the thread context, it can use the `__migrate_task()` function. The migration thread is assigned the highest priority so that it can react immediately when requested, but otherwise sleeps so as not to interfere with user tasks. When task migration is needed during interrupts, the RESCH core wakes up the migration thread, and puts the task to be migrated into the migration list. The migration thread pulls the task from the list, and calls the `__migrate_task()` function.

Henceforth, "`migrate_task(task,cpu)`" represents a migration procedure to migrate the given task to the specified CPU, by (i) the `__migrate_task()` function if the RESCH core code is in the thread context, and (ii) by using the migration thread if it is in the interrupt context. Every migration incurs one context switch in the former case, while it incurs two context switches in the latter case due to the additional invocation of the migration thread. Timing analysis must take into account this overhead, when it is applied.

3.5 Ideas for Further Extensions

We claim that RESCH is extensible for Earliest Deadline First (EDF) [23]. Given that EDF also limits scheduling points to job releases and job completions, we can still use the RESCH core for context switching. Dynamic priority assignments are required when jobs are released, but they can be supported by scheduler plugins, using the `job_release_plugin()` interface. However, since the Linux scheduler limits the priority level, priority decisions may be problematic. Consider that some jobs have already been assigned priorities p and $p+1$ respectively. What happens if another job with its deadline between them is released? From now on, this job is called *problem job*.

We do not like to reassign all the priorities due to runtime overhead. Instead, our approach manipulates the priority array implemented as runqueues in the Linux kernel. The priority array is a two-dimensional queue: each element of the array is a list, and when a task with priority p is activated, it is inserted into the tail of the list of the pth element. We make efficient use of this characteristic. Let us look back the case in which the two jobs have already been assigned priorities p and $p + 1$, and the problem job with its deadline between them is released. We just assign a priority p to the released job, and set the `TASK_UNINTERRUPTIBLE` state to all the tasks that are executing jobs with later deadlines than the released job but reside in the list of the pth element, so that they will sleep and be deactivated from the list temporarily. When the problem job completes, we wake up those sleeping tasks so that they will be scheduled again.

```
__migrate_task(task, cpu)
{
  cpumask mask;
  cpus_clear(mask);
  cpu_set(cpu, mask);
  set_cpus_allowed_ptr(task, &mask);
}
```

Figure 7. Internal function for migration.

As described above, we can realize the EDF preemptive scheduling through RESCH, without any kernel patches. In fact, the EDF implementation is also considered in the Real-Time Linux community [41, 13].

In addition to EDF, we claim that other efficient technique are also applicable for RESCH. For instance, the resource reservation technique [31] can be implemented, using the `RLIMIT_RTTIME` signal function supported in the latest kernel versions, or just using timers. The aperiodic server techniques [21, 34] can be also implemented upon the resource reservation. The details of the further extensions will be considered in future work.

4 Scheduler Plugins for Multicores

In this section, we design and implement four scheduler plugins with different scheduling algorithms, corresponding to the state-of-the-art partitioned, semi-partitioned, and global scheduling techniques for multicore platforms. In this paper, we focus on fixed-priority scheduling algorithms, particularly Deadline Monotonic [22].

Henceforth, the plugin functions called through the `task_run_plugin()`, `job_release_plugin()`, and `job_complete_plugin()` interfaces are represented as `task_run_X()`, `job_release_X()`, and `job_complete_X()` respectively, where 'X' denotes the scheduler plugin name.

4.1 Partitioned Scheduling

The Linux kernel adopts partitioned scheduling for multicores: each CPU has its own scheduler and runqueue. Tasks in each runqueue are scheduled by the local scheduler, and do not migrate onto another CPU, unless some CPUs are overloaded.

In partitioned fixed-priority scheduling, CPU allocation methods affect schedulability [11, 12, 28]. The standard Linux kernel allocates CPUs to tasks so that the CPU load is balanced. This allocation is similar to worst-fit allocation. Meanwhile, it has been reported that better schedulability is obtained by first-fit allocation, particularly with task-sorting based on utilization factors [11, 12, 28].

We develop a scheduler plugin called FP-FF, which adopts first-fit allocation for partitioning tasks among CPUs. In this paper, we do not consider task sorting, given that it incurs a lot of re-queuing procedures across CPUs at runtime. If necessary, it can be supported by another plugin.

The plugin implementation is straightforward. The plugin uses only the `task_run_plugin()` interface to carry out partitioning before execution. Every time the `task_run_FP-FF()` function is called, FP-FF seeks such a CPU that can accommodate the given task, based on a response time analysis [4]. Then, the task is migrated by the `migrate_task()` function to the CPU that is verified first. If a task cannot be assigned to any CPUs, it is just assigned the lowest priority. Runtime scheduling is dependent on the RESCH core.

4.2 Semi-Partitioned Scheduling

Recent work [17, 20] have made available a new scheduling class called *semi-partitioned scheduling*, for fixed-priority algorithms. Semi-partitioned scheduling is an extension of partitioned scheduling in that most tasks are assigned to particular CPUs, but a few tasks are permitted to migrate across more than one CPU when they cannot be successfully partitioned.

We develop a scheduler plugin called FP-PM, which leverages the semi-partitioned scheduling algorithm proposed in [17]. The algorithm combines first-fit allocation and semi-partitioned scheduling. When it finds a task that cannot be assigned to any CPUs by first-fit allocation, it permits the task to migrate across more than one CPU at runtime. This migratory task is statically assigned the highest priority. The processing time that the migratory task is allowed to consume on each CPU is computed based on a response time analysis. At runtime, when the task consumes the assigned processing time on a CPU, it is migrated to another CPU upon which it is allowed to execute. For more details, please see [17].

The implementation of FP-PM is more complicated than FP-FF. It uses the `task_run_plugin()` and `job_release_plugin()` interfaces. As FP-PM, CPU allocation is done in the `task_run_FP-PM()` function. The migration decision is also made in this function. If a given task is successfully assigned to a particular CPU, the task is scheduled in the same way as FP-FF. However, if the task needs migration to be schedulable, FP-PM conducts additional procedures in the `job_release_FP-PM()` function. If a task is verified to be unschedulable even by using migrations, it is just assigned the lowest priority.

When a job of a migratory task is released, FP-PM migrates the task to the CPU that has the lowest index in the assigned CPUs, using the `migrate_task()` function presented in Section 3.4. FP-PM also runs a timer that triggers when the assigned processing time is consumed on this CPU, to migrate the task to the next CPU. When the migration is completed, it runs a timer again for the next migration. Timer invocations are continued till the task is migrated to the CPU that has the largest index in the assigned CPUs. Since the timer invocations run in interrupt contexts, the `migrate_task()` function uses the migration thread, as mentioned in Section 3.4. Timing information to run timers is stored within the plugin module space.

In fact, for systems that are sensitive to job release overhead, the task migration to the first CPU at the release time can be alternatively done when jobs complete, using the `job_complete_plugin` interface, namely when new jobs are released, they are already running on the first CPU.

4.3 Global Scheduling

Finally, we consider the global scheduling approach, wherein tasks are conceptually stored in a global queue that is shared among all CPUs, and at any time the *m* highest-priority tasks are scheduled.

We develop two scheduler plugins called G-FP and FP-US. G-FP simply dispatches tasks in global scheduling fashion, according to the given priorities. On the other hand, FP-US classifies tasks into heavy tasks and light tasks, based on utilization factors. If the CPU utilization of a task is greater than or equal to $m/(3m-2)$, it is a heavy task. Otherwise, it is a light task. Then, all heavy tasks are statically assigned the highest priorities, while light tasks have the original priorities. This idea has been proposed in [2].

The plugins are implemented in such a way that still uses local schedulers like partitioned scheduling but imitates global scheduling, using the `job_release_plugin()` and `job_complete_plugin()` interfaces. We first focus on the implementation of G-FP.

When a job of a real-time task is released, G-FP seeks a CPU that is currently not executing a real-time task, using the `job_release_G-FP()` function. If it exists, this task is migrated to this CPU. If it does not, G-FP next checks if there are CPUs that are currently executing real-time tasks with lower priorities than this task. If there are, this task is migrated to the CPU that is executing the lowest-priority task. Otherwise, G-FP does nothing for this task, but instead this task may be later migrated to some CPU in the `job_complete_G-FP()` function when some jobs complete, as described below.

When a job of a real-time task completes, G-FP migrates a task that has the highest priority other than the current tasks, if it exists, to the CPU upon which the completed job has been running, using the `job_complete_G-FP()` function.

In the case of FP-US, we additionally make available the `task_run_FP-US()` function to classify heavy tasks and light tasks. The highest priority assignment to the heavy tasks is also processed in this function.

Since a local scheduler always dispatches the highest-priority task in its own runqueue, if every runqueue contains one of the *m* highest-priority tasks, it is subject to global scheduling. We only need a global task list that holds ready tasks in the order of priorities. However, the global task list must be protected by a lock, which introduces overhead.

5 Schedulability Benchmark

To the best of our knowledge, traditional published real-time benchmarks [27, 33, 37] are designed for specific applications and worst-case execution analysis, but not available to test schedulability on multicore platforms. Hence, we develop our original schedulability benchmarking tool for Linux, called **SchedBench**. SchedBench is designed and implemented to work with RESCH.

Schedulability is affected by many factors, such as execution times, periods, deadlines, CPU utilization, the number of CPUs, particularly on multicore platforms [6]. Thus, SchedBench generates a variety of tasksets, each of which contains tasks with many combinations of the factors, to tightly test the runtime schedulability of the scheduling algorithms installed through RESCH. Those timing values are produced according to user parameters, based on the idea presented in [6]. The available parameters of SchedBench are listed in Table 2.

SchedBench generates *working sets* $\{ws(k)\}$. Each $ws(k)$ ($\text{start} \leq k \leq \text{end}$) is a bundle of quantity tasksets, each of which contains tasks with different timing properties such that their total workload gets around ($k \times \text{cpus}$)%. Specifically, SchedBench first creates a *taskset file* that describes the names, execution times, periods, and relative deadlines of the tasks for each $ws(k)$. The timing properties are determined based on the parameters listed in Table 2.

Table 2. SchedBench parameters.

cpus	the number of CPUs to use.
(Umin, Umax)	the range of the utilization of every individual task (0 ≤ Umin ≥ Umax ≤ 1.0).
(Pmin, Pmax)	the range of the period of every individual task (by millisecond).
dist	the distribution of the utilization of every individual task (by uniform, bimodal, or exponential).
sep	the separator to district light tasks and heavy tasks in a bimodal distribution.
prob	the probability of being heavy tasks in a bimodal distribution.
mean	the mean value in an exponential distribution.
deadline	the type of deadlines (by implicit, constrained, or arbitrary).
period	the type of periods (by arbitrary or harmonic).
time	the length of benchmarking time (by millisecond).
(start, end)	the range of the system utilization that is tested by benchmarking (0% ≤ start ≤ end ≤ 100%).
step	the distance of every successive sampling system utilization to be tested by benchmarking.
quantity	the number of the tasksets to be tested by benchmarking per system load.

Once all the taskset files are created, SchedBench goes to the benchmarking stage. The taskset files are saved so that we can use the same tasksets for every benchmarking target. Before benchmarking, SchedBench measures how many empty loops consume one millisecond. Let ms be a loop count to consume one millisecond. SchedBench calls the fork system call to spawn a process for each task τ_i written in the taskset files, which executes $C_i \times ms$ empty loops. The benchmarking of $ws(k)$ is finished when time elapses or some tasks miss deadlines.

In fact, the calculation of execution times is optimistic, but it is still reasonable in that we aim at measuring the average-case performance. Besides, since empty loops do not access data cache very often, the execution time is likely stable. If users desire more precise calculation, worst-case execution time analysis tools are required.

Tasksets are said to be *successfully scheduled*, if and only if no tasks miss deadlines during benchmarking. The schedulability for each $ws(k)$ is assessed by the *success ratio*, which is defined as follows.

$$\frac{\textit{the number of successfully-scheduled tasksets}}{\textit{the number of tasksets (i.e. } \text{quantity})}$$

6 Experimental Evaluation

We now evaluate the developed scheduler plugins by SchedBench. Specifically, we test the schedulability of FP-FF, FP-PM, G-FP, FP-US, and FP (the basic real-time scheduler with no plugins). FP actually reflects the performance of the native Linux scheduler. Remember that the priority assignments are based on Deadline Monotonic.

6.1 Experimental Setup

A series of experiments is conducted on the Linux kernel Version 2.6.29.4. The testing machine has two 3.16GHz Intel Xeon CPUs (X5460), each of which contains four cores (hence the machine is considered as having eight CPUs), and 4GB of main memory.

The SchedBench parameters tested in the experiments are listed in Table 3. Due to space constraints, we test only basic parameters that have been used for the previous simulation-based schedulability evaluation [17]. We test 1,000 tasksets for each sampling system load, gradually increased by 10% starting from 50% and ending with 100%. Each taskset is scheduled for about 10 minutes.

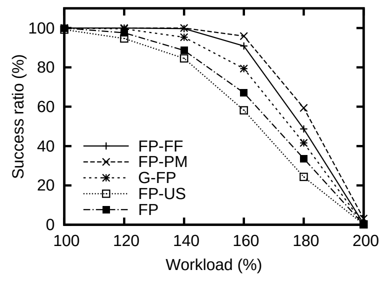
Table 3. Parameters setup.

cpus	2, 4, 8
(Umin, Umax)	(0.1, 1.0), (0.1, 0.5), (0.5, 1.0)
(Pmin, Pmax)	(1, 1000)
dist	uniform
deadline	implicit
period	arbitrary
time	600,000 (10 minutes)
(start, end)	(50%, 100%)
step	10%
quantity	1,000

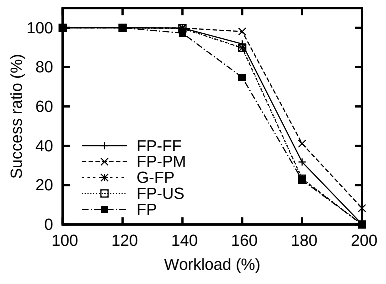
The maximum number of the tasks managed by RESCH is set to 64. We also measured the worst-case context switching overhead. Since most cases showed the overhead less than than 80μs, we define the overhead as 100μs for safety. This overhead is taken into account in response time analysis when FP-FF and FP-PM assign tasks to CPUs based on their schedulability tests. See [17] how to reflect the overhead in task assignments. Note that the definition of the overhead cost does not affect G-FP and FP-US, since they always globally dispatch the m highest tasks, and no schedulability tests are conducted.

6.2 Results: Arbitrary-Period Cases

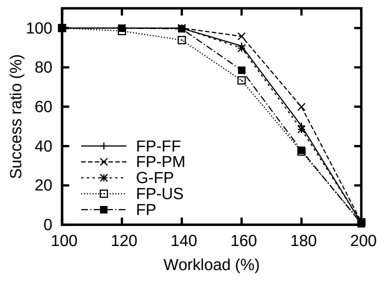
Figure 8 shows the benchmarking results for arbitrary-period cases, wherein periods of tasks have no relations. The results show that FP-PM achieves better performance than others in most cases. Since semi-partitioned scheduling is a superset of partitioned scheduling, FP-PM should outperform FP-FF and FP. FP-FF is superior to FP in all cases, which demonstrates that the CPU allocation by a first-fit heuristic improves schedulability over the one implemented in the Linux kernel. G-FP is usually better than FP while worse than FP-FF. This observation leads to that partitioned scheduling may be inferior to global scheduling and vice versa, depending on CPU allocation methods. Meanwhile, FP-US remarkably shows the worst performance in the tested plugins. This is reasoned as follows. FP-US assigns the highest priority to heavy tasks, but clearly this may incur priority inversions. Consider such a heavy task τ_i that has a long relative deadline (and period). Since it is heavy, the execution time is also likely long. As a result, this heavy task can block light tasks with much shorter deadlines than its execution time, which results in



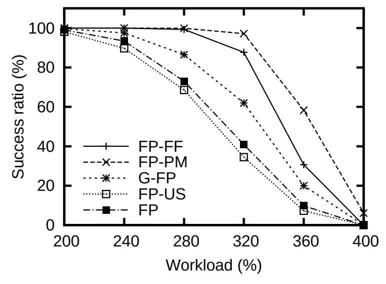
(a) (Umin,Umax)=(0.1,1.0), cpus=2



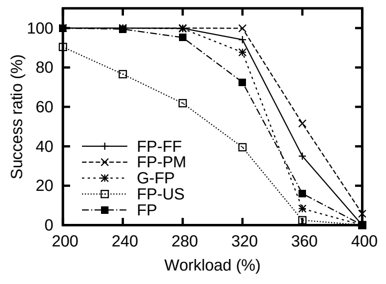
(b) (Umin,Umax)=(0.1,0.5), cpus=2



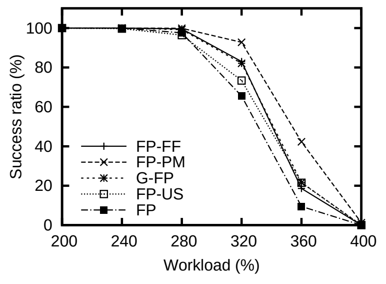
(c) (Umin,Umax)=(0.5,1.0), cpus=2



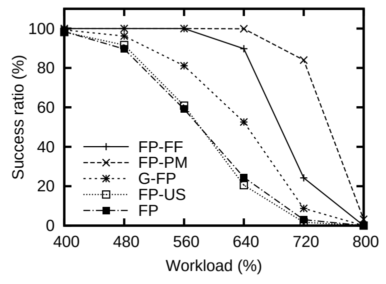
(d) (Umin,Umax)=(0.1,1.0), cpus=4



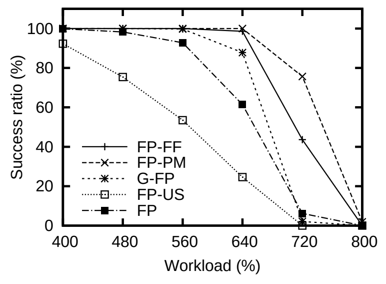
(e) (Umin,Umax)=(0.1,0.5), cpus=4



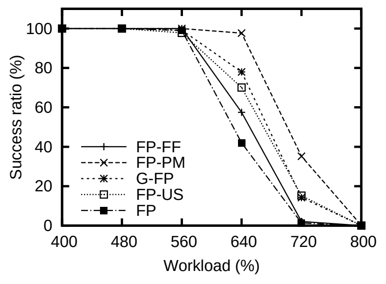
(f) (Umin,Umax)=(0.5,1.0), cpus=4



(g) (Umin,Umax)=(0.1,1.0), cpus=8



(h) (Umin,Umax)=(0.1,0.5), cpus=8



(i) (Umin,Umax)=(0.5,1.0), cpus=8

Figure 8. Benchmarking results for arbitrary periods.

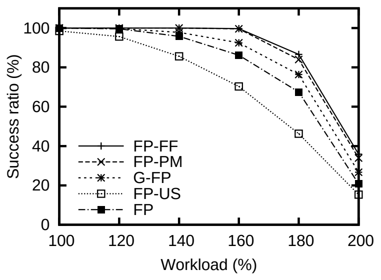
deadline misses. Therefore, FP-US can be worse than G-FP in the average case, though its worst-case schedulability is higher than G-FP [2, 7]. A hybrid of FP-US and G-FP is expected to perform better, as reported in [7].

The performance of the scheduler plugins is dependent on the range (Umin, Umax) of the utilization of every individual task. When tasksets contain both light tasks and heavy tasks, G-FP suffers from Dhall's effect [12], and tends to perform poorly as compared to other cases. FP-US is designed to avoid the Dhall's effect, but also shows poor performance, due to the reason stated above. On the other hand, the schedulability of FP-FF and FP can decline when tasks are likely heavy, as in the case of Figure 8 (i). We given an extreme example to reason this performance degradation. Consider $m+1$ tasks with utilization $(50+\alpha)$%. It is clear that one of the $m + 1$ tasks is not successfully assigned to any CPUs. In that case, it is inevitable for FP-FF and FP to cause deadline misses. FP-PM overcomes this scenario by using migrations.

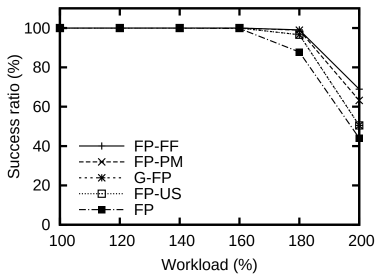
The number of CPUs also affects schedulability. In most cases, the performance of the scheduler plugins other than FP-PM declines as the number of CPUs increases. This result is natural in that the theoretical schedulable bound for partitioned scheduling and global scheduling is a function of the number of CPUs, and an increase in the CPU count results in a decrease in the bound [2, 6, 7, 24, 25]. Thus, the runtime performance reflects the theory. On the other hand, FP-PM effectively uses CPUs by task migration. In fact, the more CPUs are given, the more FP-PM obtains a chance to meet deadlines by task migration.

6.3 Results: Harmonic-Period Cases

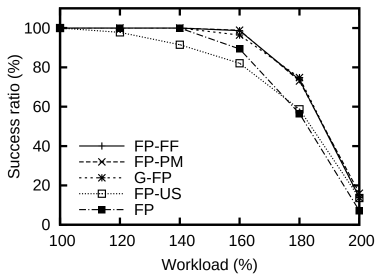
Figure 9 shows the benchmarking results for harmonic-period cases, wherein periods are integer multiples of each other. The results clearly show that FP-FF gains better schedulability, as compared to the arbitrary-period cases. Surprisingly, FP-FF outperforms FP-PM in some cases. This is reasoned as follows. Since the harmonicity improves the schedulability of fixed-priority algorithms on a single CPU [19], it obviously assists FP-FF when partitioning tasks among CPUs. This also means that the room for schedulability improvement by using task migration is decreased. Because FP-PM is aggressive in such a way that fully utilizes the CPU resource up to its bound when assigning migratory tasks, it is likely to cause timing violations if uncertainties occur during task migrations and timer invocations. So if only a little schedulability improvement is obtained by using task migration, it may be better just to give the lowest priority to the tasks that cannot be assigned



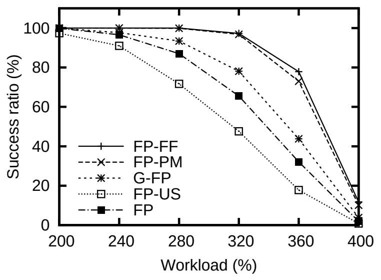
(a) (Umin,Umax)=(0.1,1.0), cpus=2



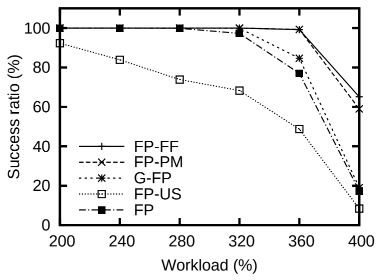
(b) (Umin,Umax)=(0.1,0.5), cpus=2



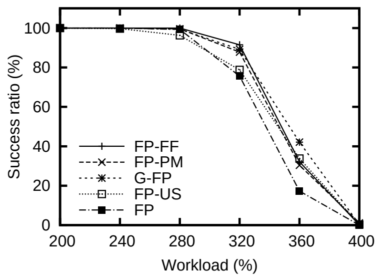
(c) (Umin,Umax)=(0.5,1.0), cpus=2



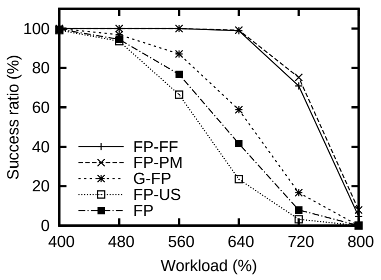
(d) (Umin,Umax)=(0.1,1.0), cpus=4



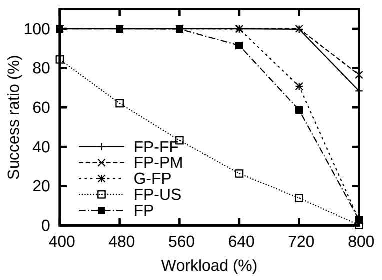
(e) (Umin,Umax)=(0.1,0.5), cpus=4



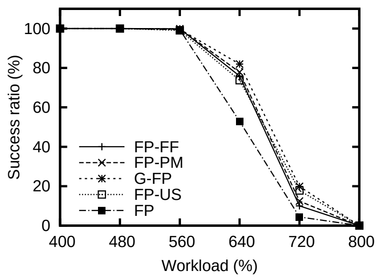
(f) (Umin,Umax)=(0.5,1.0), cpus=4



(g) (Umin,Umax)=(0.1,1.0), cpus=8



(h) (Umin,Umax)=(0.1,0.5), cpus=8



(i) (Umin,Umax)=(0.5,1.0), cpus=8

Figure 9. Benchmarking results for harmonic periods.

to any particular CPUs. On the other hand, little remarkable schedulability improvements can be seen in G-FP and FP-US. This observation leads to that global scheduling does not receive benefit from the harmonicity as much as partitioned scheduling.

7 Conclusion

In this paper, we presented RESCH, which is a loadable real-time scheduler suite that can be installed into Linux without kernel patches, regardless of kernel configurations. We designed and implemented four scheduler plugins with different scheduling algorithms based on the state-of-the-art partitioned, semi-partitioned, and global scheduling techniques for multicores. We also stated how it can be extended for EDF and other efficient techniques. In addition, we developed a schedulability benchmarking tool to assess the actual runtime performance of the install scheduling algorithms on real-world multicore machines.

Experiments were conducted to evaluate our scheduler plugins, as compared to the native Linux scheduler, using our schedulability benchmarking tool. According to the results, the semi-partitioned scheduling technique was very effective in improving multicore schedulability in most cases. The partitioned-scheduling technique was also competitive, particularly when tasks are harmonic. Meanwhile, the global scheduling was mostly inferior to them, and did not receive benefit from the harmonicity very much.

To the best of our knowledge, this is the first piece of work that succeeded in improving multicore schedulability in Linux, without applying kernel patches. Given that the requirement of kernel patches is a complex undertaking for both developers and users, we believe that RESCH and the RESCH approach are useful contributions.

In future work, we plan to verify the compatibility of RESCH with the RT-Preempt patch, which improves timing latency in the Linux kernel. We also plan to use high-resolution timer functions supported by the recent kernel versions, to make available more fine-grained processing. The RESCH core, developed in this paper, limits all timing properties to the jiffies (scheduler tick), but some real-time tasks may require finer granularity in practice. Scheduling algorithms may also need those high-resolution timers to be implemented or to be improved. The implementation of EDF and its variants [5, 16, 18, 15] will be also considered, as well as resource sharing [32], resource reservation [31], aperiodic servers [21, 34], and power management [30]. Furthermore, we will extend the benchmarking tool so that it can support practical applications, like Media Player that frequently accesses data cache.

References

[1] K. Akachi, K. Kaneko, N. Kanehira, S. Ota, G. Miyanori, M. Hirata, S. Kajita, and F. Kanehiro. Development of Humanoid Robot HRP-3P. In *Proc. of the IEEE-RAS Int'l Conf. on Humanoid Robots*, pages 50–55, 2005.

[2] B. Andersson, S. Baruah, and J. Jonsson. Static-priority Scheduling on Multiprocessors. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 193–202, 2001.

[3] A.K. Atlas and A. Bestavros. Design and Implementation of Statistical Rate Monotonic Scheduling in KURT Linux. In *Proc. of the Real-Time Systems Symp.*, pages 272–276, 1999.

[4] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):285–292, 1993.

[5] T.P. Baker. An Analysis of EDF Schedulability on a Multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16:760–768, 2005.

[6] T.P. Baker. Comparison of Empirical Success Rates of Global vs. Partitioned Fixed-Priority and EDF Scheduling for Hard Real Time. Technical report, Department of Computer Science, Florida State University, 2005.

[7] T.P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. *Real-Time Systems*, 32:49–71, 2006.

[8] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, 29:10, 2000.

[9] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 111–123, 2006.

[10] S. Childs and D. Ingram. The Linux-SRT Integrated Multimedia Operating Systems: Bringing QoS to the Desktop. In *Proc. of the IEEE Real-Time Technology and Applications Symp.*, pages 135–140, 2001.

[11] S. Davari and S.K. Dhall. An On Line Algorithm for Real-Time Allocation. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 194–200, 1986.

[12] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26:127–140, 1978.

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

[14] K. Kaneko, F. Kanehiro, H. Hirukawa, T. Kawasaki, M. Hirata, K. Akachi, and T. Isozumi. Humanoid Robot HRP-2. In *Proc. of the IEEE Int'l Conf. on Robotics and Automation*, pages 1083–1090, 2004.

[15] S. Kato and Y. Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *Proc. of the IEEE Real-Time Systems Symp.*, 2009.

[16] S. Kato and N. Yamasaki. Global EDF-based Scheduling with Efficient Priority Promotion on Multiprocessors. In *Proc. of the IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 197–206, 2008.

[17] S. Kato and N. Yamasaki. Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symp.*, pages 23–32, 2009.

[18] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-Partitioned Scheduling of Sporadic Task Systems on Multiprocessors. In *Proc. of the Euromicro Conf. on Real-Time Systems*, pages 249–258, 2009.

[19] T. Kuo and A. Mok. Load Adjustment in Adaptive Real-Time Systems. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 160–171, 1991.

[20] K. Lakshmanan, R. Rajkumar, and J.P. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In *Proc. of the Euromicro Conf. on Real-Time Systems*, pages 239–248, 2009.

[21] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 261–270, 1987.

[22] J. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation, Elsevier Science*, 22:237–250, 1982.

[23] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[24] J. Lopez, J. Diaz, and D. Garcia. Minimum and Maximum Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):642–653, 2004.

[25] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Utlization Bounds for Multiprocessor Rate-Monotonic Scheduling. *Real-Time Systems*, 24:5–28, 2003.

[26] Ingo Molnar. RT-Preemt Patch. http://www.kernel.org/pub/linux/kernel/projects/rt/.

[27] F. Nemer, H. Cass, P. Sainrat, J.P. Bahsoun, and M.D. Michiel. Papabench: A Free Real-Time Benchmark. In *Proc. of the 6th Int'l Workshop on Worst-Case Execution Time Analysis*, 2006.

[28] Y. Oh and S. Son. Allocating Fixed-Priority Periodic Tasks on Multiprocessor Systems. *Real-Time Systems*, 9:207–239, 1995.

[29] S. Oikawa and R. Rajkumar. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proc. of the IEEE Real-Time Technology and Aplications Symp.*, 1999.

[30] P. Pillai and K. Shin. Real-Time Dynamic Voltage Scaling on a Low-Power Microprocessors. In *Proc. of the ACM Symp. on Operating Systems Principles*, pages 89–102, 2001.

[31] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 298–307, 1997.

[32] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[33] B. Shirazi, L Welch, B Ravindran, and C. Cavanaugh. Dynbench: A Benchmark Suite for Dynamic Real-Time Systems. *Journal of Parallel and Distributed Computing Practices*, 3:89–107, 1999.

[34] M. Spuri and G.C. Buttazo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*, 10:179–210, 1996.

[35] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A Firm Real-Time System Implementation Using Commercial Off-the-Shelf Hardware and Free Software. In *Proc. of the IEEE Real-Time Technology and Applications Symp.*, pages 112–119, 1998.

[36] Y. Wang and K. Lin. Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 246–255, 1999.

[37] N.H. Weidernmann and N.I. Kamenoff. Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems. *Real-Time Systems*, 4(4):353–383, 1992.

[38] T. Yang, T. Liu, E.D. Berger, S.F. Kaplan, and J.E-B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proc. of the USENIX Symp. on Operating Systems Design and Implementation*, pages 73–86, 2008.

[39] V. Yodaiken. The rtlinux manifesto. In *Proc. of the 5th Linux Expo*, 1999.

[40] W. Yuan and K. Nahrstedt. Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems. In *Proc. of the ACM Symp. on Operating Systems Principles*, pages 149–163, 2003.

[41] P. Zijlstra. Deadline scheduling on Linux and why it hasn't happened yet. In *Community Developpers Track in the Real-Time Linux Workshop*, 2009.