ExSched: An External CPU Scheduler Framework for Real-Time Systems

Mikael Åsberg and Thomas Nolte
Mälardalen Real-Time Research Center
Mälardalen University (Sweden)

Shinpei Kato
Graduate School of Information Science
Nagoya University (Japan)

Ragunathan Rajkumar
Department of Electrical and Computer Engineering
Carnegie Mellon University (USA)

***Abstract*—Scheduling theory and algorithms have been well studied in the real-time systems literature. Many useful approaches and solutions have appeared in different problem domains. While their theoretical effectiveness has been extensively discussed, the community is now facing implementation challenges that show the impact of the algorithms in practice.**

In this paper, we propose a scheduler framework, called ExSched, which enables different schedulers to be developed for different operating system (OS) platforms without any modifications to the OS itself, using a unified interface. The framework will easily keep up with changes in the kernel since it is only dependent on a few kernel primitives. The usefulness of this framework is that scheduling policies can be implemented as external plug-ins. They can simply use the ExSched interface instead of platform-dependent functions, since platform details are abstracted by ExSched. The advantage for industry is that they would more easily keep up with new kernel versions since ExSched does not require patches. The advantage for academia is that we could focus on the development of schedulers instead of tedious and time-consuming installations of patched kernels.

Our prototype implementation of ExSched supports Linux and VxWorks and it comes with example schedulers which include hierarchical and multi-core schedulers in addition to traditional fixed-priority scheduling (FPS) and earliest deadline first (EDF) algorithms.

I. Introduction

The real-time systems community has addressed various scheduling problems. Examples include hierarchical scheduling, which composes multiple subtask systems into a single task system with real-time guarantees. Another notable study from the community is multi-core scheduling that extends traditional fixed-priority scheduling (FPS) and earliest deadline first (EDF) algorithms [1], [2] in a way that avoids well-known global and partitioned scheduling problems [3]. All these techniques are important in order to enhance real-time systems in performance and functionality.

While theory is becoming more and more mature, systems implementation remains to be a core challenge for the real-time systems community. We are aware of several studies of CPU-scheduler implementations from the previous work, particularly for hierarchical scheduling [4], [5], [6] and multi-core scheduling [7], [8], [9], [10]. There are also different types of implementation work [11], [12], [13], [14], [15], [16], [17], mostly targeted for Linux. However, most of this work is specific to certain platforms. Their prototype implementations are provided in one software platform, e.g., Linux, and it is not easily translated to other platforms, e.g., VxWorks. This prevents many interesting solutions (which are developed by the community) from being used in a wide range of systems. This is in particular a critical problem for real-time systems, as they have many different commodity operating system (OS) platforms, such as Linux, VxWorks, FreeRTOS, OSE, μC/OS-II, QNX, TRON, RTLinux etc. Even within each OS platform, existing solutions are often limited to some specific version of the underlying OS. The reason for this is because the solutions require patches (modifications) to parts of the original OS source code. These modifications are not necessarily consistent across different kernel versions. In particular, such version problems are significant for Linux-based solutions, since Linux continuously adds new functionality as the kernel version gets upgraded:

"*Of course, you could also dive in and modify Linux to convert it into a real-time operating system, since its source is openly available. But if you do this, you will be faced with the severe disadvantage of having a real-time Linux that can't keep pace, either features-wise or drivers-wise, with mainstream Linux. In short, your customized Linux won't benefit from the continual Linux evolution that results from the pooled efforts of thousands of developers worldwide*." [18]

Even minor version upgrades can have significant impact on the functionality of Linux, e.g., 2.6.23 for Completely Fair Scheduler, 2.6.25 for Control Groups, and 2.6.33 for GPU support.

Academia and industry can benefit from using non-intrusive solutions. Easier installation of frameworks and schedulers (which usually are patched kernels) on various software platforms (and different platform versions) could lead to more reusability of already implemented solutions in academia. The advantage for industry is that it would make it easier to update to newer kernel versions since loadable kernel-modules require much less (or no) kernel modifications compared to patches.

The contribution of this paper is a new scheduler framework that enables different scheduling techniques to be easily im-

plemented on different OS platforms. Specifically, we propose a scheduler framework, called **ExSched**, which provides a unified scheduler interface that can be used to implement different schedulers as external plug-ins for different OS platforms, without modifying to the underlying OS. One scheduler plug-in developed for some OS platform can directly be used on other platforms. Hence, with this framework we strongly argue for **(i)** portability across OS platforms/versions, and **(ii)** availability for scheduling techniques. Up until the day that OSs like Linux become so flexible in their structure that kernel source-code modifications become unnecessary, that is when ExScheds non-intrusiveness becomes pointless.

The rest of this paper is organized as follows. Section II presents related work in the area of real-time scheduler implementations in Linux. Section III provides our system model and basic assumptions. Section IV presents the design and implementation of our ExSched framework. Section V provides the development of plug-in examples with six scheduling algorithms. Section VI demonstrates the performance and overhead of the developed plug-ins on Linux and VxWorks. The paper is concluded in Section VII.

II. Related work

"Hijack" [19] is a real-time module for Linux which does not require any modifications to the underlying kernel; hence, this approach is similar to ours. Hijack uses kernel modules to intercept kernel services and relays them to user space tasks. The difference from our work is that we relay scheduling services to kernel modules. Also, our framework is not dependent on the hardware architecture, whereas Hijack relies on the assumption that the underlying hardware is x86. Another modification-free solution called Vsched [20] is capable of scheduling type-2 virtual machines in a periodic manner. This is similar to one of our ExSched plug-in schedulers [21]. In addition, we offer the possibility of plug-in scheduler development. LITMUS$^{\mathrm{RT}}$ [10] is a patch-based scheduler test-bed in Linux for multi-core schedulers. It is similar to ExSched since it also supports the development of schedulers. However, it differs in that ExSched does not require patches and we support the development of arbitrary schedulers (not just multi-core schedulers) on two different OS platforms (Linux and VxWorks). SCHED_DEADLINE [5] is another patch-based scheduler that implements EDF scheduling of servers. One of ExScheds plug-in schedulers support the same scheduling scheme. AQuoSA (Adaptive Quality of Service Architecture) [14] is a (patched) feedback-based resource reservation scheduler for Linux. The authors in [16] present a modular scheduling framework (similar to ours since it does not require kernel modifications) in the Red Linux real-time kernel. The main difference is that we target the vanilla Linux kernel (among other OSs). RT-Linux [17] is a hypervisor solution based on Linux. The fundamental idea is to let Linux execute as a process. RT-Linux targets pure hard-real time systems; however, it requires modifications to the Linux kernel. RTAI [11] is similar to RT-Linux. It uses a hypervisor (Adeos [22]) to get hard-real time capabilities out of Linux, at the cost of modifying the Linux kernel. Portable RK (Resource Kernel) [13] is a patch-based solution that enhances the real-time capabilities of Linux. The techniques used in [12], [15], [23] are also patched based. [6], [24] present two-level hierarchical scheduling without kernel modifications. [4] also implement hierarchical scheduling but it requires kernel modifications. Table I summarises the related work. As can be observed, most solutions are patch based and only ExSched is OS independent.

Solution	Type	Patch	OS independent
ExSched	**Framework**	**No**	**Yes**
Hijack [19]	Framework	No	No
Vsched [20]	Scheduler	No	No
LITMUS$^{\mathrm{RT}}$ [10]	Framework	Yes	No
SCHED_DEADLINE [5]	Scheduler	Yes	No
AQuoSA [14]	Framework	Yes	No
Alloc. Disp. [16]	Framework	No	No
RT-Linux [17]	Hypervisor	Yes	No
RTAI [11]	Hypervisor	Yes	No
RK [13]	Framework	Yes	No
Linux-SRT [12]	Framework	Yes	No
Firm RT [15]	Scheduler	Yes	No
Kurt [23]	Framework	Yes	No
HSF-VxWorks [24]	Scheduler	No	No
HSF-FreeRTOS [6]	Scheduler	No	No
HLS [4]	Framework	Yes	No

TABLE I
Overview of the related work.

III. System Model and Limitation

We assume that the task system is composed of periodic and/or sporadic tasks with single or multiple CPU cores. Each task τ_i is characterized by a tuple (C_i, D_i, T_i, pr_i), where C_i is the worst-case computation time, D_i is the relative deadline, T_i is the minimum inter-arrival time (period) and pr_i is the priority (lower value indicates a higher priority). The utilization of τ_i is also denoted by $U_i = C_i/T_i$. We particularly assume constrained-deadline systems that satisfy $C_i \leq D_i \leq T_i$ for any τ_i. When a task τ_i has $D_i > T_i$ then we transform 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 that is released at time t has its deadline at time $t + D_i$.

Schedulability tests and admission-control mechanisms are not within the scope of this paper. Although they are essential to guarantee that the system will run in a safe manner. We assume that the submitted task system is schedulable with the underlying scheduler. Integration of schedulability tests and admission-control mechanisms into ExSched are left open for future work.

IV. ExSched Framework

In this section, we present the ExSched framework that conceal platform details and provide high-level primitives for scheduler plug-ins. It also provides application programming interface (API) functions for user programs. Neither scheduler plug-ins nor user programs will access OS native functions. The core component of ExSched is a kernel-space module that controls the CPU scheduler via scheduler-related functions

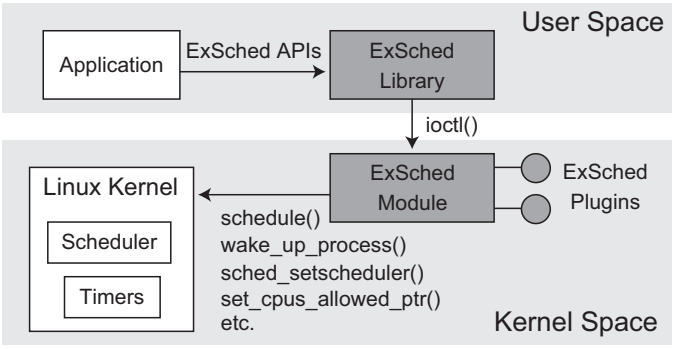


Fig. 1. The ExSched framework for Linux.

exported by the underlying OS. For instance, ExSched uses these functions to switch between tasks, migrate tasks to other CPU cores, and change the priorities of tasks.

Figure 1 illustrates the ExSched framework for Linux. The ExSched core is built as a character-device module and it is accessed through a device file `/dev/exsched`. User programs call the ExSched API functions provided by the ExSched user-space library. These calls are then relayed to the corresponding functions provided by the ExSched kernel module, using an `ioctl()` system call. The ExSched plug-in schedulers (if any are installed) are invoked by the core kernel-module through callback functions. The last step is to call appropriate scheduler-related functions (exported by the OS) that will schedule tasks in accordance with the given algorithm. It should be noted that KURT Linux [23] has a similar mechanism but it requires patches to the OS.

Our Linux version of ExSched uses a real-time scheduling class, i.e., `rt_sched_class`, to isolate real-time tasks from non-real-time tasks. Non-real-time tasks are scheduled by a fair scheduling class, i.e., `fair_sched_class`. It is also possible to use ExSched with the well-known RT-Preempt patch[1] if further isolation and low latency is required.

The VxWorks version of ExSched is similar to the Linux version except that the ExSched library and module reside in the kernel space by default. There is no need to provide `ioctl()` calls as VxWorks does not support user space mode. The internal functions exported by the OS are also different from those in Linux. However, VxWorks has the corresponding functions for scheduling tasks, migrating tasks etc. In addition, all tasks in VxWorks are real-time tasks; hence, we do not need multiple scheduling classes.

A. User API

Table II shows a basic set of API functions that ExSched provides for user programs. Figure 2 shows a sample C program, using these API functions. The program enters the real-time mode, using the `rt_enter()` call (not applicable in VxWorks). Next, the worst-case execution time, the period, the deadline, and the priority is set. Then, this task gets scheduled by ExSched immediately with no delay (`rt_run(0)`). The task submits `nr_jobs` number of jobs,

[1]RT-Preempt http://www.kernel.org/pub/linux/kernel/projects/rt/

rt_enter()	Change a caller to a real-time task.
rt_exit()	Change a caller to a normal task.
rt_run(timeout)	Start ExSched mode in @timeout time.
rt_wait_for_period()	Wait (sleep) for the next period.
rt_set_wcet(wcet)	Set the worst-case exec. time to @wcet.
rt_set_period(period)	Set the min. inter-arrival time to @period.
rt_set_deadline(deadline)	Set the relative deadline to @deadline.
rt_set_priority(priority)	Set the priority (1-99) to @priority.

TABLE II
BASIC EXSCHED API FUNCTIONS FOR USER PROGRAMS.

each of which executes the user's code in the `for` loop. It returns to the normal mode, using the `rt_exit()` API call (not applicable in VxWorks). We believe that our ExSched API is reasonable, given that many existing Linux-based real-time schedulers [11], [13], [15], [17] also use a similar API.

```
1: main(timeval C, timeval T, timeval D, int prio, int nr_jobs) {
2:         rt_enter();
3:         rt_set_wcet(C);
4:         rt_set_period(T);
5:         rt_set_deadline(D);
6:         rt_set_priority(prio);
7:         rt_run(0);
8:         for (i = 0; i < nr_jobs; i++) {
9:                 /* User's code. */
10:                rt_wait_for_period();
11:        }
12:        rt_exit();
13: }
```

Fig. 2. Sample code using the ExSched API.

B. Management of Timing Properties

We must attach timing properties to each task, for example release time, deadline, WCET etc., in order to schedule them as real-time tasks. However, neither Linux nor VxWorks have task descriptors that contain members that relate to these timing properties. Although these members might be supported in future versions, the current available versions of the underlying OS will be strictly limited without the timing properties. Hence, ExSched has its own task descriptor in the core module. Figure 3 shows the Linux version of the ExSched task-descriptor. The `task` field is a pointer to the original task descriptor provided by Linux.

C. Basic Approach to Real-Time Scheduling

This section presents the ExSched approach in using OS native functions to schedule real-time tasks. The implementation of the ExSched core module depends on the OS platform. The following presents our prototype implementations for Linux and VxWorks.

1) Linux: Linux provides two POSIX-compliant scheduling policies for real-time tasks: `SCHED_RR` and `SCHED_FIFO`. ExSched uses `SCHED_FIFO` that breaks ties for tasks with the same priority level in a first-in-first-out fashion. On the other hand, the priorities of tasks are managed by the ExSched

```
1: struct exsched_task_struct {
2:         struct task_struct *task;
3:         unsigned long wcet;
4:         unsigned long period;
5:         unsigned long deadline;
6:         unsigned long exec_time;
7:         unsigned long release_time;
8:         unsigned char flags;
9:         unsigned char server_id;
10: } exsched_task[NR_EXSCHED_TASKS];
```

Fig. 3. ExSched task descriptor.

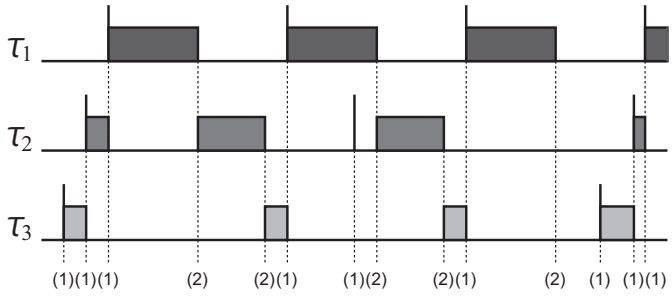


Fig. 4. Example of FPS with three tasks.

```
1: job_release(struct exsched_task_struct *p) {
2:     p->deadline += p->release_time;
3:     job_release_plugin(p);
4:     if ((p->flags & SET_BIT(PREVENT_RELEASE)) == 0)
5:         wake_up_process(p->task);
6:     else // User can prevent activation
7:         p->flags ^= SET_BIT(PREVENT_RELEASE);
8: }
9: sleep_in_period(struct exsched_task_struct *p) {
10:    setup_timer(timer, job_release, p);
11:    mod_timer(timer, p->release_time);
12:    p->task->state = TASK_UNINTERRUPTIBLE;
13:    schedule();
14:    del_timer(timer);
15: }
16: job_complete(struct exsched_task_struct *p) {
17:    p->release_time += p->period;
18:    job_complete_plugin(p);
19:    if (p->deadline < jiffies)
20:        sleep_in_period(p);
21: }
22: rt_run_internal(int k, int timeout) {
23:    exsched_task[k].release_time = jiffies + timeout;
24:    task_run_plugin(&exsched_task[k]);
25:    sleep_in_period(&exsched_task[k]);
26: }
27: rt_wait_for_period_internal(int k) {
28:    job_complete(&exsched_task[k]);
29: }
```

Fig. 5. ExSched functions for job release and completion.

module according to the given algorithms. ExSched uses the following functions exported by the Linux kernel (function names may differ slightly depending on kernel versions).

- `schedule()` switches the current execution context to the highest-priority task that is ready on the local CPU.
- `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 re-activates) the timer object so that it gets invoked when the timeout expires.
- `set_cpus_allowed_ptr(task,cpumask)` specifies the CPUs that the task is allowed to execute on and it is also used to force migrations of tasks.

It is important to understand that priority-driven schedulers require context switches only (i) when jobs with higher priority (than the current job) are released or (ii) when jobs complete their execution. Figure 4 depicts an example of FPS with three periodic tasks: τ_1, τ_2, and τ_3 (tasks with lower indices have higher priority). It is easy to observe that context switches only occur when there are job releases and job completions. This is marked with "(1)" and "(2)" respectively.

The previous discussion suggests that the `schedule()` function should be called when jobs are released or have completed, given that Linux already supports FPS. Figure 5 shows how and when ExSched invokes the `schedule()` function. It also shows how and when the plug-in interfaces are called.

The user task calls the `rt_wait_for_period()` API call (see Figure 2) every time a job completes. ExSched will then invoke the corresponding internal function `rt_wait_for_period_internal()`, which in turn calls `job_complete()`. This internal function calls `sleep_in_period()` which will suspend the task (using `schedule()`). The timer is associated with an internal function handler, `job_release()`, which is invoked at the next release time. On invocation, it awakens the task given by its argument (unless the user set `flags` in `job_release_plugin`), and the new job is released. The `sleep_in_period()` function is also called by an internal function, `rt_run_internal()`, which corresponds to the `rt_run()` API call.

The internal functions `rt_run_internal()`, `job_release()`, and `job_complete()` contain the `task_run_plugin`, `job_release_plugin`, and `job_complete_plugin` interfaces respectively. These plug-in interfaces are function pointers, which point to functions implemented by the user of the scheduler plug-ins. ExSched does not offer any further functions, i.e., plug-ins (Section V) can instead be used to extend the functionality.

Figure 6 illustrates a time-line flow from a job completion to a job release, including plug-in interface calls. A user task will be suspended when it calls the `rt_wait_for_period()` function, and it will be resumed at the next release time. Figure 6 illustrates an example sequence in which the priority of the task in focus is the highest among all ready tasks when it is released. Hence, it can preempt the preceding tasks at the release time. In this way, the FPS of periodic tasks is made possible without applying patches to the Linux kernel. Scheduler plug-ins can hook into the plug-in execution parts

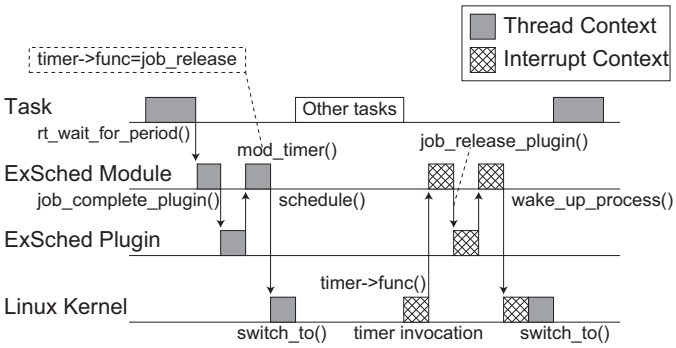


Fig. 6. Control flow in ExSched.

and thereby extend the scheduling functionality of ExSched.

Task Migration: The `set_cpus_allowed_ptr()` function, also know as `set_cpus_allowed` in earlier versions, is used to migrate tasks across CPU cores in Linux. However, there are two scenarios that we must take into account when migrating tasks. The simple scenario is when the task executes in the thread context. In this scenario, we can directly migrate the task. However, the other scenario is when the task executes in the interrupt context. We should not migrate the task directly, as it will trigger the `schedule()` function which is not allowed to be called in the interrupt context (unless Linux is built with the `CONFIG_PREEMPT` option). If the `CONFIG_PREEMPT` option is not set, then we create a real-time kernel thread with the highest priority that is awakened upon request to migrate the caller task. Henceforth, "`migrate_task(task,cpu)`" represents a procedure to migrate the given task to the specified CPU, using one of the two mechanisms described.

2) VxWorks: The base scheduler in VxWorks is fixed-priority driven, but it does not support periodic tasks. The ExSched API function `rt_wait_for_period()` is mapped to the VxWorks primitive `taskSuspend` which removes the calling task from the VxWorks ready-queue. The periodic releases of tasks is implemented using the VxWorks watchdog primitive `wdStart`. It calls an interrupt handler after a specified time has elapsed. Unlike in Linux, where the release list of tasks is implemented in the kernel by `mod_timer`, the VxWorks version of ExSched uses an internal bitmap-based queue for the management of task releases. The primitive `Q_PUT` is used to insert released ExSched tasks into the VxWorks ready-queue.

Task Migration: When it comes to CPU migration of tasks, we use the system call `taskCpuAffinitySet`. However, in VxWorks we face the same problem as in Linux that this primitive may not be called from the interrupt context. Hence, we apply the same technique here as in Linux. We simply release a high priority task to perform the task migration.

V. Plug-in Development

In this section, we describe how to develop plug-ins by showing some example schedulers developed in ExSched. It covers the implementation of in total six schedulers, where two are hierarchical schedulers and four of them are multicore schedulers. We want to emphasize the availability of ExSched and its simple usage for implementing different scheduling techniques. We managed to implement a variety of different scheduler algorithms using only the `task_run_plugin`, the `job_release_plugin`, and the `job_complete_plugin` interfaces (as well as timer management primitives).

A. Hierarchical Scheduling

We first provide the implementation of a 2-level hierarchical scheduler. This type of scheduler includes two schedulers. The *global* scheduler schedules virtual tasks that we refer to as *servers*. They are released periodically and they run a fixed time-length called *budget*. The variation of this scheduler is FPS and EDF. The second level scheduler (local scheduler), which resides within each server, schedules tasks based on FPS using the ExSched interface. The fundamental idea with hierarchical scheduling is that tasks should only execute within the time budget of their server.

```
1: struct server_struct {
2:         int id;
3:         int period;
4:         int budget;
5:         int priority;
6:         int remain_budget;
7:         unsigned long release_time;
8:         unsigned long budget_exp_time;
9:         unsigned long tstamp;
10:        struct timer_list timer;
11:        struct exsched_task_struct *task_list[NR_TASKS_IN_SERVER];
12: } SERVERS[NR_OF_SERVERS];
```

Fig. 7. ExSched server descriptor.

Figure 7 shows the descriptor of a server in ExSched. Line (11) shows a list of ExSched task descriptors of tasks that belong to this server. Lines (1) and (11) in Figure 8 represent two ExSched callback functions, i.e., the hierarchical scheduler plug-in will get notifications from the ExSched core about task releases and completions through these two functions. Line (7) will notify ExSched that it should not activate the task. Line (8) will notify the `server_release_handler()` function that it should activate this task at the corresponding servers next release. The functions on line (15) and (35) are interrupt handlers (they execute in the interrupt context). These two functions are responsible for releasing and suspending servers. These functions get triggered by server release and deplete events through timer activations which are initiated on lines (26-30), (39-42) and (49-52). The server ready-queue is implemented using bitmaps, i.e., in the same way as the Linux 2.6 native task ready-queue.

The EDF version of the hierarchical scheduler has a similar implementation as the FPS version. The EDF version stores the server absolute deadlines (instead of storing server priorities) in a bitmap queue, as to determine which server to execute. Hence, in the EDF version, lines (2), (17), (18), (37) and

```
1: void job_release_plugin(struct exsched_task_struct *rt) {
2:    high_prio_server = bitmap_get(&SERVER_READY_QUEUE);
3:    if (high_prio_server == NULL)
4:       goto setflags;
5:    if (SERVERS[rt->server_id].id != high_prio_server->id) {
6: setflags:
7:       rt->flags |= SET_BIT(PREVENT_RELEASE);
8:       rt->flags |= SET_BIT(ACTIVATE);
9:    }
10: }
11: void job_complete_plugin(struct exsched_task_struct *rt) {
12:  if ((rt->flags & SET_BIT(ACTIVATE))==SET_BIT(ACTIVATE))
13:     rt->flags ^= SET_BIT(ACTIVATE);
14:}
15: void server_release_handler(unsigned long __data) {
16:  struct server_struct *released_server = (struct server_struct *)__data;
17:  high_prio_server = bitmap_get(&SERVER_READY_QUEUE);
18:  bitmap_insert(&SERVER_READY_QUEUE, released_server);
19:  if (high_prio_server == NULL)
20:     goto settimer;
21:  if (released_server->priority < high_prio_server->priority) {
22:     high_prio_server->remain_budget-=jiffies-high_prio_server->tstamp;
23:  // Deactivate the preempted server (deactivate its tasks etc.)
24: settimer:
25:     // Activate the released server (activate its tasks etc.)
26:     setup_timer_on_stack(&(released_server->timer),
27:     server_complete_handler, (unsigned long)released_server);
28:     released_server->budget_exp_time=jiffies+released_server->budget;
29:     mod_timer(&(released_server->timer),
30:     released_server->budget_exp_time);
31:     released_server->tstamp = jiffies;
32:     released_server->remain_budget = released_server->budget;
33:  }
34:}
35: void server_complete_handler(unsigned long __data) {
36:  struct server_struct*completed_server=(struct server_struct *)__data;
37:  bitmap_retrieve(&SERVER_READY_QUEUE);
38:  completed_server->release_time += completed_server->period;
39:  setup_timer_on_stack(&(completed_server->timer),
40:  server_release_handler, (unsigned long)completed_server);
41:  mod_timer(&(completed_server->timer),
42:  completed_server->release_time);
43:  // Deactivate the completed server (deactivate its tasks etc.)
44:  high_prio_server = bitmap_get(&SERVER_READY_QUEUE);
45:  if (high_prio_server != NULL) {
46:     // Activate the high priority server (activate its tasks etc.)
47:     high_prio_server->budget_exp_time = jiffies +
48:     high_prio_server->remain_budget;
49:     setup_timer_on_stack(&(high_prio_server->timer),
50:     server_complete_handler,(unsigned long)high_prio_server);
51:     mod_timer(&(high_prio_server->timer),
52:     high_prio_server->budget_exp_time);
53:     high_prio_server->tstamp = jiffies;
54:  }
55:}
```

Fig. 8. Hierarchical scheduler.

(44) in Figure 8 are replaced with a bitmap queue that stores absolute deadlines of servers. Our EDF version is similar to the SCHED_DEADLINE [5] and VSCHED [20] schedulers.

B. Multi-core Scheduling

We next provide the implementations of our multi-core schedulers. We will assume FPS algorithms for the sake of simplifying this description. Specifically, we provide four multi-core scheduler plug-ins; G-FP, FP-US, FP-FF, and FP-PM. G-FP and FP-US are based on multi-core global scheduling, while FP-FF and FP-PM are based on partitioned and semi-partitioned scheduling respectively. More details will be provided in the rest of this section. For the sake of simplifying our presentation, we represent the plug-in functions pointed to by the `task_run_plugin()`, `job_release_plugin()`, and `job_complete_plugin`, as `task_run_X()`, `job_release_X()`, and `job_complete_X()` respectively, where '`X`' denotes the plug-in name.

1) Partitioned Scheduling: For partitioned scheduling, we developed a plug-in called FP-FF, which adopts a first-fit heuristic to assign tasks to CPUs. The plug-in implementation is straightforward. FP-FF 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 tries to find a CPU that can accommodate the given task, by using the response-time analysis [25]. The task is then migrated to the CPU that is verified first, using the `migrate_task()` function. A task starts to execute in the background if it cannot be assigned to any CPU.

2) Semi-Partitioned Scheduling: For semi-partitioned scheduling, we developed a plug-in called FP-PM, which adopts to the migration policy of the DM-PM algorithm [26]. It allows tasks to migrate across multiple CPUs if the task cannot be assigned to any CPU by a first-fit allocation. This migratory task is statically assigned the highest priority. The maximum CPU time that the migratory task is allowed to consume on each CPU is computed based on a response-time analysis. The task migrates to another CPU (on which it is assigned CPU time) once it consumes the assigned CPU time on a CPU. Other tasks are scheduled in the same manner as FP-FF (for more details see [26]).

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

FP-PM migrates tasks to the CPU that has the lowest index in the list of assigned CPUs when a job of a migratory task is released, using the `migrate_task()` function. FP-PM also activates a timer that triggers when the assigned processing time is consumed on a CPU. This will migrate the task to the next CPU. It activates a timer again for the next migration event when the migration is completed, and so on. Timer invocations are continued until the task is migrated to the CPU that has the largest index among the assigned CPUs. Timing information that is related to the activation of timers resides within the plug-in module space.

The task migration to the first CPU (at the time of the release) can alternatively be done when jobs complete instead,

using the job_complete_plugin interface. This is suitable for systems that are sensitive to job-release overhead.

3) Global Scheduling: We have also developed two plug-ins called G-FP and FP-US for global scheduling. The G-FP algorithm simply dispatches tasks in a global scheduling fashion, according to the given priorities. On the other hand, FP-US classifies tasks as heavy and light tasks, based on the utilization factors. A task is categorised as a heavy task if the CPU utilization of the task is greater than or equal to $m/(3m-2)$. It is marked as a light task in any other case. All heavy tasks are statically assigned the highest priorities, while light tasks keep the original priorities. This idea has been proposed in [27].

The plug-ins are implemented in such a way that they still use local schedulers like partitioned scheduling. However, they imitate global scheduling using the job_release_plugin() and job_complete_plugin() interfaces. We first focus on the implementation of G-FP.

G-FP starts to seek a CPU that is currently not executing any real-time task when a job of a real-time task is released. It uses the job_release_G-FP() function to accomplish this. The task will be migrated if such a CPU exists. If none exist then G-FP checks if there are CPUs that are currently executing real-time tasks with lower priorities than this task. The task will be migrated to the CPU that is executing the lowest-priority tasks if this scenario is true. In any other case, G-FP will do nothing for this task. The task may later be migrated to another CPU in the job_complete_G-FP() function as soon as another job completes.

G-FP migrates the task that has the highest priority (not including the current tasks), if it exists, when a job of a real-time task completes. It is migrated to the CPU upon which the completed job has been running on. This is done using the function job_complete_G-FP().

We additionally create the task_run_FP-US() function which can classify heavy and light tasks (in the case of FP-US). The priority assignment of heavy tasks is also processed in this function.

We imitate global scheduling since local schedulers always dispatch the highest-priority task in their own runqueue and we assume that every runqueue contains one of the m highest-priority tasks. We only need a global task-list that contains ready tasks, ordered by priorities. However, the global task-list must be protected by a lock which introduces overhead.

VI. Experimental Evaluation

We demonstrate our experiments in this section in order to show the runtime performance of ExSched. We will show the performance of our hierarchical schedulers in both Linux and VxWorks, and also our multi-core schedulers in Linux. In particular, we have observed the overhead of our hierarchical schedulers, showing that our ExSched approach has a limited performance penalty. We also show that our multi-core scheduling algorithms, implemented using the ExSched framework, perform as expected compared to the previous work on analysis and simulations.

A. Scheduler Overhead in VxWorks

We have conducted experiments with our FPS and EDF hierarchical schedulers. We measured the overhead of these schedulers and compared the results against an equivalent scheduler [24]. The HSF scheduler [24] is the only similar scheduler that we can find for the VxWorks platform.

1) Experimental Setup: We used the platform VxWorks 6.6 on a single-core Pentium4. The measurements were done using VxWorks timestamp libraries. Neither the VxWorks scheduler nor task context-switches were included in these overhead measurements (this gives a fair comparison). We ran 2-8 servers with 1-10 (synthetic) tasks in each server. Every experiment ran for 4 minutes. Server periods were in the range of 5-20 and task periods 50-150 milliseconds.

2) Results: Our results are presented in Figure 9. The difference in overhead should mostly depend on the queue management. The HSF schedulers [24] are based on the median linked-list implementation which has good performance when the number of queue elements are below 50 [28]. Our previous studies [29] confirm that median linked-list queues have good performance when the number of elements are low. However, our experiments (Figure 9) indicate that the ExSched bitmap-based schedulers outperform HSF [24], both for FPS and EDF. The overhead rarely peeks, and it climps steadily as the number of servers and tasks increase. The reason for this increase is due to that the scheduler executes more frequently when there are more entities (tasks/servers) to schedule.

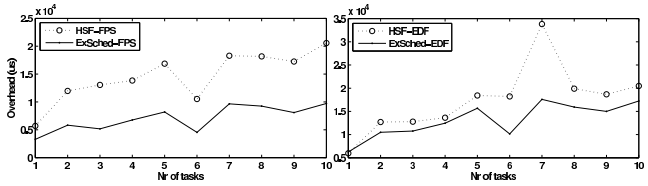
B. Scheduler Overhead in Linux

The second experiment was done in Linux and we compared our EDF hierarchical scheduler with the SCHED_DEADLINE [5] scheduler. We have chosen to compare our scheduler against SCHED_DEADLINE because it resembles our EDF scheduler in that it also schedules servers with the EDF algorithm. We deactivated our local FPS-scheduler in order to make our EDF scheduler comparable to the SCHED_DEADLINE scheduler.

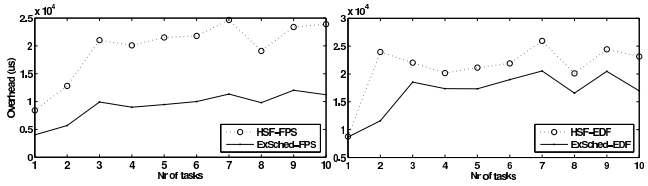
1) Experimental Setup: The platform used for this experiment was a Linux-kernel (version 2.6.36) patched with the SCHED_DEADLINE scheduler (we used the latest stable release from the SCHED_DEADLINE project at the time of writing this paper). We used a dual-core Pentium4 hardware-platform (only 1 core was used in these experiments).

The measurements in ExSched-EDF were accomplished by timestamping the execution of the two interrupt handlers server_release_handler and server_complete_handler (Figure 8) which are responsible for server releases and budget depletions respectively.

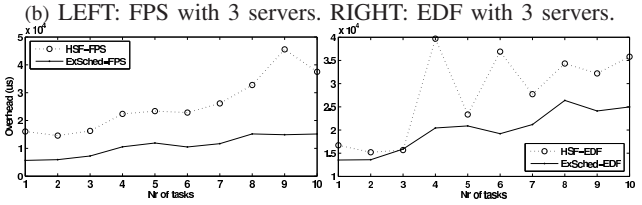
We patched the SCHED_DEADLINE kernel with timestamp functions in locations related to the resource-reservation mechanism (in order to measure the execution time). We also instrumented the dl_task_timer timer-handler function and part of the update_curr_dl function



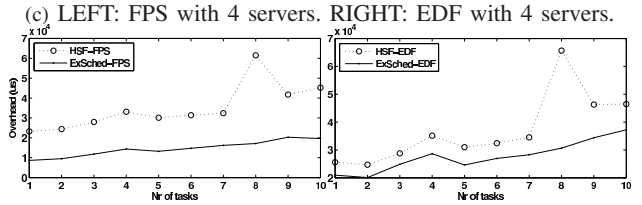
(a) LEFT: FPS with 2 servers. RIGHT: EDF with 2 servers.



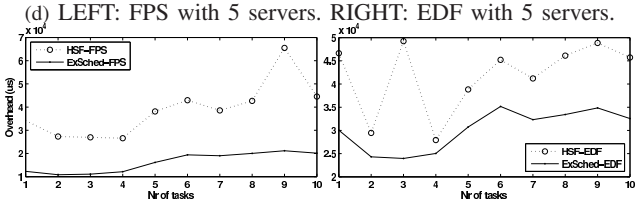
(b) LEFT: FPS with 3 servers. RIGHT: EDF with 3 servers.



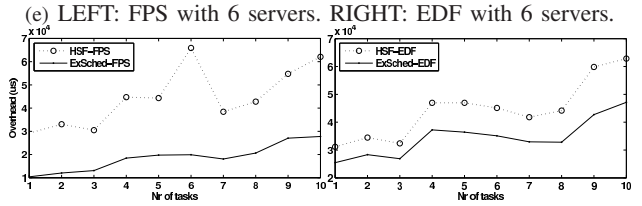
(c) LEFT: FPS with 4 servers. RIGHT: EDF with 4 servers.



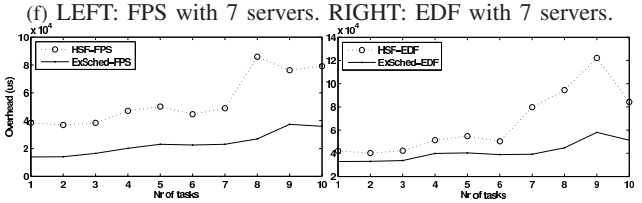
(d) LEFT: FPS with 5 servers. RIGHT: EDF with 5 servers.



(e) LEFT: FPS with 6 servers. RIGHT: EDF with 6 servers.



(f) LEFT: FPS with 7 servers. RIGHT: EDF with 7 servers.



(g) LEFT: FPS with 8 servers. RIGHT: EDF with 8 servers.

Fig. 9. Overhead measuring (in microseconds) of the ExSched and HSF scheduler in VxWorks 6.6.

in the SCHED_DEADLINE scheduling class (sched_dl.c). dl_task_timer is related to the enforcement of resource reservation (similar to our two interrupt handlers) and the part in update_curr_dl relates to the checking of server deadlines and exceeded budget executions.

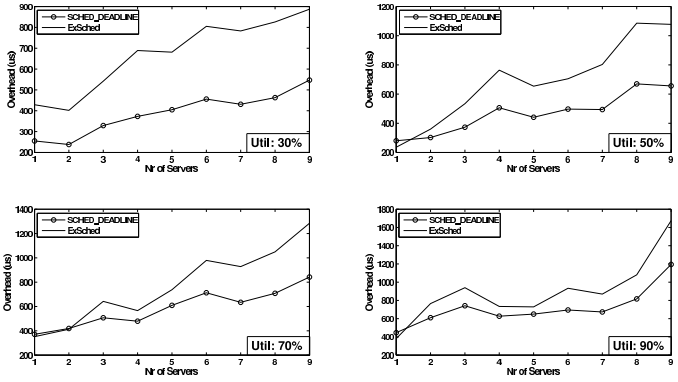


Fig. 10. Overhead measurements of the ExSched and SCHED_DEADLINE schedulers in Linux 2.6.36.

We ran 2 to 10 servers (with 1 synthetic task per server) with system utilization ranging from 30% to 90%. The server period interval was set to 10-160ms. We ran in total 28 experiments with one server configuration in each experiment. Each experiment was conducted twice and the presented values represent the average of the two.

2) Results: Figure 10 shows the overhead-measurement results of the two schedulers when running 2 to 10 servers with system utilization 30%, 50%, 70% and 90%. The overhead of ExSched with respect to SCHED_DEADLINE tends to decrease when the system utilization increases. ExSched has a maximum of 181% more overhead than SCHED_DEADLINE at 30% utilization. It drops to 164% (at 50% utilization), and later 152% (at 70% utilization). Finally, at 90% utilization, the maximum diff is only 140%. Another observation is that SCHED_DEADLINE is more efficient when there are more servers. The conclusion is that there is a performance penalty to pay when having a kernel modification-free solution like ExSched. We have shown that this penalty cost is in average (of all maximums) 160% of overhead compared to SCHED_DEADLINE.

C. Multi-core Scheduler Performance

In this section we evaluate the performance of our multi-core scheduler plug-ins. Specifically, we measure the run-time schedulability of FP-FF, FP-PM, G-FP, FP-US, and FP. FP is a plain Linux `SCHED_FIFO` scheduler, which reflects the performance of the native Linux scheduler. Priority assignments are based on the Deadline Monotonic (DM) algorithm [2]. A note to the reader is that these experiments are not simulations, i.e., the experiments are conducted in a real Linux kernel running on a multi-core hardware platform.

1) Experimental Setup: Our experiments are conducted in the Linux kernel 2.6.29.4 running on two 3.16GHz Intel Xeon CPUs (X5460). Each CPU contains four cores, hence, the machine includes eight CPU cores in total. In order to assess the schedulability, we submit many sets of randomly generated (synthetic) busy-loop periodic tasks to the system. We then observe the ratio of task sets that are successfully scheduled without missing their deadlines.

The generated sets of periodic tasks are similar to the ones

employed in the previous work [8], [9], [10]. We submit 1000 sets of periodic tasks. In order to measure the schedulability for the given workload we let each set produce the same amount of workload W. Each task set is generated as follows. The CPU utilization U_i of a newly generated task τ_i is determined based on a uniform distribution. The range of the distribution is parametric. We have three test cases in our evaluation: $[10\%, 100\%]$ (both heavy and light tasks), $[10\%, 50\%]$ (only light tasks), and $[50\%, 100\%]$ (only heavy tasks). New tasks are created until the total CPU utilization reaches W. The period T_i of τ_i is also uniformly determined within the range of $[1ms, 100ms]$. The execution time of τ_i is set to $C_i = U_i T_i$.

We measure the count n of busy-loops that consume 1 microsecond. Each task τ_i then loops $n \times C_i$ iterations in each period. We execute these busy-loop tasks for 10 minutes. A task-set is said to be successfully scheduled if and only if all jobs complete within their periods during the measurements. We then evaluate by the success ratio: *the ratio of the number of successfully scheduled task-sets with respect to the total number of submitted task-sets*.

2) Results: Figure 11 shows the experimental results. FP-PM has better performance than the others in most cases. Semi-partitioned scheduling is a superset of partitioned scheduling, hence, 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 the schedulability compared to the one implemented in the Linux kernel. G-FP is usually better than FP while it is worse than FP-FF. This observation leads to the conclusion that partitioned scheduling may be inferior to global scheduling and vice versa, depending on the CPU allocation methods. Meanwhile, FP-US shows the worst performance of all the tested plug-ins. This is reasoned as follows. FP-US assigns 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 to be long. As a result, this heavy task can block light tasks that have much shorter deadline than their execution time. This results in deadline misses for light tasks. Therefore, FP-US can be worse than G-FP in the average case, even though its worst-case schedulability is higher than G-FP [27], [30].

The performance of the scheduler plug-ins is dependent on the range (Umin, Umax) of every individual tasks utilization. G-FP suffers from Dhall's effect [3] and tends to perform poorly as compared to other cases when task sets contain both light and heavy tasks. FP-US is designed to avoid Dhall's effect but it also shows poor performance due to the reason stated previously. On the other hand, the schedulability of FP-FF and FP can decline when tasks are likely to be heavy, as in the case of Figure 11 (i). An extreme example which reasons about this performance degradation is when we consider $m+1$ tasks with utilization $(50+\alpha)\%$. It is clear that one of the $m+1$ tasks can not be successfully assigned to any of the CPUs. In this case, it is inevitable for FP-FF and FP to cause deadline misses. FP-PM overcomes this issue by using migrations.

The number of CPUs will also affect the schedulability. In most cases, the performance of the scheduler plug-ins (except for FP-PM) decline as the number of CPUs increase. This result is natural since the theoretical schedulable bound for partitioned and global scheduling is a function of the number of CPUs. An increase in the CPU count results in a decrease in the bound [27], [30], [31], [32], [33]. Thus, the runtime performance reflects the theory. On the other hand, FP-PM utilizes the CPUs effectively by using task migrations. In fact, the more CPUs that are given, the greater is the chance that FP-PM meets the task deadlines.

VII. Conclusion

We have presented ExSched: a platform and scheduling-policy independent scheduler framework for real-time systems. It supports the development of different scheduling techniques on different OS platforms. Our prototype implementation of ExSched supports hierarchical and multi-core schedulers in Linux and VxWorks. We have presented the overhead measurements of ExSched with experimental results. The multi-core scheduling algorithms implemented as ExSched plug-ins perform as studied in theory. To the best of our knowledge, this is the first real-time scheduler framework that achieves both portability across different OS platforms, and availability for different scheduling techniques. We believe that ExSched is a useful contribution for the real-time systems community in terms of transforming well-studied theory into practice.

The future work includes the development of ExSched to support more OS platforms and scheduling techniques. Further, we will also extend ExSched to support shared resources for both tasks and servers.

References

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

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

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

[4] J. Regehr and J. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *RTSS'01*, 2001.

[5] D. Faggioli, M. Trimarchi, and F. Checconi, "An implementation of the Earliest Deadline First algorithm in Linux," 2009.

[6] R. Inam, J. Maki-Turja, M. Sjodin, S. Ashjaei, and S. Afshar, "Support for Hierarchical Scheduling in FreeRTOS," in *ETFA'11*, 2011.

[7] A. Bastoni, B. Brandenburg, and J. Anderson, "Is Semi-Partitioned Scheduling Practical?" in *ECRTS'11*, 2011.

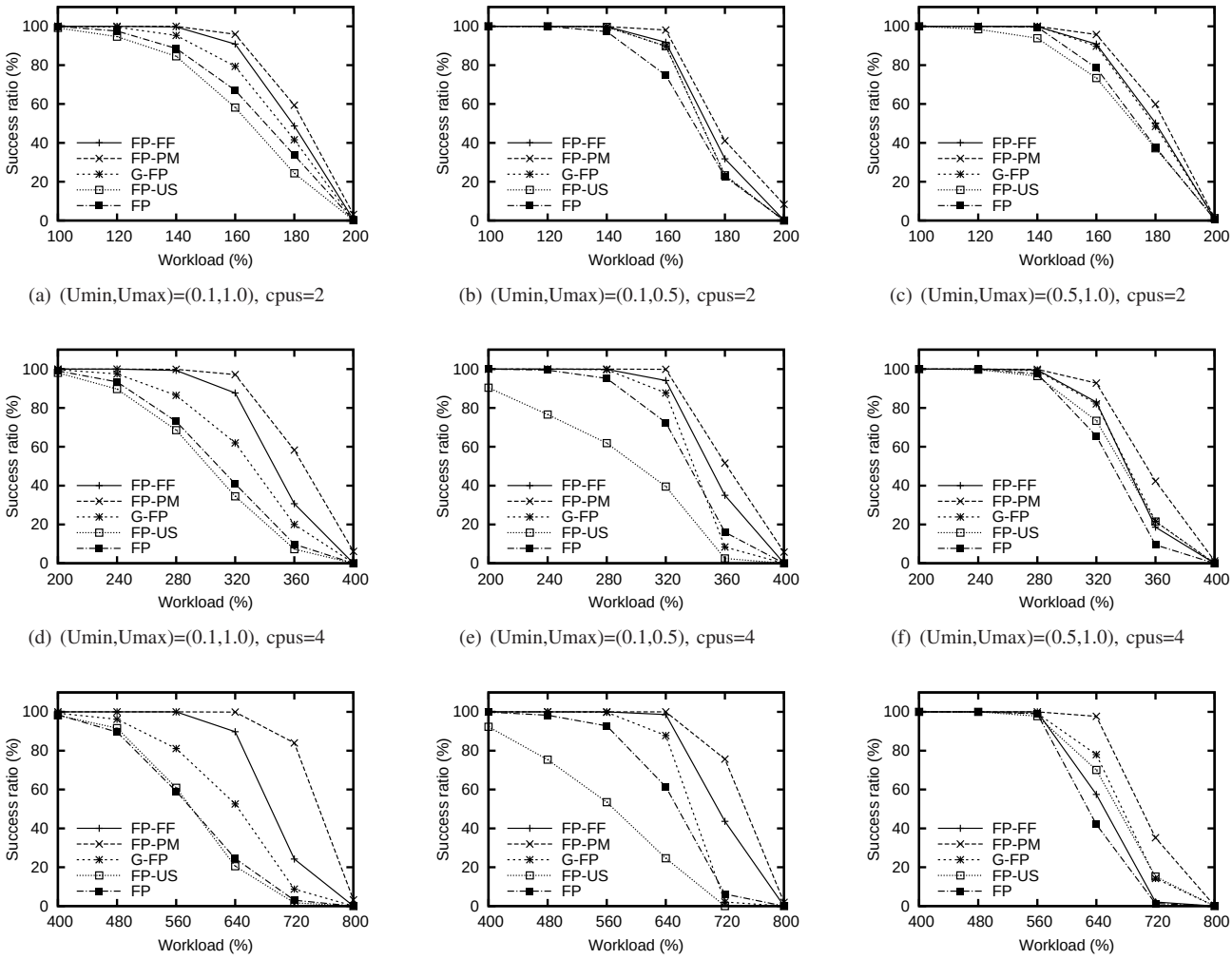
[8] B. Brandenburg, J. Calandrino, and J. Anderson, "On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study," in *RTSS'08*, 2008.

[9] B. Brandenburg and J. Anderson, "On the Implementation of Global Real-Time Schedulers," in *RTSS'09*, 2009.

[10] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS$^{\mathrm{RT}}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *RTSS'06*, 2006.

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

[12] S. Childs and D. Ingram, "The Linux-SRT Integrated Multimedia Operating Systems: Bringing QoS to the Desktop," in *RTAS'01*, 2001.



(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

Fig. 11. Schedulability results for multi-core schedulers.

[13] S. Oikawa and R. Rajkumar, "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior," in *RTAS'99*, 1999.

[14] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA: Adaptive Quality of Service Architecture," *Software Practice and Experience*, vol. 39, pp. 1–31, 2009.

[15] 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 *RTAS'98*, 1998.

[16] Y. Wang and K. Lin, "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," in *RTSS'99*, 1999.

[17] V. Yodaiken, "The RTLinux Manifesto," in *Linux Expo*, 1999.

[18] R. Lehrbaum, "Using Linux in Embedded and Real-Time Systems," *Linux Journal*, no. 75, 2000.

[19] G. Parmer and R. West, "Hijack: Taking Control of COTS Systems for Real-Time User-Level Services," in *RTAS'07*, 2007.

[20] B. Lin and P. A. Dinda, "VSched: Mixing Batch and Interactive Virtual Machines Using Periodic Real-time Scheduling," in *SC'05*, 2005.

[21] M. Åsberg, N. Forsberg, T. Nolte, and S. Kato, "Towards Real-Time Scheduling of Virtual Machines Without Kernel Modifications," in *W.I.P. session in ETFA'11*, 2011.

[22] K. Yaghmour, "Adaptive Domain Environment for Operating Systems," *Opersys inc*, 2001.

[23] A. Atlas and A. Bestavros, "Design and Implementation of Statistical Rate Monotonic Scheduling in KURT Linux," in *RTSS'99*, 1999.

[24] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. Bril, "Towards Hierarchical Scheduling in VxWorks," in *OSPERT'08*, 2008.

[25] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority preemptive scheduling," *Software Engineering Journal*, vol. 8, pp. 285–292, 1993.

[26] S. Kato and N. Yamasaki, "Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors," in *RTAS'09*, 2009.

[27] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority Scheduling on Multiprocessors," in *RTSS'01*, 2001.

[28] R. Rönngren and R. Ayani, "A Comparative Study of Parallel and Sequential Priority Queue Algorithms," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, pp. 157–209, 1997.

[29] M. Åsberg, "Comparison of Priority Queue algorithms for Hierarchical Scheduling Framework," Malardalen University, Nr. 2598, 2011.

[30] T. Baker, "An Analysis of Fixed-Priority Schedulability on a Multiprocessor," *Real-Time Systems*, vol. 32, pp. 49–71, 2006.

[31] T. P. Baker, "Comparison of Empirical Success Rates of Global vs. Partitioned Fixed-Priority and EDF Scheduling for Hard Real Time," Dep. of Computer Science, Florida State University, TR-050601, 2005.

[32] J. Lopez, J. Diaz, and D. Garcia, "Minimum and Maximum Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, pp. 642–653, 2004.

[33] J. Lopez, M. Garcia, J. Diaz, and D. Garcia, "Utlization Bounds for Multiprocessor Rate-Monotonic Scheduling," *Real-Time Systems*, vol. 24, pp. 5–28, 2003.