Implementation and Empirical Comparison of Partitioning-based Multi-core Scheduling

Yi Zhang†, Nan Guan†‡, Yanbin Xiao†, Wang Yi†‡

†Department of Computer Science and Technology, Northeastern University, China
‡Department of Information Technology, Uppsala University, Sweden

***Abstract*—Recent theoretical studies have shown that partitioning-based scheduling has better real-time performance than other scheduling paradigms like global scheduling on multi-cores. Especially, a class of partitioning-based scheduling algorithms (called semi-partitioned scheduling), which allow to split a small number of tasks among different cores, offer very high resource utilization. The major concern about the semi-partitioned scheduling is that due to the task splitting, some tasks will migrate from one core to another at run time, which incurs higher context switch overhead. So one would suspect whether the extra overhead caused by task splitting would counteract the theoretical performance gain of semi-partitioned scheduling.**

In this work, we implement a semi-partitioned scheduler in the Linux operating system, and run experiments on an Intel Core-i7 4-cores machine to measure the real overhead in both partitioned scheduling and semi-partitioned scheduling. Then we integrate the measured overhead into the state-of-the-art partitioned scheduling and semi-partitioned scheduling algorithms, and conduct empirical comparisons of their real-time performance. Our results show that the extra overhead caused by task splitting in semi-partitioned scheduling is very low, and its effect on the system schedulability is very small. Semi-partitioned scheduling indeed outperforms partitioned scheduling in realistic systems.

I. Introduction

It has been widely believed that future real-time systems will be deployed on multi-core processors, to satisfy the dramatically increasing high-performance and low-power requirements. There are two basic approaches for scheduling real-time tasks on multiprocessor/multi-core platforms [1]: In the *global* approach, each task can execute on any available processor at run time. In the *partitioned* approach, each task is assigned to a processor beforehand and during run time it can only execute on this particular processor. Recent studies showed that the partitioned approach is superior in scheduling hard real-time systems, for both theoretical and practical reasons. However, partitioned scheduling still suffers from resource waste similar to the bin-packing problem: a task would fail to be partitioned to any of the processors when the total available capacity of the whole system is still large. When the individual task utilization is high, this waste could be significant, and in the worst-case only half of the system resource can be used.

To overcome this problem, recently researchers have proposed *semi-partitioned scheduling* [2], [3], [4], [5], [6], [7], in which most tasks are statically assigned to one fixed processor as in partitioned scheduling, while a few number of tasks are split into several subtasks, which are assigned to different processors. Theoretical studies have shown that semi-partitioned scheduling can significantly improve the resource utilization over partitioned scheduling, and appears to a promising solution for scheduling real-time systems on multi-cores.

While there have been quite a few works on implementing global and partitioned scheduling algorithms in existing operating systems and studying their characterizations like run-time overheads, the study of semi-partitioned scheduling algorithms is mainly on the theoretical aspect. The semi-partitioned scheduling has not been accepted as a mainstream design choice due to the lack of evidences on its practicability. Particularly, in semi-partitioned scheduling, some tasks will migrate from one core to another at run time, and might incur higher context switch overhead than partitioned scheduling. So one would suspect whether the extra overhead caused by task splitting would counteract the theoretical performance gain of semi-partitioned scheduling.

In this work, we answer the question on the practicability of semi-partitioned scheduling by implementing RTPS (Real-Time Partitioning-based Scheduler) in Linux to support semi-partitioned scheduling, and evaluate its actual real-time performance. By a careful design, RTPS has very low run-time overhead although non-trivial mechanisms are instrumented to support task migration. Then we measure the RTPS's realistic run-time overhead as well as the cache related overhead on an Intel Core-i7 4-cores machine. Finally we integrate the measured overhead into empirical comparison of the state-of-the-art partitioned scheduling and semi-partitioned scheduling algorithms. Our experiments show that semi-partitioned scheduling indeed outperforms partitioned scheduling in the presence of realistic run-time overheads.

The remainder of this paper is organized as follows: Section II reviews related works; Section III introduces the background on semi-partitioned scheduling; Section IV presents the implementation of the semi-partitioned

scheduler on Linux; Section V identifies the run-time overheads. Section V presents the overhead measurement results. Section VII introduces the empirical comparison experiments, and finally conclusions are drawn in Section VIII.

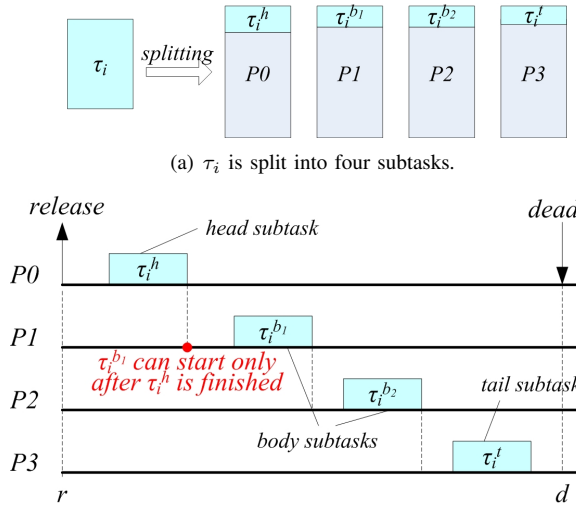
II. Related Work

Although Linux was originally designed for general-purpose computer systems, more and more real-time supports have been adopted during its evolution. For example, the 2.6 kernel has taken a big step of making the kernel almost fully preemptive and providing the "preemptive" compiling option, which leads significant improvements to the responsiveness. Nowadays, it has been widely accepted that the standard Linux can be used for many real-time applications (at least soft real-time applications). Some Linux real-time extensions can provide competitive real-time performance compared with other commercial real-time operating systems [8], [9], [10].

Many works have been done to implement and evaluate the state-of-the-art real-time scheduling research efforts in Linux on single-core machines. For example, SCHED_EDF [11] implements the Earliest Deadline First (EDF) algorithm by adding new scheduling class into Linux and uses the Constant Bandwidth Server (CBS) to allocate the execution time for each task. However, the newly added scheduling class is "lower" than POSIX real-time scheduling class `sched_rt` and would be interfered with POSIX real-time tasks.

The UNC research team led by James Anderson developed LITMUS^{RT}[12] as a testbed for evaluating scheduling algorithms and synchronization on multi-core platforms. Based on LITMUS^{RT}, they performed a series of empirical works to compare the performance of the state-of-art multiprocessor scheduling algorithms on several different multi-core machines [13], [14], [15]. One of the major conclusions of these works is that, the global scheduling, even with a very careful design, is clearly inferior to partitioned schedulers, because global scheduling incurs heavy contentions on the scheduler and frequently task migration.

Shinpei [16] implements a loadable real-time scheduler suite RESCH to support different scheduling algorithms on multi-core systems, including semi-partitioned scheduling. RESCH can be installed into Linux without patches. RESCH employs a special *migration thread* to operate task migrations, which would introduce extra overheads and degrade the responsiveness of the tasks managed by RESCH. For example, the POSIX FIFO real-time task would delay the migration thread, which makes the task migration overhead to be unpredictable. Due to its high and unpredictable run-time overhead, RESCH would not be suitable for hard real-time systems. In contrast, we implement the semi-partitioned scheduler by Linux scheduling class for a low and predictable run-time overhead.



(a) τ_i is split into four subtasks.

(b) The subtasks need synchronization to execute correctly.

Fig. 1. Illustration of task splitting

III. Semi-Partitioned Scheduling

In this section we will introduce the background on semi-partitioned scheduling. We start with the task model. We use τ to denote a task set consisting of N independent periodic tasks. Each periodic task τ_i is a tuple $\langle C_i, T_i \rangle$, where C_i is the worst-case execution time (WCET) and T_i is the minimal inter-release separation (period) of τ_i. We assume the implicit deadline model, i.e., T_i is also the relative deadline of τ_i.

A semi-partitioned scheduling algorithm consists of two parts: the *partitioning algorithm*, which determines how to split and assign each task (or rather each part of it) to a fixed processor, and the *scheduling algorithm*, which determines how to schedule the tasks assigned to each processor.

With the partitioning algorithm, most tasks are assigned to a processor and only execute on this processor at run time. We call them *non-split tasks*. The other tasks are called *split tasks*, since they are split into several *subtasks*. Each subtask of split task τ_i is assigned to (thereby executes on) a different processor, and the sum of the execution time of all subtasks equals C_i. For example, in Figure 1 the task τ_i is split into four subtasks executing on processor P_0, P_1, P_2 and P_3, respectively. The first subtask is called the *head subtask*, denoted by τ_i^h, and the last subtask is called the *tail subtask*, denoted by τ_i^t. Each of the other subtasks is called a *body subtask*, and the j^{th} body subtask is denoted by $\tau_i^{b_j}$. The subtasks of a task need to be synchronized to execute correctly. For example, in Figure 1, $\tau_i^{b_j}$ can not start execution until τ_i^h is finished.

In this paper, we focus on (task-level) fixed priority scheduling. Note that our scheduler implementation and overhead measurement can be easily applied to other scheduling paradigms like EDF as well. Several fixed-priority semi-partitioned algorithms have been proposed

[4]. In this work we adopt a recent developed algorithm FP-TS [4], which has both high worst-case utilization guarantees (can achieve hight utilization bounds) and good average-case real-time performance (exhibits high acceptance ratio in empirical evaluations). In the following we give a brief description of FP-TS's work flow. More details about FP-TS can be found in [4].

- FP-TS assigns tasks to processors in increasing priority order. FP-TS always selects the processor on which the total utilization of tasks that have been assigned so far is *minimal* among all processors.
- A task (subtask) can be entirely assigned to the current processor, if all tasks including this one on this processor can meet their deadlines.
- When a task (subtask) can not be assigned entirely to the current selected processor, FP-TS splits it into two parts. The first part is assigned to this processor. The splitting is done such that the portion of the first part is as big as possible guaranteeing no task on this processor misses its deadline; the second part is left for the assignment to the next selected processor.

IV. Scheduler Implementation

We implement RTPS as the highest-priority Linux scheduling class. Each time the Linux scheduler is invoked, it first checks whether there is any task in RTPS to be scheduled for execution. Therefore the real-time tasks in RTPS will not be interfered by tasks in other scheduling classes. Further, RTPS is implemented as a SMP (Symmetric Multi-processor) scheduler, i.e., each core runs an instance of the scheduler code, and uses the shared-memory abstraction for data consistency.

A good real-time schedulers should meet two requirements: (1) high timing resolution, and (2) low run-time overhead. In the following, we will present the implementation detail of RTPS, to show how these two requirements are met in RTPS.

A. Event-driven Scheduling based on HRtimer

The original Linux periodic scheduler is based on the *tick-driven mechanism*, which is not suitable for hard real-time tasks. The dilemma in tick-driven schedulers is that, a long tick period causes a large response delay which may invalidate the timing requirement of high-frequency tasks, while a short tick period causes very often scheduler invocation which increases the run-time overhead.

In the contrast, our Real-time Partitioning-based Scheduler (RTPS) employs the *event-driven mechanism*. RTPS uses the High-Resolution timer (HRtimer) to maintain the temporal information. The HRtimer is a per-core hardware counter with nanosecond precision, which has been adopted in most mainstream processor architectures. Each HRtimer is related with a time-ordered event tree. When it reaches the counting time of current pending event, the current event's callback function is invoked and the next pending event is reloaded. In the following we will introduce how to manage the scheduling events by the one-per-core HRtimers.

There are two types of scheduling events needed to be tracked by the HRtimer. The first type is *task release*, i.e., the HRtimer should keep track of the future time points when a task should be released. Since tasks are periodic, the time of all releases of a task are actually fixed (as long as we know the first release time). Keeping track of task release events is easy: when a task τ_i is released at time t, its next release event is simply calculated by $t + T_i$.

The second type of scheduling events is *budget expiration*. Suppose τ_i is a split task with two parts τ_i^1 and τ_i^2, hosted by core P_1 and P_2 respectively, and the first part execution budget is C_i^1. After executing for C_i^1, τ_i should stop the execution on P_1 and migrate to P_2 to execute the reminded execution. Therefore, the HRtimer on P_1 needs to keep track of the time when the τ_i^1's execution budget C_i^1 is expired, to invoke the scheduler to do the task migration. Different from the task release events, the budget expiration events are dynamic. This is because in the preemptive scheduling a running task may be preempted by other tasks, and resume execution at some future time point. So it is inadequate to only set the budget expiration events when a task starts execution.

RTPS manages budget expiration events as follows: At any time, at most one budget expiration event is recorded, no matter how many tasks on one core need the budget control. Each task records its *remained* budget in the $task\_struct$ data structure. Each time when a task is switched to the CPU for execution, the budget expiration event is updated to the time point when this task's remained budget will be expired if it executes without being preempted; each time a task is taken off from the CPU, i.e., preempted by other tasks, its corresponding budget expiration event is canceled and its remained budget is subtracted by the time length it just has executed for. Another benefit of this approach is to avoid inter-core contention on HRtimer caused by task migrations, which will be introduced in detail in next subsection.

B. Data Structures: as Local as Possible

Besides the HRtimer resource, RTPS maintains two queues on each core:

- *Sleep Queue* hosts the tasks that have finished execution in the current period and are waiting to be released for next period.
- *Ready Queue* hosts the tasks that have been released but not finished its execution, and are waiting to run on that core.

In strictly partitioned scheduling (where task splitting is not allowed), these per-core queues are all local data structures, and the operations are quite straightforward: newly released tasks are moved from the sleep queue to the ready queue; finished tasks are moved from the Ready Queue to the

Sleep Queue; in case of preemption, the current running task is put back to the Ready Queue, and the preempting task is moved out of the Ready Queue for execution.

However, in semi-partitioned scheduling, a splitting task will first execute on one core for a certain amount of time, then migrate to another core. Since Linux runs an instance of the scheduler on each core, these queues might be accessed by the scheduler instances on different cores and thereby are not local any more. In RTPS, we use spin locks to guarantee the mutually exclusive access. Spin lock guarantees that at any time only one of the lock requestors obtain the lock, and the other requestors spin around by repeatedly executing a tight loop. Spin lock is simple, but unfair: there is no guarantee that a requestor can obtain the lock as long as there are still other requestors. Therefore, for each requestor, a safe estimation of the waiting time for a lock should count the delay caused by all other requestors to this lock. So a key of reducing the scheduler overhead is to share as few data structures as possible.

There are in total three one-per-core data structures used by the scheduler: Sleep Queue, Ready Queue and HRtimer. All of them are potentially accessed by multiple scheduler instances on different cores:

- Sleep Queue: After the tail subtask of τ_i is finished, τ_i needs to be somehow put back to the Sleep Queue of core P_1 which hosts τ_i's head subtask, such that the next released instance of τ_i starts execution on P_1.
- Ready Queue: When τ_i has expired the execution budget on one core, it needs to be inserted to the Ready Queue of the core hosting the next part of its execution.
- HRtimer: After a task migrates from one core to another, the HRtimer of the destination core needs to be updated at some point such that it can correctly track the time when the budget of this task on the destination core is expired.

However, we manage to implement RTPS in the way that inter-core contentions only happen on the Ready Queue, and both Sleep Queue and HRtimer are local data structures.

Sleep Queue is "localized" by the following approach: When the tail subtask of τ_i finishes, τ_i is inserted into the Sleep Queue of the tail core (the core hosting the last subtask) instead of putting it back to the head core (the core hosting the first subtask). So the HRtimer on the tail core keeps track of the time when this task should be released again. When this task's release time arrives, the HRtimer on the tail core invokes an interrupt, in which τ_i is directly inserted to the head core's ready queue, bypassing the head core's Sleep Queue. Then the scheduler on the tail core sends an inter-core interrupt to the head core, to invoke the scheduling on the head core.

Now we will show that HRtimer is also "localized", by the above approach and the budget expiration management mechanism introduced in Section IV-A. Recall that there are two types of scheduling events, task release and budge expiration, with which HRtimer needs to be updated. We will show in our implementation there is no inter-core operation on HRtimer with either of them:

- Since the task release event of a split task only needs to be recorded on the tail core, there is no inter-core operation on HRtimer due to task releases.
- The budget expiration event is inserted into HRtimer only when a task is switched on the CPU to execute, which is after this task being moved to current core. So updating HRtimer is a initiated by the local scheduler, but not an inter-core operation.

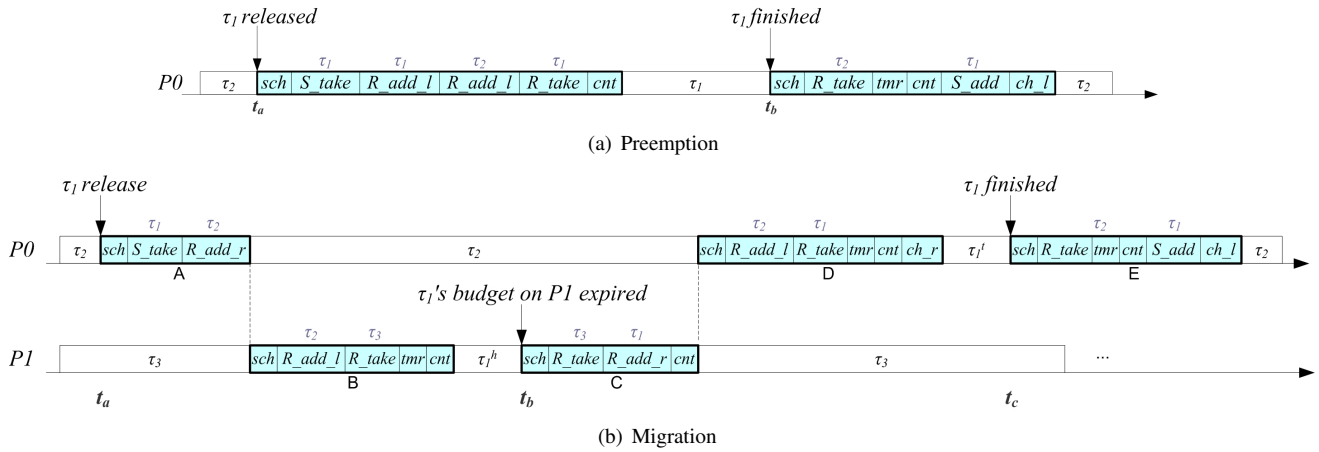
In summary, there is no inter-core operations on Sleep Queue and HRtimer, so they can be used as local data structures without protected by spin locks. Only the Ready Queues are shared data structures, and all the related operations need to be wrapped up by spin locks.

V. Overhead Identification and Accounting

In this section, we will study the run-time overhead of RTPS. We first identify all possible run-time overhead caused by task preemption and migration, and then introduce how these overheads should be accounted into the partitioning algorithm.

The possible run-time overheads are listed as follows.

- Scheduler Invocation (sch). Overhead to invoke the scheduling, which includes the time of responding and serving the interrupt, switching between the user mode and kernel mode.
- Context Switch (cnt). Overhead due to the context switch from the preempted task to the preempting task. It involves storing the preempted task's context, and loading the preempting task's context.
- HRtimer operations (tmr). Overhead for scheduling events to be inserted into or removed from HRtimer's event tree.
- Queue operations overheads due to the operation on the Sleep and Ready Queue. Sleep Queue on each core is a local data structure, and all its related operations are local. Ready Queue on each core is a global shared data structure, and its "adding" operations can be either local or remote. There are different types of queue operation overheads:
  - S_add Overhead of inserting a task into the Sleep Queue.
  - S_take Overhead of searching a task and removing it from the Sleep Queue.
  - R_add_r Overhead of inserting a task into the Ready Queue on another core (the last letter "r" stands for "remote").
  - R_add_l Overhead of inserting a task into the Ready Queue on the local core (the last letter "l" stands for "local").
  - R_take Overhead of searching a task and removing it from the Ready Queue.



(a) Preemption

(b) Migration

Fig. 2. An example to illustrate the run-time overhead.

- Cache Related Overhead due to the task preemption and migration:
  - Preemption Cache Overhead (ch_l). When a task, which is preempted by higher priority tasks at some earlier time point, resumes execution on the same core, it needs to first reload its memory context that have been replace out of the cache (the last letter "l" stands for "local").
  - Migration Cache Overhead (ch_r). When a task migrates to from the first core to the second core, its memory context needs to be moved from the private cache of the first core to the second core (the last letter "r" stands for "remote").

We consider the typical scenarios of task preemption and migration in Figure 2, to demonstrate how the overheads introduced above occur at runtime.

Figure 2-(a) shows the scenario that τ_2 is preempted by a higher-priority task τ_1, and later resumes execution when τ_1 is finished. At time t_a τ_1 is released and the scheduler is invoked by interrupt (sch). The scheduler takes τ_1 out of the Sleep Queue (S_take) and inserts it into the Ready Queue (R_add_l). Since τ_1's priority is higher than τ_2, τ_1 will execute instead of τ_2, so the scheduler puts τ_2 back to the Ready Queue (R_add_l) and takes τ_1 out of the Ready Queue (R_take). After the context switch cnt, τ_1 starts execution. At time t_b τ_1 is finished. It first invokes the scheduler (sch). The scheduler takes the next-to-run task τ_2 out of the Ready Queue (R_take), switches τ_2's context onto CPU (cnt) and updates HRtimer (tmr), to insert the budget expiration event of τ_2 (this is not needed if τ_2 is not split task). Then the scheduler puts the finished task τ_1 back to the Sleep Queue (S_add). Finally τ_2 load its memory content to cache (ch_l) and starts execution. Note that, the cache related overhead of τ_2 is ch_l (l means local) as it resumes execution on the same core. We can sum up the overhead in the above illustration as a safe margin added to a non-split task τ_i's original WCET C_i, to get a safe bound for its overhead-aware execution time $\overline{C_i}$:

$$\begin{aligned}\overline{C_i} \;=\; & C_i + sch + S\_take + S\_add + R\_add\_l + \\ & 2 \times R\_take + tmr + 2 \times cnt + ch\_l\end{aligned}$$

Note that we assume ch_l is a common upper bound for the local cache related overhead for all tasks. So the ch_l before τ_2 resume execution can be accounted into τ_1's execution time, i.e, we always let the task who causes a preemption to be responsible to the cache related overhead of this preemption.

Figure 2-(b) shows a typical scenario of task migration. τ_1 has the highest priority among, and is split into two parts: the first part is assigned to P_1 and the second part to P_0. At time t_a, τ_1 is released, and the scheduler on P_0 is invoked by interrupt (sch). Recall that as introduced in Section IV-B, when a split task is finished, it will be inserted into the Sleep Queue of its tail core, which is P_0 in this example. The scheduler takes τ_1 out of the Sleep Queue on P_0 (S_take), and adds it to the Ready Queue of P_1, which is a remote queue operation (R_add_r), then the scheduler on P_0 issues an inter-core interrupt to invoke the scheduler on P_1, after which τ_2 continues execution. On P_1, the scheduler is invoked by the interrupt from P_0 (sch). Since τ_1 has the highest priority, the scheduler puts the current running task τ_3 back to the Ready Queue (R_add_l), takes τ_1 out of the Ready Queue (R_take), sets HRtimer to keep track the time when τ_1's budget on P_1 expires, and switches the context of τ_1 to CPU to let it execute (cnt).

At time t_b τ_1's budget on P_0 expires, so it invokes the scheduler (sch), which takes τ_3 out of the P_0's Ready Queue, inserts the τ_1 to P_1's Ready Queue which is a remote operation and issues an inter-core interrupt to invoke the scheduler on P_0. Then τ_3 resumes execution after the context switch (cnt). On P_0, τ_1 will execute instead of τ_2 since τ_1 has higher priority. The scheduler inserts the current running task τ_2 back to the Ready Queue, takes τ_1 out of the Ready Queue, set HRtimer and

so on. Since τ_1 migrates from P_1 to P_0, its cache content is remotely loaded (ch_r). At t_c, τ_1 finishes execution and τ_2 resumes, the overhead of which is similar to corresponding part in Figure 2-(a).

Now we discuss how the overheads are accounted for split tasks. We group the overheads into several parts as in Figure 2-(b):

$$\begin{aligned} A &= sch + S\_take + R\_add\_r \\ B &= sch + R\_add\_l + R\_take + tmr + cnt \\ C &= sch + R\_take + R\_add\_r + cnt \\ D &= sch + R\_add\_l + R\_take + tmr + cnt + ch\_r \\ E &= sch + R\_take + tmr + cnt + S\_add + ch\_l \end{aligned}$$

As suggested in Figure 2-(b), for the head subtask τ_i^h (the first part of τ_i), the overhead includes B and C, i.e.,

$$\overline{C_i^h} = C_i^h + B + C$$

and for the tail subtask τ_i^t (the last part of τ_i), the overhead includes A, D and E, i.e.,

$$\overline{C_i^t} = C_i^t + A + D + E$$

For a body subtask $\tau_i^{b_j}$, the overhead involves, first, the "receiving" part from its precedent subtask (D), and second, the "sending" part to its successive subtask (C), so we have:

$$\overline{C_i^{b_j}} = C_i^{b_j} + C + D$$

Finally, one can replace C_i by $\overline{C_i}$ for non-split tasks, and replace C_i^h, C_i^t and $C_i^{b_j}$ by $\overline{C_i^h}$, $\overline{C_i^t}$ and $\overline{C_i^{b_j}}$ respectively for split subtask tasks in the algorithm FP-TS such that the tasks assigned to each processor are guaranteed to be schedulable by RMS in the presence of run-time overheads.

Note that the overhead accounting we presented above is a general approach which works for any job-level fixed priority scheduling algorithm. For a restricted subset of scheduling algorithms it's possible to have a more precise overhead accounting, for example, for task-level fixed priority scheduling, one can easily integrate the overheads into the RTA calculation instead of adding a margin to the WCET, to obtain a tighter overhead accounting. However, to provide a general feeling how much does the run-time overhead affect the real-time performance, we will adopt the general overhead accounting presented above. It's obvious that if the overhead effect is ignorable with this general accounting approach, then the same conclusion also holds with other more precise overhead accounting approaches.

VI. Overhead Measurement

The measurement is conducted on a machine with a 4-core Intel Core i7 870 processor. The processor frequency is 2.93GHz. Each core has a 32k 8-way set associative L1 instruction cache, 32k 4-way set associative L1 data cache, and a private 256k 8-way unified L2 cache. All the four cores share one 8M 16-way set associative L3 cache and a 3G DDR3 memory.

TABLE I
Measured scheduler overheads (in CPU cycles).

overhead	avg/max	no MA		MA	
		64	256	64	256
cnt	avg	542	547	1250	1287
	max	2320	2232	5712	**10268**
tmr	avg	775	743	1293	1233
	max	2076	2948	3464	**7224**
sch	avg	652	670	1559	1592
	max	1532	1512	2700	**4556**
$R\_add\_l^*$	avg	474	466	770	846
	max	1404	1500	1916	**2712**
$R\_add\_r^*$	avg	1541	1486	3374	3233
	max	2388	2300	**5064**	4956
R_take^*	avg	420	461	639	648
	max	1672	1696	3812	**7148**
S_add	avg	735	713	1413	1482
	max	1704	1672	3948	**8132**
S_take	avg	262	260	335	351
	max	1004	1020	1732	**2320**

A. Scheduler Overhead

Since the Ready Queue is a shared data structure, the operations R_add_r, R_add_l and R_take are wrapped by spin locks for mutual exclusive access. In the worst-case, the spinlock acquisition would take the time of performing all other competitor's operations on this queue. So the overhead of operations on the Ready Queue depends on the maximal number of parallel operations. If a core hosts n split tasks, then the maximal number of competitors is n, i.e., the worst-case of a queue operation will task n times longer than an operation without competitors. In our measurement, we only measure the time of single Ready Queue operations without contention, denoted by $R\_add\_r^*$, $R\_add\_l^*$ and R_take^*, then multiply the number of split tasks hosted by corresponding core (known in the task partitioning procedure) to them, in order to obtain R_add_r, R_add_l and R_take.

Table I shows the measurement results of the scheduler overheads in four different settings: with "no MA" tasks do not issue any memory access but only run an empty while loop, while with "MA" each task iteratively visit a 1MB array; with "64" the total number of tasks in the system is 64 (16 tasks on each core), while with "256" the total number of tasks is 256 (64 tasks on each core). Both the maximal ("max") and average ("avg") measured value are provide in the table. We have the following observations:

- The scheduler overhead is insensitive to the number of tasks on each core. As shown in the table, increasing the number of tasks on each core from 16 to 64 essentially does not cause overhead increment.
- The scheduler overhead is sensitive to the tasks' memory usage. In the experiments with "no MA", the tested tasks only consume very few cache capacity, and the scheduler data structures can be completely accommodated by the 256KB L2 cache. In the experiments with "MA", the overhead is on average two

to three times as much as the "no MA" case, which coincides with the ratio between the delay of loading data from L2 cache and L3 cache. Therefore, we can infer that the scheduler data structures still reside in the L3 cache, although the total memory footprint of the tested task set is much larger than the size of the L3 cache (8MB). This is because, the scheduler data structures are the most frequently visited data, so the 16-way set associative L3 cache has a good chance to keep them not be replaced to the memory.

- The remote adding operation on the shared Ready Queue R_add_r is more expensive than the local version R_add_l. The difference is mainly caused by cache coherence protocol. Recall that in our implementation of RTPS, the date structures are design to be as local as possible, and the only shared data structure is the Ready Queue, and the only remote queue operation is R_add_r.

B. Cache Related Overhead

To measure the cache related overhead due to preemptions, we use the task set of one test task and a competing task. Each competing task iteratively updates a 512KB array. We first let the test task to execute without any interruption (with the highest priority in the highest priority scheduling class RTPS), and record its execution time, as the *net time*. Then we let the test task to run with the competing tasks, where the test task has the highest priority. After running the test task for a while (to make sure its working set has been loaded into the cache), the test task suspends itself for 1 second, during which the competing task executes, which will evict the working set of the test task out of the L2 cache. We record test task's execution time in this case, as the *gross time*. The overhead equals the gross time minus the net time.

To measure the cache related overhead due to task migrations, the net time is also obtained by running it without interruptions. The gross time is obtained by measuring the case that a migration is invoked during the task's execution, after which it immediately continues to run on the destination core.

Table II is the measurement results of the preemption and migration overheads with different working set sizes. We can observe that

- Both the preemption and migration cache related overhead roughly increase linearly with respect to the size of the test task's working set.
- The migration cache related overhead is roughly twice as many as the preemption cache related overhead.

Note that in all the experiments of Table II, the working set of the test task can stay in the L3 cache. For the case of large working set that can not be fit into the L3 cache, the cache related overheads caused by preemption and migration are similar, since in both cases a resumed task has to load its memory content from the off-chip memory.

TABLE II
CACHE RELATED OVERHEAD (IN CPU CYCLES).

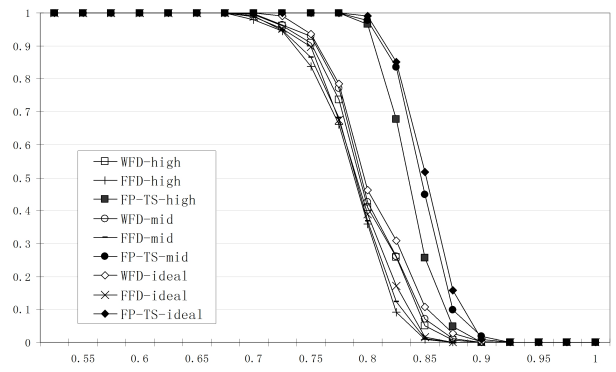
WS (KB)	Preemption	Migration
1	241	305
2	424	610
4	991	1213
8	1497	4176
16	4145	8346
32	6645	14594
64	8318	20592
128	31686	43519
256	63366	82572
512	105852	186137
1024	166534	391402

VII. REAL-TIME PERFORMANCE COMPARISON

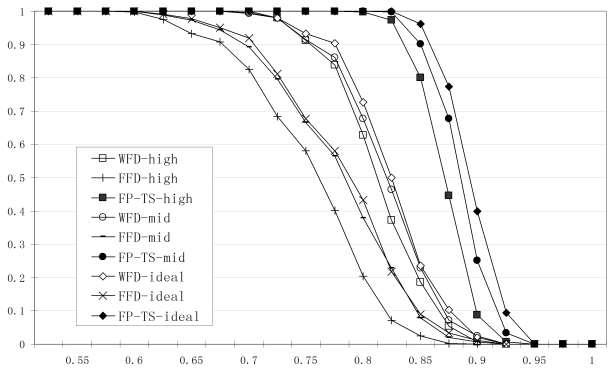
We conduct comparison of the real-time performance in terms of acceptance ratio of FP-TS and two widely-used fixed-priority partitioned scheduling algorithm FFD (first-fit decreasing size) and WFD (worst-fit decreasing size), with randomly generated task sets. We follow the method in [17] to generate synthetic task sets: A task set of M + 1 tasks is generated and tested for feasibility using all the algorithms mentioned above. Then the number of tasks is increased by 1 to generate a new task set, and it is tested for feasibility again. This process is iterated until the total processor utilization exceeds M. The whole procedure is then repeated, starting with a new task set of M + 1 tasks, until 100,000 task sets have been generated and tested.

The hardware setting is based on the machine we used for overhead measurement: There are 4 cores in the system, and $1us$ equals 2930 CPU cycles. We apply the overhead accounting approach presented in Section V and the measured value in Section V to the worst-case execution time of each task, to get the overhead-aware execution time bound, and fed them into the partitioning algorithms (FP-TS, FFD and WFD). For the scheduler overhead, we use the maximal measured value of each type (the bold values in Table I). For the cache related overhead, we let the preemption overhead (ch_l) to be $50us$, and the migration overhead (ch_r) to be $200us$ (both larger than the maximal value in Table II).

Figure 3-(a) shows the first group of experiments in which the individual task utilization is uniformly distributed in $[0.1, 0.3]$. For each partitioning algorithm, we test for three different scales of task frequency (distinguished by the suffix "high", "mid" and "ideal"). The task periods are uniformly distributed in $[10ms, 100ms]$ in the "high" frequency setting, and $[20ms, 200ms]$ in the "mid" uniformly. In the "ideal" frequency setting, we let all the run-time overhead to be 0, to represent the case that the task period/WCET is very large and the overhead has little effect on the timing parameters. The x-axis of the figure is the system utilization (we only keep the part with system utilization above 0.5), and the y-axis is the acceptance ratio. From the figure we can see that, although



(a) U: $0.1 \sim 0.3$



(b) U: $0.1 \sim 0.5$

Fig. 3. An example to illustrate the run-time overhead.

the acceptance ratio of semi-partitioned scheduling with overheads is lower than the ideal case (with zero overhead), it is still clearly higher than the partitioned algorithms (even higher than their ideal case).

Figure 3-(b) has the same setting as Figure 3-(a), except that the individual task utilization is uniformly distributed in $[0.1, 0.5]$. In this figure we can see that, the performance of partitioned scheduling algorithms degrades seriously as the average individual utilization increases, while the performance of semi-partitioned scheduling is not decreased. So the superiority of semi-partitioned scheduling is even larger with "heavy" task sets.

VIII. Conclusion

In this work, we present the implementation of a Linux based semi-partitioned scheduler RTPS. RTPS has the features of both high time resolution and low/predictable run-time overhead. Then we conduct experiments to measure the realistic run-time overheads of RTPS, as well as the cache related overheads. We integrate these overhead measurement into the state-of-the-art semi-partitioned and partitioned scheduling algorithms, and conduct empirical experiments to compare their real-time performance. Our experiments show that, for task sets with reasonable parameters settings, semi-partitioned scheduling indeed outperforms partitioned scheduling in the presence of realistic overheads. So we believe semi-partitioned scheduling should be considered as one of the major design choices for real-time systems on multi-cores. In the future, we will extend our work to support resource sharing in the semi-partitioned scheduler. Another potential direction is to use the mechanism in our semi-partitioned scheduler to support visualization, for a flexible and efficient application-level resource allocation.

References

[1] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*, 2004.

[2] B. Andersson, K. Bletsas, and S. Baruah, "Scheduling arbitrary-deadline sporadic task systems multiprocessors," in *RTSS*, 2008.

[3] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *RTCSA*, 2006.

[4] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-priority multiprocessor scheduling with Liu & Layland's utilization bound," in *RTAS*, 2010.

[5] S. Kato and N. Yamasaki, "Portioned EDF-based scheduling on multiprocessors," in *EMSOFT*, 2008.

[6] ——, "Semi-partitioned fixed-priority scheduling on multiprocessors," in *RTAS*, 2009.

[7] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *ECRTS*, 2009.

[8] RTAI home page. http://www.rtai.org.

[9] Xenomai home page. http://www.Xenomai.org.

[10] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of VxWorks, Linux, RTAI and Xenomai in a hard real-time application," in *Real-Time Conference, 2007 15th IEEE-NPSS*. IEEE, 2007, pp. 1–5.

[11] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *Proceeding of the Real-Time Linux Workshop*, 2009.

[12] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUSˆ RT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *27th IEEE International Real-Time Systems Symposium, 2006. RTSS'06*, 2006, pp. 111–126.

[13] B. Brandenburg, J. Calandrino, and J. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *2008 Real-Time Systems Symposium*, 2008, pp. 157–169.

[14] A. Bastoni, B. Brandenburg, and J. Anderson, "An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor Real-Time Schedulers," in *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010, pp. 14–24.

[15] B. Brandenburg and J. Anderson, "On the Implementation of Global Real-Time Schedulers," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 214–224.

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

[17] T. Baker, "A comparison of global and partitioned EDF schedulability tests for multiprocessors," in *Proceedings of International Conference on Real-Time and Network Systems In Proceedings of International Conference on Real-Time and Network Systems*, 2006, pp. 119–127.