Towards the Implementation and Evaluation of Semi-Partitioned Multi-Core Scheduling

[Work in Progress]

Yi Zhang[1], Nan Guan[2], and Wang Yi[2]

1 **Northeastern University, China**
2 **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, and appear to be a promising solution for scheduling real-time systems on multi-cores. 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, 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 implement a semi-partitioned scheduler in the Linux operating system, and run experiments on a Intel Core-i7 4-cores machine to measure the real overhead in both partitioned scheduling and semi-partitioned scheduling. Then we integrate the obtained overhead into the state-of-the-art partitioned scheduling and semi-partitioned scheduling algorithms, and conduct empirical comparison 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.

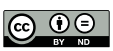
1998 ACM Subject Classification C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

Keywords and phrases real-time operating system, multi-core, semi-partitioned scheduling

Digital Object Identifier 10.4230/OASIcs.PPES.2011.42

1 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 [3]: In the *global* approach, each task can execute on any available processor at run time. In the *partitioned* approach, each tasks is assigned to a processor beforehand and during the run time each task 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



Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 42–46

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the individual task utilization is high, this waste could be significant. In the worst-case only half of the system resource can be utilized in partitioned scheduling.

To overcome this problem, recently researchers proposed *semi-partitioned scheduling* [1, 2, 4, 5, 6, 7], in which most tasks are statically assigned to a corresponding 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 consider the implementation and characterization of semi-partitioned scheduling in realistic systems. We implement a semi-partitioned scheduler in Linux 2.6.32. Then we measure its realistic run-time 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.

2 Implementation of Semi-Partitioned Scheduler

Several semi-partitioned algorithms have been proposed [4]. In this work we adopt a recent developed algorithm FP-TS [4], which is based on Rate-Monotonic Scheduling. FP-TS 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). A detailed description of FP-TS can be found in [4]. Our semi-partitioned scheduler implementation can be easily extended to support a wide range of semi-partitioned algorithms based on both fixed-priority and EDF scheduling.

Now we introduce our semi-partitioned scheduler implementation in Linux 2.6.32. The basic framework of our semi-partitioned scheduler is as follows: Each core has its own Ready queue, which records the tasks have been released but not finished on this core. When a task is released, it will be inserted into the ready queue, and trigger the scheduler. The scheduler decides the task to be executed according to the priority order. The timing parameters of each task are stored in the date structure *task_struct* when the task is created.

There are two types of tasks in the system: (1) *normal tasks*, which execute on a fixed core, and (2) *split tasks*, which will migrate among different cores. The main challenge of the semi-partitioned scheduling is to support splitting tasks to correctly execute on different cores, and to migrate from one core to another core with the timing constraint (obtained from the partitioning algorithm) with as small as possible run-time overhead.

In our implementation, each core maintains its own sleep queue, which records tasks on this core that are currently not active, and its own ready queue, which records tasks on this core that are currently active. The ready queue is implemented by a binomial heap and the

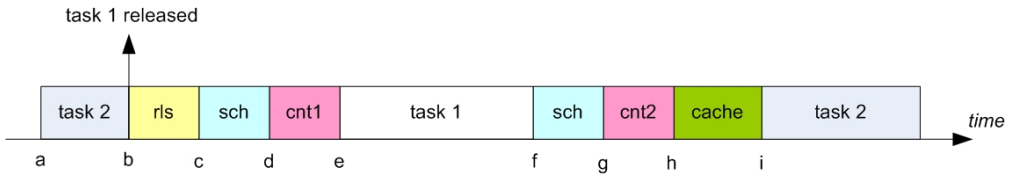


Figure 1 An example to illustrate the run-time overhead.

sleep queue is implemented by a red-black tree. For a split task, we need to control when a subtask on one core will migrate to another. This is done by recording the time budget in the split task's *task_struct* data structure. The main difference between normal tasks and split tasks is in the scheduling action after their budgets on this core are run out. If it is a normal task, the scheduler will put this task to the sleep queue of this core. If it is a split task, the scheduler will: (1) if it is a body subtask, the scheduler will insert the next subtask into the ready queue of the migration destination core, and trigger the scheduling on the destination core; (2) if it is a tail subtask, the scheduler will put this task back to the sleep queue of the core hosting the first subtask of this split task.

3 Overhead Measurement

We use the example in Figure 1 to illustrate the overhead that may happen at runtime. We assume at time a a lower-priority task τ_2 is executing, and at time b, a higher-priority task τ_1 is released. The time between b and e is the overhead due to the release of τ_1 and context switch from τ_2 to τ_1. Task τ_1 finishes its execution at time f, and the time between f and i is the overhead due to the context switch from τ_1 and τ_2. From time i, τ_2 continue to execute the unfinished work. Now we introduce different parts of the overhead one by one.

- ***rls***: This is the overhead due to the task release: When a task is released, the function *release*() is invoked to insert this task into the ready queue. ***rls*** includes the delay from requesting the access to getting access to the ready queue, and the time of doing the insert operation on the ready queue.
- ***sch***: This is the overhead due to the scheduling actions, which is in the function *sch*(). It may happen in two cases: (1) Task release. In this case, *sch*() will select the highest-priority task from the ready queue. If there happens a preemption, *sch*() will put the current running back to the ready queue. (2) Task finish. In this case, *sch*() will select the highest-priority task from the ready queue.
- ***cnt1***: This is the overhead due to the context switch from the preempted task to the preempting task, which is in the function *cnt_swth*(). It will store the preempted task's context and load the preempting tasks's context.
- ***cnt2***: This overhead is also in the function *cnt_swth*(). It may happen in three cases: (1) The current task is a normal task, and has finished its work. In this case, *cnt_swth*() will load the context of the task to run (the highest-priority task selected by *sch*()), then insert the finished task into the sleep queue. (2) The current task is a split task, and it has run out of its budget on this core and will migrate to another. In this case, *cnt_swth*() will reload the context of the task to run next, then insert this task to the ready queue of the destination core. (3) The current task is a split task, and it has finished its execution. In this case, *cnt_swth*() will reload the context of the task to run next, then insert this task into the sleep queue of core which hosts the first subtask of this split task.

Operation	local ($N = 4$)	remote ($N = 4$)	local ($N = 64$)	remote ($N = 64$)
sleep queue – add	2.5	2.9	4.3	4.4
sleep queue – delete	**3.3**	N/A	**5.8**	N/A
ready queue – add	1.5	**3.3**	4.4	**4.6**
ready queue – delete	2.7	N/A	4.6	N/A

Table 1 The measured queue operation durations, all in μs

.

- ***cache***: The preempted task's working space would be (partially) replaced out from the cache, and when it resumes execution, it needs to reload its working space.

The table shows the maximal measured duration of a single ready queue operation and sleep queue operation. We set θ and δ to be the worst-case value among them: when $N = 4$, $\delta = 3.3\mu s$ and $\theta = 3.3\mu s$; when $N = 64$, $\delta = 4.6\mu s$ and $\theta = 5.8\mu s$ (N is the maximal number of tasks in the queue, i.e., the number of tasks on this core). Apart from the delay due to the access to the ready and sleep queues, we also measure the pure execution time of the functions $relase()$, $sch()$ and $cnt\_swth()$, they are $3\mu s$, $5\mu s$ and $1.5\mu s$ respectively.

The last overhead we measured is the cache-related overhead. This overhead is highly dependent on the application memory characters. An important issue is the difference between local context switches and task migrations between cores. Our measurement shows that in general the cache-related overhead due to task migrations and local context switches is in the same order of magnitude. This is due to the shared lower-hierarchy caches (L3 cache in our case): in both local context switches and task migrations, most of the working space of the preempted/to-migrate task will be replaced out from the private cache (L1 and L2 cache in our case), and stay in the shared lower-hierarchy caches. Of course, if an application has generally very small working space (much smaller than the size of private cache, which is rather rare in realistic applications), the cache-related delay of local context switches would be significantly smaller than task migrations, since there is a better chance for the working space of the preempted task to stay in the private cache, until it resumes execution.

4 Results and Conclusion

We conduct comparison of the performance in terms of acceptance ratio of FP-TS and two widely used fixed-priority partitioned scheduling algorithm FFD (first-fit decreasing size partitioning) and WFD (worst-fit decreasing size partitioning), with randomly generated task sets, taking into account the measured overheads shown in last section. Our experiments show that semi-partitioned scheduling indeed outperforms partitioned scheduling in the presence of realistic run-time overheads.

References

1 B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems multiprocessors. In *RTSS*, 2008.

2 B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *RTCSA*, 2006.

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

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 S. Kato and N. Yamasaki. 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.