Experimental Evaluation of Global and Partitioned Semi-Fixed-Priority Scheduling Algorithms on Multicore Systems

Hiroyuki Chishiro and Nobuyuki Yamasaki
School of Science for Open and Environment Systems
Keio University, Yokohama, Japan
{chishiro,yamasaki}@ny.ics.keio.ac.jp

Abstract

Nowadays multicore systems have been used in real-time applications such as robots. In robots, imprecise tasks such as image processing tasks are required to detect and avoid objects. However, existing real-time operating systems have evaluated multiprocessor real-time scheduling algorithms in Liu and Layland's model and have not evaluated those in the imprecise computation model. This paper performs experimental evaluations of global and partitioned semi-fixed-priority scheduling algorithms in the extended imprecise computation model on multicore systems. Experimental results show that semi-fixed-priority scheduling has comparable overhead to fixed-priority scheduling. In addition, global semi-fixed-priority scheduling has lower overhead than partitioned semi-fixed-priority scheduling.

1. Introduction

Nowadays multicore systems have larger number of cores in real-time applications such as robots [12, 19, 1, 6]. In order to detect and avoid objects, robots usually require imprecise tasks [16] such as image processing tasks, which consume much CPU time. In addition, robots have many real-time tasks such as motor control, motion control and sensor processing. In order to complete these real-time tasks by their deadlines, both multiprocessor real-time scheduling policies and real-time scheduling algorithms are important.

There are mainly two multiprocessor real-time scheduling policies: partitioned scheduling and global scheduling. Partitioned scheduling assigns all tasks to specific processors. Each processor has its own run queue and scheduler. Global scheduling permits to migrate tasks between processors. Ready tasks are enqueued in a logical global queue and the \( M \) highest priority tasks, where \( M \) is the number of processors, are assigned to processors.

The task assignment algorithm is performed off-line and all tasks are statically assigned to processors. In contrast, global scheduling permits to migrate tasks between processors. Ready tasks are enqueued in a logical global queue and the \( M \) highest priority tasks, where \( M \) is the number of processors, are assigned to processors.

The overhead of multiprocessor scheduler is a main factor to complete real-time tasks by their deadlines. Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (LITMUSR\(_{RT}\)) [5] is a real-time extension of Linux to evaluate the overhead of multiprocessor real-time scheduling. Bastoni et al. claim that partitioned scheduling is more effective approach than global scheduling [3] in LITMUSR\(_{RT}\) on a 24-core Intel system. On the other hand, SCHED\_DEADLINE is an efficient Earliest Deadline First implementation in Linux [10, 11]. Lelli et al. claim that the implementation of global scheduler in SCHED\_DEADLINE is as scalable and efficient as SCHED\_FIFO on a 48-core AMD system [15]. The overhead of global scheduler depends on both hardware platform and software implementation so that the practicality of global scheduler is still an open problem. However, these real-time operating systems have evaluated multiprocessor real-time scheduling algorithms in Liu and Layland’s model [17] and have not evaluated those in imprecise computation [16].

Semi-fixed-priority scheduling [7] supports the extended imprecise computation model [13, 14], which has a second mandatory part as a wind-up part. However, multiprocessor semi-fixed-priority scheduling algorithms have not been implemented on multicore systems. Now we implement multiprocessor semi-fixed-priority scheduling algorithms such as Global Rate Monotonic with Wind-up Part (G-RMWP) [8] and Partitioned Rate Monotonic with Wind-up Part (P-RMWP) [7].

This paper performs experimental evaluations of global and partitioned semi-fixed-priority scheduling algorithms on multicore systems. Experimental results show that semi-fixed-priority scheduling has comparable overhead to fixed-priority scheduling. In addition, global semi-fixed-priority
scheduling has lower overhead than partitioned semi-fixed-priority scheduling.

The contribution of this paper is to implement global and partitioned semi-fixed-priority scheduling algorithms with low overhead. In addition, we evaluate the practicality of semi-fixed-priority scheduling on multicore systems. We believe that our implementation for global and partitioned scheduling algorithms is widely used and refined.

The remainder of this paper is organized as follows: Section 2 describes the system model. Section 3 introduces related work. Section 4 explains semi-fixed-priority scheduling. Section 5 presents the implementation of global and partitioned semi-fixed-priority scheduling algorithms. Section 6 performs experimental evaluations of global and partitioned semi-fixed-priority scheduling algorithms on multicore systems. Finally we offer concluding remarks in Section 7.

2. System Model

Figure 1 shows the extended imprecise computation model [13, 14]. The extended imprecise computation model adds the wind-up part to the imprecise computation model [16]. The imprecise computation model assumes that the processing to terminate or complete the optional part is not required. However, image processing tasks in autonomous mobile robots require the processings to output the results. They must guarantee the schedulability of them so that completing it is not relevant to scheduling the task set successfully. Hence, the system utilization within n tasks can be defined as $U = \sum U_i / M$. All tasks are ordered by increasing their periods and task $\tau_1$ has the shortest period.

An optional deadline [7] is a time when an optional part is terminated and a wind-up part is released. Each wind-up part is ready to be executed after each optional deadline and can be completed if each mandatory part is completed by each optional deadline. If each mandatory part is not completed by its optional deadline, each wind-up part may miss its deadline. Figure 2 shows the optional deadline of each task. Each solid up arrow, solid down arrow and dotted down arrow represent release time, deadline and optional deadline respectively. Task $\tau_1$ completes its mandatory part by $OD_1$ and executes its optional part until $OD_1$. After $OD_1$, then $\tau_1$ executes its wind-up part. In contrast, task $\tau_2$ does not complete its mandatory part by $OD_2$. When $\tau_2$ completes its mandatory part, then $\tau_2$ executes its wind-up part and does not execute its optional part.

3. Related Work

First of all, we discuss the problem of multiprocessor real-time scheduling in current hardware architectures and software implementations. Next we claim the requirement of imprecise computation on multicore systems.

Current representative real-time operating systems supporting multiprocessor real-time scheduling are LITMUSRT [5] and SCHED_DEADLINE [10, 11], which are real-time extensions of Linux. LITMUSRT and SCHED_DEADLINE mainly evaluate multiprocessor real-time scheduling in Liu and Layland’s model [17]. The practicality of multiprocessor real-time scheduling depends on the WCET analysis. The WCET analysis on multicore systems is difficult because of both hardware and software complexities.

In the hardware layer, the major complexity of the WCET analysis is the behavior of cache. The Cache-related Preemption and Migration Delay (CPMD) strongly affects the overhead of multiprocessor real-time scheduling materially. Bastoni et al. propose two approaches to measure CPMD: schedule-sensitive method that can measure scheduler-dependent cache effects and synthetic method that can be used to quickly record a large number of sam-
semi-fixed-priority scheduling

semi-fixed-priority

Figure 3. General scheduling and semi-fixed-priority scheduling

Figure 4. Task queue

4. Semi-Fixed-Priority Scheduling

Semi-fixed-priority scheduling [7] is defined as part-level fixed-priority scheduling in the extended imprecise computation model [13, 14]. That is to say, semi-fixed-priority scheduling fixes the priority of each part in the extended imprecise task and changes the priority of each extended imprecise task only in the two cases: (i) when the extended imprecise task completes its mandatory part and executes its optional part; (ii) when the extended imprecise task terminates or completes its optional part and executes its wind-up part. Figure 3 shows the difference between general scheduling in Liu and Layland’s model and semi-fixed-priority scheduling in the extended imprecise computation model. In general scheduling, when task \( \tau_i \) is released at 0, then the remaining execution time \( R_i(t) \) is set to \( m_i + w_i \) and monotonically decreasing until \( R_i(t) \) becomes 0 at \( m_i \). In semi-fixed-priority scheduling, when task \( \tau_i \) is released at 0, then \( R_i(t) \) is set to \( m_i \) and monotonically decreasing until \( R_i(t) \) becomes 0 at \( m_i \). When \( R_i(t) \) is 0 at \( m_i \), then \( \tau_i \) sleeps until \( OD_i \). When \( \tau_i \) is released at \( OD_i \), then \( R_i(t) \) is set to \( w_i \) and monotonically decreasing until \( R_i(t) \) becomes 0 at \( OD_i + w_i \). If \( \tau_i \) does not complete its mandatory part by \( OD_i \), then \( R_i(t) \) is set to \( w_i \) at the time when \( \tau_i \) completes its mandatory part. In general scheduling and semi-fixed-priority scheduling, \( \tau_i \) completes its wind-up part by \( D_i \).

RMWP [7] is one of semi-fixed-priority scheduling algorithms in the extended imprecise computation model. As shown in Figure 4, RMWP manages three task queues: Real-Time Queue (RTQ), Non-Real-Time Queue (NRTQ) and Sleep Queue (SQ). RTQ holds tasks which are ready to execute their mandatory or wind-up parts in Rate Monotonic (RM) order. One task is not allowed to execute its mandatory and wind-up parts simultaneously. NRTQ holds tasks which are ready to execute their optional parts in RM order. Every task in RTQ has higher priority than that in NRTQ. SQ holds tasks which complete their optional parts by their optional deadlines or their wind-up parts by their deadlines. The calculation of each relative optional deadline in RMWP is shown in [7].

G-RMWP [8] is based on and extends RMWP for global scheduling on multiprocessors. Like RMWP based on RM, G-RMWP is based on G-RM. The calculation of each relative optional deadline in G-RMWP is shown in [8].

P-RMWP [7] assigns tasks to processors by bin-packing heuristics such as first-fit, next-fit and best-fit using Response Time Analysis [2] or the least upper bound [17] for RM. Because one task set is schedulable by RMWP if the task set is schedulable by RM [7]. After assigning tasks to processors, each processor calculates the relative optional deadline of each task.
5. Implementation

In this section, we present how to implement G-RMWP and P-RMWP in RT-Est [9], which is a real-time operating system for semi-fixed-priority scheduling algorithms.

5.1 Scheduler

First of all, we explain how to implement G-RMWP. Figure 5 shows the dual scheduler. A running queue manages running tasks, the number of which is at most that of processors and a ready queue manages ready tasks. Each queue has $N$ double circular linked lists, where $N$ is the number of priorities, as the $O(1)$ scheduler [18] in Linux kernel 2.6. Like Linux, a smaller number indicates higher priority. In addition, the implementation of the dual scheduler can be adapted to global fixed-priority scheduling algorithms such as G-RM in Liu and Layland’s model if $OD_i = 0$ and $o_i = 0$. Compared to Liu and Layland’s model, the extended imprecise computation model has upward compatibility.

Next we explain how to implement P-RMWP. The hybrid $O(1)$ scheduler [9] is an extension of $O(1)$ scheduler for RMWP on uniprocessors. The detail of the hybrid $O(1)$ scheduler is shown in [9]. Each processor has each hybrid $O(1)$ scheduler and schedules ready tasks in RMWP order.

5.2 Imprecise Computation

Considering the extended imprecise computation model as shown in Figure 1, we now introduce three following functions: end_mandatory, end_optional and terminate_optional.

Algorithm 1 shows end_mandatory function for G-RMWP, which is called when each task completes its mandatory part. If the optional deadline of the current task does not expire at the current time, then the current task starts to execute its optional part. The current task is dequeued from the running queue, decreases its priority and changes its part from mandatory part to optional part. Next the scheduler finds the ready task which has the highest priority in the ready queue. If the ready task has higher priority
if current time < current task optional deadline then
  DequeueTask (running queue, current task);
  DecreasePriority (current task);
  ChangePart (OPTIONAL);
  ready task ← Get Highest Priority Task (ready queue);
if ready task has higher priority than current task then
  EnqueueTask (current task, ready queue);
  DequeueTask (ready task, ready queue);
  EnqueueTask (ready task, running queue);
end
else
  // not preempt current task
  EnqueueTask (current task, running queue);
end
end

Algorithm 1: end mandatory function

ChangePart (WINDUP);
SleepUntil (current task optional deadline, current task);
IncreasePriority (current task);

Algorithm 2: end optional function

than the current task, preempt the current task. The current
task is enqueued to the ready queue. After that, the ready
task is dequeued from the ready queue and enqueued to the
running queue. Otherwise the current task is enqueued to
the running queue, again. If the optional deadline of the
current task does not expire at the current time, then the
current task starts to execute its wind-up part. In this case,
the current task changes its part from mandatory part to wind-
up part. That is to say, the current task discards its optional
part.

Algorithm 2 shows end optional function for G-
RMWP, which is called when each task completes its op-
tional part. The current task changes its part from optional
part to wind-up part, sleeps until its optional deadline and
increases its priority.

Algorithm 3 shows terminate optional function
for G-RMWP, which is called when each task terminates
its optional part at its optional deadline. First the sched-
ulator gets the task at the head of the optional deadline
queue which sorts tasks by increasing absolute optional deadline.
Next the scheduler checks if the absolute optional deadline
of the task expires. If this is true, execute the following
operations. If the task is ready to execute or executing its
optional part, then the task is dequeued from the optional
deadline queue. If the task is running, then the task is de-
queued from the running queue. Otherwise the task is de-
queued from the ready queue. After that, the task increases
its priority and changes its part from optional part to wind-
up part. Next the scheduler gets an ID of idle processor. If
there is an idle processor, then the task is enqueued to the
running queue and assigned to the processor. Otherwise the
scheduler gets the running task which has the lowest pri-
ority in the running queue. If the task has higher priority
than that running task, then preempt that running task. The
scheduler gets the processor ID of the running task. Next
the running task is dequeued from the running queue and
enqueued to the ready queue. After that, the task is en-
queued to the ready queue and assigned to the processor.
Otherwise the task is enqueued to the ready queue. Finally
the scheduler gets the next task and checks if the absolute
optional deadline of the next task expires, again.

P-RMWP schedules ready tasks in RMWP order on
each processor. Therefore, P-RMWP does not need
to find the lowest priority task in running tasks in
terminate optional function, unlike G-RMWP. In
addition, enqueued and dequeued operations on each pro-
cessor are performed in each hybrid $O(1)$ scheduler [9].
5.3 APIC

RT-Est supports x86 multiprocessors, which have Advanced Programmable Interrupt Controller (APIC). Each APIC has a local APIC timer which generates a local APIC timer interrupt on each processor. Now we explain x86-specific implementation on multiprocessors.

When booting the system on x86 multiprocessors, the Boot Strap Processor (BSP) is booted. Next the BSP initializes IRQ and sends startup inter-processor interrupts to processors except BSP, called Application Processors (APs). Then all APs are booted. When all processors including BSP and APs are ready to start each scheduler, then each processor generates each local APIC timer interrupt and starts to execute tasks. The IRQ ID of each local APIC timer interrupt on each processor is different because the interrupt handler should be executed simultaneously to reduce the overhead.

6. Experimental Evaluations

The system has Corei5 750 2.66GHz quad core processor and 2GB DDR3SDRAM 1,333MHz. We implement multiprocessor real-time scheduling algorithms in RT-Est real-time operating system [9]. As previously discussed in Section 3, the WCET analysis on multicore systems is too pessimistic or restrictive due to cache. We would like to support various real-time applications without such limitation so that the experimental evaluations do not use cache.

6.1 Experimental Setups

The experimental evaluations use 1,000 task sets in each system utilization and evaluates G-RMWP, P-RMWP, G-RM and P-RM. Each \( U_i \) is selected within \([0.02, 0.03, 0.04, \ldots, 0.1]\) and splits \( U_i \) into two utilizations which are assigned to \( m_i \) and \( w_i \) respectively. In autonomous mobile robots, there are various periodic tasks. Therefore, the period \( T_i \) of each task \( \tau_i \) is selected within \([1\text{ms}, 2\text{ms}, 3\text{ms}, \ldots, 30\text{ms}]\). The CPU utilization of \( o_i \) is within the range of \([0, 0.3]\). The system utilization \( U \) is selected from \([0.15, 0.2, 0.25, \ldots, 0.6]\). We verify that all generated task sets are schedulable if the overall overhead including scheduler and interrupt handler is equal to 0. The execution length of the \( k^{th} \) task set is the hyperperiod \( H_k \). The task assignment algorithm for P-RMWP and P-RM is the next-fit heuristic to even the number of tasks on each core.

6.2 Overhead Measurements

Figure 6 shows the overhead of end_mandatory function. G-RMWP has higher average and maximum overheads than P-RMWP because G-RMWP manages ready tasks in a logical global queue, which usually manages more ready tasks than each ready queue on each core in P-RMWP. Figure 7 shows the overhead of end_optional function. Like Figure 6, the overhead of end_optional function is lower than that of end_mandatory function. Figure 8 shows the overhead of terminate_optional function. Unlike Figures 6 and 7, the average overhead of G-RMWP has as much as that of P-RMWP. In addition, G-RMWP has lower maximum overhead than P-RMWP. In G-RMWP, the overhead of finding the lowest priority task in running tasks is very low.

Figure 9 shows the overhead of scheduler. Interestingly, G-RMWP has lower average and maximum overheads than P-RMWP. G-RM also has lower average and maximum overheads than P-RM. One of the reasons why global scheduling has lower overhead than partitioned scheduling is because there is no cache penalty in these experimental evaluations.

Figure 10 shows the overall overhead. Like Figure 9, the overall overhead is approximately constant regardless of the number of tasks. In addition, the primary overhead is the access latency of Intel 8259 and local
APIC on x86 multiprocessors. That is to say, the dual scheduler is scalable and achieves low overhead. The overheads of end_mandatory, end_optional and terminate_optional functions are very lower than the overall overhead. Therefore, the overhead of multiprocessor semi-fixed-priority scheduling is tiny in the overall overhead.

7. Concluding Remarks

This paper performed experimental evaluations of global and partitioned semi-fixed-priority scheduling algorithms on multicore systems. Experimental results show that semi-fixed-priority scheduling has comparable overhead to fixed-priority scheduling without cache. In addition, G-RMWP has lower overhead than P-RMWP, thanks to the dual scheduler.

In future work, we will compare other multiprocessor real-time scheduling algorithms and queueing policies to global and partitioned semi-fixed-priority scheduling algorithms. Moreover, we will evaluate the scalability of the dual scheduler on many-core systems.

References


Figure 9. Overhead of scheduler

Figure 10. Overall overhead