RT-Est: Real-Time Operating System for Semi-Fixed-Priority Scheduling Algorithms*

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

This paper presents RT-Est, which is a real-time operating system for semi-fixed-priority scheduling algorithms. RT-Est implements the following mechanisms: (i) the hybrid $O(1)$ scheduler, which is an extension of the $O(1)$ scheduler in Linux kernel 2.6, to achieve semi-fixed-priority scheduling with low overhead; (ii) the high resolution timer, which performs to terminate optional parts at optional deadlines; (iii) SIM, which is an architecture for simulating real-time scheduling. Experimental evaluations show that semi-fixed-priority scheduling is well suited to autonomous mobile robots.

1. Introduction

Real-time systems such as autonomous mobile robots [9, 23, 2, 5] have required the Worst Case Execution Time (WCET) of each task completed by its deadline. The Actual Case Execution Time (ACET) of each task in autonomous mobile robots tends to change, because their behaviors depend on their environments. Current real-time operating systems [4, 26, 20] mainly supports real-time scheduling algorithms in Liu and Layland's model [15]. Compared to Liu and Layland's model, the imprecise computation model [14] is one of the techniques to overcome the gap between theory and practice. The crucial point is that the computation is split into two parts: mandatory part and optional part. A mandatory part affects the correctness of the result and an optional part only affects the quality of the result. By restricting the execution of the optional part to only after the completion of the mandatory part, real-time applications based on the imprecise computation model can provide the

correct output with lower quality, by terminating the optional part. However, the imprecise tasks in autonomous mobile robots require the processings to output the results after terminating or completing their optional parts. For example, object detection tasks by image processing need to output proper instructions to actuators for avoiding objects. When the imprecise tasks terminate or complete their optional parts, the imprecise computation model cannot guarantee to complete them by their deadlines. In order to overcome the weakness of the imprecise computation model, we use the extended imprecise computation model [11, 13] with a second mandatory part, called wind-up part.

RT-Frontier [12] is a real-time operating system supporting the extended imprecise computation model and only implements Earliest Deadline First (EDF)-based dynamic-priority scheduling [15] such as Mandatory-First with Wind-up Part (M-FWP) [11, 13]. M-FWP finds the highest priority task with $O(\log n)$ runtime complexity in heap ready queue, where n is the number of tasks. In contrast, fixed-priority scheduling such as Rate Monotonic (RM) [15] can find the highest priority task with $O(1)$ runtime complexity in $O(1)$ scheduler [18] in Linux kernel 2.6, when the number of priorities is constant. Unfortunately, RM cannot be adapted to the extended imprecise computation model, because one task may miss its deadline due to the overrun of the optional part. In order to adapt fixed-priority scheduling to the extended imprecise computation model, we proposed semi-fixed-priority scheduling [6] and a semi-fixed-priority scheduling algorithm based on RM, called Rate Monotonic with Wind-up Part (RMWP) [6]. Moreover, RMWP schedules the part of each extended imprecise task by fixed-priority so that the runtime complexity finding the highest priority task can also be $O(1)$.

This paper presents RT-Est, which is a real-time operating system for semi-fixed-priority scheduling algorithms. RT-Est implements the following mechanisms: (i) the hybrid $O(1)$ scheduler, which is an extension of the $O(1)$ scheduler [18] in Linux kernel 2.6, to achieve semi-fixed-priority scheduling with low overhead; (ii) the high reso-

*This research was supported in part by CREST, JST. This research was also supported by Grant in Aid for the Global Center of Excellence Program for "Center for Education and Research of Symbiotic, Safe and Secure System Design" from the Ministry of Education, Culture, Sport, and Technology in Japan.

lution timer, which performs to terminate optional parts at optional deadlines; (iii) SIM, which is an architecture for simulating real-time scheduling. Experimental evaluations show that semi-fixed-priority scheduling is well suited to autonomous mobile robots.

The contribution of this paper is to develop a real-time operating system for semi-fixed-priority scheduling algorithms from scratch. In addition, we believe that these mechanisms in our real-time operating system will be widely used.

The remainder of this paper is organized as follows: Section 2 introduces real-time operating systems and shows their problems. Section 3 describes the system model. Section 4 explains semi-fixed-priority scheduling and RMWP. Section 5 presents our real-time operating system, called RT-Est. The effectiveness of RT-Est is evaluated in Section 6. Finally we offer concluding remarks in Section 7.

2. Related Work

Real-time operating systems based on Linux [4, 26, 20] are implemented as kernel patches for specific versions so that reducing maintenance and update costs of these real-time operating systems is difficult.

Danish et al. develop a real-time operating system from scratch, called Quest [8]. They claim that it would be easier to design a new operating system rather than retrofit a system that is not fundamentally designed to be real-time, as they progressed with their Linux developments.

RT-Mach [25] is developed as a real-time version of Mach [22] and supports predictable real-time computing environments. Unfortunately, the development of RT-Mach has been stopped because the porting costs from Mach to RT-Mach and the implementation costs of new device drivers are too much [19].

These real-time operating systems mainly implement real-time scheduling algorithms with Liu and Layland's model [15], which considers the WCET of each task. However, the ACET of each task is important to achieve precise motions in robots, because the relative finishing jitter [3] of each task depends on its ACET.

RT-Frontier [12] is developed from scratch and only implements EDF-based dynamic-priority scheduling [15] such as M-FWP [11, 13] with the extended imprecise computation model. Unfortunately, dynamic-priority scheduling finds the highest priority task with $O(\log n)$ runtime complexity in heap ready queue. In contrast, fixed-priority scheduling such as RM finds the highest priority task with $O(1)$ runtime complexity in $O(1)$ scheduler and cannot be adapted to the extended imprecise computation model. Because one task may miss its deadline due to the overrun of the optional part. Therefore, the implementation of

Figure 1. Extended imprecise computation model

Figure 2. Optional deadline

semi-fixed-priority scheduling [6] is required to achieve autonomous mobile robots.

3. System Model

Figure 1 shows the extended imprecise computation model [11, 13]. The extended imprecise computation model adds the wind-up part to the imprecise computation model [14]. 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 the extended imprecise computation model has the wind-up part.

We assume that the system has one processor and a task set Γ consisted of n tasks with harmonic period relations. Task τ_i is represented as the following tuple $(T_i, D_i, OD_i, m_i, o_i, w_i)$: where T_i is the period, D_i is the relative deadline, OD_i is the relative optional deadline, m_i is the WCET of the mandatory part, o_i is the Required Execution Time (RET) of the optional part and w_i is the WCET of the wind-up part. The RET of each optional part tends to be underestimated or overestimated from time to time because autonomous mobile robots run in uncertain environments. The relative deadline D_i of each task τ_i is equal to its T_i . The j^{th} instance of τ_i is called job $\tau_{i,j}$. The utilization of each periodic task is defined as $U_i = (m_i + w_i)/T_i$. The reason why U_i does not include o_i is because the optional part of τ_i is a non-real-time part so that completing it is not relevant to scheduling the task set successfully. Hence, the utilization of the system within n tasks can be defined as $U = \sum_{i=1}^n U_i$. All tasks are ordered by increasing their periods and τ_1 has the shortest period. We also assume that the WCET of each task includes the overhead of scheduler. That is to say, one task set with harmonic period relations

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

Figure 4. Task queue

scheduled by RM [15] is always feasible on uniprocessors when $U \leq 1$. In addition, the number of priorities is constant. Therefore, we define that the runtime complexity finding the highest priority task in $O(1)$ scheduler [18] is $O(1)$.

An optional deadline [6] 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. Figure 2 shows the optional deadline of each task. Solid up arrow, solid down arrow and dotted down arrow represent release time, deadline and optional deadline respectively. Task τ_1 completes its mandatory part by OD_1 and executes its optional part until OD_1 . After OD_1 , then τ_1 executes its wind-up part. In contrast, task τ_2 does not complete its mandatory part by OD_2 . When τ_2 completes its mandatory part, τ_2 executes its wind-up part and does not execute its optional part.

4. Semi-Fixed-Priority Scheduling

Semi-fixed-priority scheduling [6] is part-level fixed-priority scheduling. 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 with Liu and Layland's model [15] and semi-fixed-priority scheduling with our model. In general scheduling, when τ_i is released at 0, then remaining execution time $R_i(t)$ is set to $m_i + w_i$ and monotonically decreasing until $R_i(t)$ becomes 0 at $m_i + w_i$. In semi-fixed-priority scheduling, when τ_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 τ_i sleeps until OD_i . When τ_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 τ_i does not complete its mandatory part by OD_i , then $R_i(t)$ is set to w_i at the time when τ_i completes its mandatory part. In both schedulings, τ_i completes its wind-up part by D_i .

RMWP [6] is one of semi-fixed-priority scheduling algorithms with 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 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 wind-up parts by their deadlines. The calculation of each relative optional deadline is shown in [6].

5. RT-Est

RT-Est is a real-time operating system for semi-fixed-priority scheduling algorithms. RT-Est has been developed from scratch and implements the following mechanisms: (i) the hybrid $O(1)$ scheduler, which is an extension of the $O(1)$ scheduler [18] in Linux kernel 2.6, to achieve semi-fixed-priority scheduling with low overhead; (ii) the high resolution timer, which performs to terminate optional parts at optional deadlines; (iii) SIM, which is an architecture for simulating real-time scheduling. We first explain the implementation of the extended imprecise computation model and next present these mechanisms.

5.1 Extended Imprecise Computation Model

Figure 5 shows the pseudo code of the extended imprecise computation model. First each task saves its context including general purpose registers and the program counter in `save_context` function. Because each

```

part = save_context();
switch (part) {
case MANDATORY:
    exec_mandatory();
    res = end_mandatory();
case OPTIONAL:
    if (res != DISCARD) {
        exec_optional();
        end_optional();
    }
case WINDUP:
    exec_windup();
}
end_job();

```

Figure 5. Pseudo code of extended imprecise computation model

task resumes its context in the timer interrupt routine for terminating the optional part at the optional deadline. If `save_context` function is called via the timer interrupt routine to terminate the optional part at the optional deadline, `save_context` function returns `WINDUP`. Otherwise `save_context` function returns `MANDATORY`. Next each task executes its mandatory part in `exec_mandatory` function. After completing its mandatory part, each task calls `end_mandatory` function. If the return value of `end_mandatory` function is `DISCARD`, each task discards its optional part and executes its wind-up part in `exec_windup` function. Otherwise each task executes its optional part in `exec_optional` function. If each task completes its optional part, each task calls `end_optional` function and executes its wind-up part in `exec_windup` function. If each task terminates its optional part at its optional deadline, the scheduler resumes its context in the timer interrupt routine and calls `save_context` function. In this case, the return value of `save_context` function is `WINDUP` so that the resumed task executes its wind-up part in `exec_windup` function. After completing the wind-up part, each task calls `end_job` function to complete its job.

5.2 Hybrid $O(1)$ Scheduler

The hybrid $O(1)$ scheduler extends the $O(1)$ scheduler [18] in Linux kernel 2.6 for semi-fixed-priority scheduling algorithms. The hybrid $O(1)$ scheduler manages both general tasks with Liu and Layland's model [15] (by setting o_i to 0) and extended imprecise tasks with our model. The hybrid $O(1)$ scheduler as well as the $O(1)$ scheduler finds the highest priority task with $O(1)$ runtime complexity, when the number of priorities is constant.

Figure 6 shows the hybrid $O(1)$ scheduler with

Figure 6. Hybrid $O(1)$ scheduler

512 ($0x200$) priority level. Because Liu claims that 256 priority levels are sufficient even for the most complex rate-monotonically scheduled systems [16]. In addition, the sum of priority levels including both real-time and non-real-time parts in the extended imprecise tasks is 512 and the highest priority level is 0. The priority range of RTQ is $[0x0, 0xff]$ and that of NRTQ is $[0x100, 0x1ff]$. The hybrid $O(1)$ scheduler manages each task queue by each double circular linked list in FIFO order.

Now we explain the behavior of the hybrid $O(1)$ scheduler. For example, task τ_i , the priority of which is $0x0$, is created and enqueued to SQ. When τ_i is released, τ_i is dequeued from SQ, is ready to execute its mandatory part and is enqueued to RTQ. When τ_i completes its mandatory part, τ_i is dequeued from RTQ, sets its priority to $0x100$ and is enqueued to NRTQ. When τ_i terminates its optional part, τ_i is dequeued from NRTQ, sets its priority to $0x0$, is ready to execute its wind-up part and is enqueued to RTQ. When τ_i completes its optional part or wind-up part, τ_i is dequeued from RTQ or NRTQ and is enqueued to SQ.

The hybrid $O(1)$ scheduler executes queueing operations in RTQ and NRTQ with $O(1)$ runtime complexity as well as the $O(1)$ scheduler, though the priority of each task is changed. The hybrid $O(1)$ scheduler achieves semi-fixed-priority scheduling with $O(1)$ runtime complexity.

Figure 7. Periodic and high resolution timers

```

update_jiffies(next_timer_interrupt) {
  if (next_timer_interrupt < SU2EU(1)) {
    tmp = EU2SU((SU2EU(1) -
      next_timer_interrupt) * read(TCOR));
    stop_timer();
    write(read(TCNT) - tmp - TMU_ADJUST,
      TCNT);
    start_timer();
    exec_jiffies += next_timer_interrupt;
  } else {
    exec_jiffies = SU2EU(1);
  }
}

```

Figure 8. update_jiffies function

5.3 System Time Management

RT-Est supports the high resolution timer, which performs to terminate optional parts at optional deadlines. RT-Est uses `sys_jiffies` and `exec_jiffies` to manage system time. `sys_jiffies` is incremented every periodic timer interrupt and `exec_jiffies` is set to the relative elapsed time to `sys_jiffies`. The high resolution timer can occur per `exec_jiffies`. Also, the unit of `sys_jiffies` is different from that of `exec_jiffies`. Therefore, RT-Est uses `SU2EU` macro which changes the unit of `sys_jiffies` to that of `exec_jiffies` and `EU2SU` macro which changes the unit of `exec_jiffies` to that of `sys_jiffies`. Using `sys_jiffies` and `exec_jiffies`, RT-Est can manage various grained timers including fine-grained and coarse-grained timers.

We implement RT-Est to a SH-4A processor, which uses Time Management Unit (TMU) to manage system time. Now we explain how to implement the high resolution timer using TMU and consider the overhead of setting TMU.

Figure 7 shows the behaviors of both periodic and high resolution timers. The dotted line and the solid line repre-

sent the periodic timer and the high resolution timer of the timer counter register (TCNT) through time respectively. The timer constant register (TCOR) is set to st_1 which is equal to `sys_jiffies`.

First we explain the behavior of the periodic timer in $[0, st_2)$. When the periodic timer is started at 0, the timer interrupt occurs, because the initial value of TCNT is 0, and then TCNT is set to TCOR by hardware. TCNT is monotonically decreasing until TCNT becomes 0. When TCNT is 0 at st_1 , the timer interrupt occurs and then TCNT is set to TCOR by hardware. After that, TCNT is also monotonically decreasing until 0 and the timer interrupt occurs at st_2 .

Next we explain the behavior of the high resolution timer in $[0, st_2)$. When the high resolution timer is started at 0, the timer interrupt occurs and TCNT is set to TCOR by hardware. The timer interrupt routine subtracts the timer counter in $[ht_1, st_1)$ from TCNT. Thanks to the subtraction, the next timer interrupt occurs at ht_1 . When time timer interrupt occurs at ht_1 , the timer interrupt routine subtracts the timer counter in $[0, ht_1)$ from TCNT and the next timer interrupt occurs at st_1 . The following behavior of the high resolution timer in $[st_1, st_2)$ is the same as that of the periodic timer.

Figure 8 shows `update_jiffies` function, which is called in the timer interrupt routine. Reading data from `addr` is represented as `data=read(addr)` and writing data to `addr` is represented as `write(data, addr)`. `next_timer_interrupt` is set to the timer counter until the next timer interrupt occurs. If `next_timer_interrupt` is less than `SU2EU(1)`, the timer counter of the next timer interrupt is less than that of `sys_jiffies`. Otherwise the timer counter of the next timer interrupt is equal to that of `sys_jiffies`. In order to subtract the timer counter from TCNT, the timer must be stopped in a SH-4A processor. If the scheduler stops the timer, changes the value of TCNT and restarts the timer, the system time is different from the actual time. Therefore, we consider the overhead of setting the value of TCNT by subtracting `TMU_ADJUST` from TCNT. `TMU_ADJUST` is the overhead of setting the value of TCNT and is measured before experimental evaluations. Thanks to `TMU_ADJUST`, we consider the overhead of setting the value of TCNT.

5.4 SIM Architecture

SIM is an architecture for simulating real-time scheduling in RT-Est. Figure 9 shows the configuration of RT-Est. RT-Est has an architecture module for selecting architectures and an algorithm module for selecting scheduling algorithms. Unlike User-Mode Linux [1], a port of a full Linux kernel to run in user space, SIM does not measure the actual CPU time consumed by tasks. Like Real-Time system Simulator (RTSIM) [21], SIM executes simulations of

Figure 9. Configuration of RT-Est

Table 1. Specification of LEPRACAUN

CPU	SH-7785(R8A77850BA) 600 MHz
SDRAM	DDR2 SDRAM 256MB
I-Cache	32KB 4-way set associative
D-Cache	32KB 4-way set associative
Size	width 92mm × depth 55mm × height 20.5mm

real-time scheduling algorithms theoretically. In addition, SIM can make use of source codes of architecture independent parts to those of other architectures, unlike RTSIM. Therefore, SIM is an effective technique for not only measuring the performance of real-time scheduling algorithms but also improving the efficiency of development.

6. Experimental Evaluations

6.1 Experimental Setups

We implement RT-Est on LEPRACAUN [7], which has SH-4A architecture and multiple I/Os to achieve autonomous mobile robots. The detail specification of LEPRACAUN is shown in Table 1. The size of LEPRACAUN is approximately as same as that of a card so that we can make use of LEPRACAUN for thin parts such as arms and legs in autonomous mobile robots.

The evaluation uses 1,000 task sets in each CPU utilization and compares RMWP with both M-FWP and RM. The period T_i of each task τ_i is selected within $[1ms, 2ms, 4ms, 8ms, 16ms, 32ms, 64ms, 128ms]$. Each U_i is selected from $[0.02, 0.03, 0.04, \dots, 0.25]$ and splits U_i into two utilizations which are assigned to m_i and w_i respectively. The CPU utilization of $o_{i,j}$ is within the range of $[0, 0.3]$. The CPU utilization U is selected from $[0.3, 0.35, 0.4, \dots, 1.0]$. The execution time of the k^{th} task set is 1,024 times of the hyperperiod H_k . As previously described in Section 3, we assume that the WCET of each task includes the overhead of scheduler and the least upper bounds of all evaluated algorithms with harmonic task sets are $U_{lub} = 1$ [6, 15, 10] so that they are feasible.

We consider the effectiveness of cache to measure both the average and worst case overheads so that we evaluate

the two cases: ID-Cache and No-Cache. ID-Cache enables both I-Cache and D-Cache and, in contrast, No-Cache disables both I-cache and D-Cache. As previous evaluations, we measure the overhead of TMU_ADJUST in Figure 8. The average overheads of TMU_ADJUST in 100 times are $0.92\mu s$ and $1.14\mu s$ in ID-Cache and No-Cache respectively so that we set these values to TMU_ADJUST.

6.2 Overhead Measurements

Figure 10 shows the overhead of `end_mandatory` function in Figure 5. Both average and maximum overheads of RMWP are approximately less than those of M-FWP. Because M-FWP calculates the assignable time of the optional part dynamically with $O(n)$ runtime complexity. In contrast, RMWP dequeues the task from RTQ and enqueues tasks to RTQ or NRTQ, if the optional deadline expires or does not expire, with $O(1)$ runtime complexity.

Figure 11 shows the overhead of `end_optional` function in Figure 5. Unlike Figure 10, both average and maximum overheads of M-FWP are less than those of RMWP. Because RMWP enqueues the task to SQ and M-FWP starts to execute its wind-up part without queueing operations in RTQ and NRTQ. That is to say, when the task completes its optional part in M-FWP, the task executes its wind-up part immediately. Therefore both average and maximum overheads of M-FWP are tiny.

Next we show both average and maximum overheads of each scheduler including both overheads of `end_mandatory` and `end_optional` functions. The difference between average and maximum overheads of each scheduler is wide so that we divide these results into two graphs.

Figure 12 shows the average overhead of each scheduler. In Figure 12(a) and 12(b), the average overheads of RMWP and RM schedulers are approximately constant. The reason why the average overhead of RMWP scheduler is more than that of RM scheduler is because the hybrid $O(1)$ scheduler executes more operations to complete mandatory parts and terminate optional parts than the $O(1)$ scheduler in Figure 6. In contrast, the average overhead of M-FWP scheduler is dramatically more than RMWP and RM schedulers, if the number of tasks is more and more. Figure 13 shows the maximum overhead of each scheduler. The results of the maximum overheads of all schedulers are similar trends to those in Figure 12.

7. Concluding Remarks

This paper presented RT-Est, which is a real-time operating system for semi-fixed-priority scheduling algorithms. RT-Est implements the hybrid $O(1)$ scheduler, which finds the highest priority task with $O(1)$ runtime complexity,

(a) ID-Cache

(b) No-Cache

Figure 10. Overhead of end_mandatory function

when the number of priorities is constant. RT-Est also implements SIM, which can simulate real-time scheduling and make use of implementations of architecture independent parts to those of other architectures. Experimental evaluations show that the overhead of the hybrid $O(1)$ scheduler is approximately constant regardless of the number of tasks. Therefore, RMWP is well suited to autonomous mobile robots.

In future work, we will make use of RT-Est to our autonomous mobile robots [23, 24]. We will execute the actual tasks in autonomous mobile robots, such as image processing tasks [17], to evaluate the effectiveness of RT-Est. In addition, we will also extend the hybrid $O(1)$ scheduler for multiprocessors, because autonomous mobile robots require high performance to manage multiple devices such as motors, sensors and cameras. We believe that the hybrid $O(1)$ scheduler is as scalable as the $O(1)$ scheduler.

References

- [1] <http://user-mode-linux.sourceforge.net/>.
- [2] H. S. Ahn, Y. M. Beak, I.-K. Sa, W. S. Kang, J. H. Na, and J. Y. Choi. Design of Reconfigurable Heterogeneous Modular Architecture for Service Robots. In *IEEE/RSJ 2008 In-*

(a) ID-Cache

(b) No-Cache

Figure 11. Overhead of end_optional function

ternational Conference on Intelligent Robots and Systems, pages 1313–1318, Sept. 2008.

- [3] G. C. Buttazzo. *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*. Springer, 2nd edition, 2004.
- [4] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, Dec. 2006.
- [5] F. R. T. Center. <http://www.furo.org/>.
- [6] H. Chishiro, A. Takeda, K. Funaoka, and N. Yamasaki. Semi-Fixed-Priority Scheduling: New Priority Assignment Policy for Practical Imprecise Computation. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 339–348, Aug. 2010.
- [7] G. R. Corporation. LEPRACAUN. <http://www.generalrobotix.com/product/LEPRACAUN-CPU.htm>.
- [8] M. Danish, Y. Li, and R. West. Virtual-CPU Scheduling in the Quest Operating System. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 169–179, Apr. 2011.
- [9] F. Kanehiro, H. Hirukawa, and S. Kajita. OpenHRP: Open Architecture Humanoid Robotics Platform. *The International Journal of Robotics Research*, 23(2):155–165, 2004.

(a) ID-Cache

(b) No-Cache

Figure 12. Average overhead of scheduler

(a) ID-Cache

(b) No-Cache

Figure 13. Maximum overhead of scheduler

- [10] H. Kobayashi. *REAL-TIME SCHEDULING OF PRACTICAL IMPRECISE TASKS UNDER TRANSIENT AND PERSISTENT OVERLOAD*. PhD thesis, Keio University, Mar. 2006.
- [11] H. Kobayashi and N. Yamasaki. An Integrated Approach for Implementing Imprecise Computations. *IEICE transactions on information and systems*, 86(10):2040–2048, 2003.
- [12] H. Kobayashi and N. Yamasaki. RT-Frontier: A Real-Time Operating System for Practical Imprecise Computation. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 255–264, May 2004.
- [13] H. Kobayashi, N. Yamasaki, and Y. Anzai. Scheduling Imprecise Computations with Wind-up Parts. In *Proceedings of the 18th International Conference on Computers and Their Applications*, pages 232–235, Mar. 2003.
- [14] K. Lin, S. Natarajan, and J. W. S. Liu. Imprecise Results: Utilizing Partial Computations in Real-Time Systems. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 210–217, Dec. 1987.
- [15] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.
- [16] J. W.-S. Liu. *Real-Time Systems*. Prentice Hall, 1st edition, 2000.
- [17] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.
- [18] I. Molnar. Ultra-Scalable $O(1)$ SMP and UP Scheduler. <http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.0/0810.html>, Mar. 2002.
- [19] T. Nakajima and S. Oikawa. Experiences with Building Real-Time Mach Operating Systems. *Computer Software*, 23(1):24–44, 2006 (in Japanese).
- [20] D. of Aerospace Engineering of Politecnico di Milano. Real-Time Application Interface. <http://www.rtai.org>.
- [21] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, G. Bolognini, and P. Ancilotti. An object oriented tool for simulating distributed real-time control systems. *Software - Practice and Experience*, 32(9):907–932, 2002.
- [22] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the 1989 IEEE International Conference*, pages 176–178, 1989.
- [23] T. Taira, N. Kamata, and N. Yamasaki. Design and Implementation of Reconfigurable Modular Robot Architecture. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3566–3571, 2005.
- [24] T. Taira and N. Yamasaki. Functionally Distributed Control Architecture for Autonomous Mobile Robots. *Journal of Robotics and Mechatronics*, 16(2):217–224, 2004.
- [25] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pages 73–82, Oct. 1990.
- [26] V. Yodaiken and M. Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference*, Jan. 1997.