ABSTRACT
Responsive Multithreaded Processor (RMTP) has the Simultaneous Multithreading (SMT) architecture with priority for distributed real-time processings, called prioritized SMT architecture. In RMTP, execution efficiencies of tasks executing in threads except the highest priority thread fluctuate by multiple combinations of tasks executing simultaneously. Therefore, it is difficult to guarantee the schedulability of tasks. Many real-time scheduling algorithms having only real-time part is not well suited to the prioritized SMT architecture. Because they cannot make use of the remaining times of threads except the highest priority thread. In contrast, semi-fixed-priority scheduling has an optional part which is a non-real-time part and improves the quality of the result so that semi-fixed-priority scheduling is well suited to the prioritized SMT architecture. This paper evaluates the performance of semi-fixed-priority scheduling on prioritized SMT processors. Experimental evaluations show that semi-fixed-priority scheduling is well suited to prioritized SMT processors.

KEY WORDS
Scheduling, Operating Systems, Real-Time and Embedded Systems, Semi-Fixed-Priority Scheduling

1 Introduction
Real-time systems such as autonomous mobile robots [2, 6, 14] require high throughput and schedulability. Moreover, autonomous mobile robots run in uncertain environments so that the ratio of Actual Case Execution Time (ACET) to Worst Case Execution Time (WCET) is fluctuated. In order to achieve these real-time systems on uniprocessors is difficult due to lack of CPU resources so that Simultaneous Multithreading (SMT) [17, 16] and Chip Multiprocessing (CMP) [13] architectures are well used. SMT shares hardware resources in hardware threads so that SMT can make use of more hardware resources than CMP. However, SMT is difficult to guarantee the schedulability of hardware threads. Barre et al. analyze the predictability of various policies implemented in SMT processors to control the sharing of resources by concurrent threads [1]. Unfortunately, the confliction of hardware resources occurs in these policies because the hardware threads in SMT processors have the same priorities. In order to overcome the weakness of SMT, we have developed Responsive Multithreaded Processor (RMTP) [19].

RMTP has the SMT architecture with priority for distributed real-time processings, called prioritized SMT architecture. In RMTP, execution efficiencies of tasks executing in threads except the highest priority thread fluctuate by multiple combinations of tasks executing simultaneously. Therefore, it is difficult to guarantee the schedulability of tasks. Many real-time scheduling algorithms with only real-time part with Liu and Layland’s model such as Rate Monotonic (RM) [12] is not well suited to the prioritized SMT architecture. Because they cannot make use of the remaining times of threads except the highest priority thread. In contrast, semi-fixed-priority scheduling such as Rate Monotonic with Wind-up Part (RMWP) [3] has an optional part which is a non-real-time part and improves the quality of the result, called reward, in the extended imprecise computation model [8, 10]. Therefore, semi-fixed-priority scheduling is well suited to the prioritized SMT architecture to improve reward.

This paper evaluates the performance of semi-fixed-priority scheduling on prioritized SMT processors from various aspects. We measure the prioritized SMT execution in RMTP and reward in semi-fixed-priority scheduling. Experimental evaluations show that the prioritized SMT execution executes each thread by prioritized order and semi-fixed-priority scheduling can make use of prioritized SMT execution to improve reward.

The contribution of this paper is an effective combination of prioritized SMT architecture and real-time scheduling. We believe that semi-fixed-priority scheduling is well suited to many real-time systems in prioritized SMT processors.

The remainder of this paper is organized as follows: Section 2 introduces our prioritized SMT processor, called RMTP. Section 3 describes the system model. Section 4 explains semi-fixed-priority scheduling and RMWP. Section 5 shows how to implement RMWP on RMTP. The effectiveness of semi-fixed-priority scheduling on prioritized SMT processors is evaluated in Section 6. Section 7 discusses whether other real-time scheduling algorithms except semi-fixed-priority scheduling can be adapted to prioritized SMT processors. Finally we offer concluding remarks in Section 8.
2 Prioritized SMT Processor

RMTP [19] has 8-way SMT architecture so that RMTP can execute eight hardware threads simultaneously. In this paper, the hardware thread in RMTP is called Logical Processor (LP).

Figure 1 shows the effectiveness of prioritized SMT execution. There are three tasks running in LP_0 – LP_2 and the task with smaller ID has higher priority. Task1 in LP_0 is the highest so that the execution efficiency of task1 is also the highest. When task1 in LP_0 is completed at t_1, the lower priority tasks (task2 and task3) in LP_1-LP_2 increase their execution efficiencies. When task2 in LP_1 is completed at t_2, task3 in LP_2 increases its execution efficiency. In prioritized SMT processors, the highest priority task can get hardware resources every time. The prioritized SMT processor can guarantee the schedulability of the highest priority task as same as that of the uniprocessors. Moreover, tasks except the highest priority task can get hardware resources which higher priority tasks do not use.

3 System Model

Figure 2 shows the extended imprecise computation model [8, 10]. The extended imprecise computation model adds the wind-up part to the imprecise computation model [11]. 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 RMTP and a task set $\Gamma$ consisted of $n$ tasks with harmonic period relations. We discuss why tasks have harmonic period relations. Table 1 shows tasks in our autonomous mobile robot [15, 14], periods of which are harmonic relations. There is strong evidence that task sets are harmonic period relations. Task $\tau_i$ 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 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 $\tau_i$ is equal to its $T_i$. The $j^{th}$ instance of $\tau_i$ 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 $\tau_i$ is a non-real-time part so that completing it is no 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 decreasing priority so that $\tau_1$ has the shortest period. The remaining execution time of $\tau_i$ at time $t$ is represented as $R_i(t)$.

An optional deadline 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 3 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 $\tau_1$ completes its mandatory part by $OD_1$ and executes its optional part until $OD_2$. 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, $\tau_2$ executes its wind-up part and does not execute its optional part.
model [12] and semi-fixed-priority scheduling with the extended imprecise computation model [8, 10]. In general scheduling, when \( \tau_i \) is released at 0, then \( R_i(t) \) 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 \( \tau_i \) 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 both schedulings, task \( \tau_i \) completes its wind-up part by \( D_i \).

RMWP [3] is one of semi-fixed-priority scheduling algorithms with the extended imprecise computation model on unprocessors. As shown in Figure 6, 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 ready 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. Figure 7 shows RMWP algorithm. RMWP executes each scheduling event when the following conditions are met: 1) when \( \tau_i \) becomes ready; 2) when \( \tau_i \) completes its mandatory part; 3) when \( \tau_i \) completes its optional part; 4) when \( OD_i \) expires; 5) when \( \tau_i \) completes its wind-up part; 6) when there are one or multiple tasks in RTQ; 7) when there is no task in RTQ and there are one or multiple tasks in NRTQ. In order to terminate the optional part, each task has the optional deadline. The optional deadline of each task is calculated by Response Time Analysis for Optimal Optional Deadline with Harmonic task sets (RTA-OODH) [3]. The optimal optional deadline \( OD_k \) of task \( \tau_k \) is defined as the time when the assignable time of \( \tau_k \) in \( [OD_k, D_k] \) is equal to \( w_k \) if the ACET of each task is always equal to its WCET. In order to describe RTA-OODH, we show the following theorems in [3].

4 Semi-Fixed-Priority Scheduling

Semi-fixed-priority scheduling [3] is part-level fixed-priority scheduling with the extended imprecise computation model. 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: when the extended imprecise task completes its mandatory part and begins its optional part and when the extended imprecise task terminates or completes its optional part and begins its wind-up part. In addition, semi-fixed-priority scheduling splits one extended imprecise task into two general tasks. The two general tasks have the same periods and same or different release times, cannot be executed simultaneously and are scheduled by fixed-priority in Figure 4. Task \( \tau_i^m \) and \( \tau_i^w \) are the mandatory part and the wind-up part of \( \tau_i \), respectively. In addition, the release times of the first jobs of \( \tau_i^m \) and \( \tau_i^w \) are 0 and \( OD_i \), respectively. When there is no task which is ready to execute its mandatory or wind-up part, each task executes its optional part.

Figure 5 shows the difference between general scheduling such as RM and EDF with Liu and Layland’s...
1. When \( \tau_i \) becomes ready, set \( R_i(t) \) to \( m_i \), dequeue \( \tau_i \) from SQ and enqueue \( \tau_i \) to RTQ. If \( \tau_i \) has the highest priority in RTQ, preempt the current task.

2. When \( \tau_i \) completes its mandatory part:
   (a) If \( OD_i \) expired, set \( R_i(t) \) to \( w_i \).
   (b) Otherwise set \( R_i(t) \) to \( w_i \), dequeue \( \tau_i \) from RTQ and enqueue \( \tau_i \) to NRTQ. If there are one or multiple tasks in RTQ or NRTQ which have higher priority than \( \tau_i \), preempt \( \tau_i \).

3. When \( \tau_i \) completes its optional part, dequeue \( \tau_i \) from NRTQ and enqueue \( \tau_i \) to SQ.

4. When \( OD_i \) expires:
   (a) If \( \tau_i \) is in RTQ and does not complete its mandatory part, do nothing.
   (b) If \( \tau_i \) is in NRTQ, terminate and dequeue \( \tau_i \) from NRTQ, set \( R_i(t) \) to \( w_i \) and enqueue \( \tau_i \) to RTQ. If \( \tau_i \) has the highest priority in RTQ, preempt the current task.
   (c) If \( \tau_i \) is in SQ, dequeue \( \tau_i \) from SQ, set \( R_i(t) \) to \( w_i \) and enqueue \( \tau_i \) to RTQ.

5. When \( \tau_i \) completes its wind-up part, dequeue \( \tau_i \) from RTQ and enqueue \( \tau_i \) to SQ.

6. When there are one or multiple tasks in RTQ, perform RM in RTQ.

7. When there is no task in RTQ and there are one or multiples tasks in NRTQ, perform RM in NRTQ.

Figure 7. RMWP algorithm

**Theorem 1** (Worst Case Interference Time by Higher Priority Tasks). The worst case interference time \( I_k^i \) \( (i < k) \) which is the upper bound time when \( \tau_k \) is interfered by \( \tau_i \) is
\[
I_k^i = \left\lceil \frac{T_k}{T_i} \right\rceil (m_i + w_i).
\]

**Theorem 2** (Assignable Time with Harmonic Task Sets). The assignable time \( A_k \) of task \( \tau_k \) except \( w_k \) is
\[
A_k = D_k - w_k - \sum_{i=0}^{k-1} I_k^i.
\]

**Theorem 3** (Worst Case Interference Time in \([0, OD_k)\)). The worst case interference time \( I_k \) of \( \tau_k \) in \([0, OD_k)\) is
\[
I_k = \sum_{i=0}^{k-1} \left( \left\lceil \frac{OD_k}{T_i} \right\rceil m_i + \left\lceil \frac{OD_k - OD_i}{T_i} \right\rceil w_i \right).
\]

**Theorem 4** (RTA-OODH). The optimal optional deadline \( OD_k \) of task \( \tau_k \) with harmonic task sets is
\[
OD_k = A_k + I_k.
\]

Figure 8 shows an example of schedule by RMWP using RTA-OODH with the following task set \( \Gamma = \{\tau_1 = (5, 5, 4, 1, 0, 1), \tau_2 = (10, 10, 8, 2, 0, 1), \tau_3 = (20, 20, 14, 2, 2, 2)\} \) on uniprocessors. Every \( OD_i \) is calculated by theorem 4. For example, we calculate \( OD_3 \) of \( \tau_3 \) and the process of \( OD_3 \) through time is shown in the bottom part of Figure 8. The optional deadline \( OD_3 \) by theorem 4 is 14. Therefore, job \( \tau_{3,1} \) can execute its optional part in \([7, 8)\) and \([13, 14)\).

Figure 8 is an example of schedule on uniprocessors. In RMTP, tasks except the highest priority task run simultaneously. Therefore, we consider the effective use of semi-fixed-priority scheduling to improve reward on RMTP.

## 5 Implementation

In this section, we explain how to implement RMWP to RT-Est [4], which is a real-time operating system for semi-fixed-priority scheduling, in RMTP.

Here, we explain how to create and execute tasks in RMTP. Figure 9 shows the startup function. When an operating system is boot in RMTP, the task with thread ID 0 (TH0) runs in \( LP_0 \). In startup function, TH0 makes threads (TH1-TH7) in \( LP_1-LP_7 \) by mkth instruction implemented as an inline function in RT-Est. There are two
#define NR_LPS 8
void startup(void) {
  int i;
  for (i = 0; i < NR_LPS - 1; i++) {
    mkth(i + 1, func);
    runth(i + 1);
  }
}

Figure 9. startup function

void task(void *arg) {
  while (1) {
    exec_task();
    stopslf();
  }
}

Figure 10. task function

arguments in mkth function, where the first argument is the thread ID and the second argument is the start program counter of the thread. Next TH0 starts to execute TH1-TH7 in LP1-LP7 respectively by runth inline function, where the first argument is the thread ID.

Next we show how to execute periodic tasks with Liu and Layland’s model [12] before showing how to execute them with the extended imprecise computation model [8, 10].

Figure 10 shows task function, which is a function for periodic tasks with Liu and Layland’s model. Each task runs in task function. When each task is released, the task calls exec_task function, which is a task-specific function. After completing exec_task function, the task calls stopslf, which is an inline function for stopslf instruction to stop its own thread. When this is the release time of each task, TH0 calls runth inline function, where the first argument is the thread ID, in the timer interrupt routine. This sequential operation is the periodic task execution with Liu and Layland’s model in RMTP.

Figure 11 shows extask function, which is a function for periodic tasks with the extended imprecise computation model. As previously shown in Figure 2, there are three parts in the extended imprecise computation model: mandatory part, optional part and wind-up part. Now we explain how to implement the extended imprecise computation model for RMWP.

First, each task saves its contexts including general purpose registers and its program counter in save_context function. save_context function returns MANDATORY unless this function is called by the timer interrupt routine to terminate its optional part at its optional deadline. If this function is called by the timer interrupt routine to terminate its optional part at its optional deadline, this function returns WINDUP.

Next each task executes its mandatory part in exec_mandatory function. When each task completes its mandatory part, then the task calls end_mandatory function, which checks whether there is an assignable time of each optional part. If there is an assignable time of each optional part, the task executes its optional part in exec_optional function. Otherwise, discard its optional part and executes its wind-up part in exec_windup function. If the task completes its mandatory part, call end Optional function and next call exec_windup function. If the task terminates its optional part at its optional deadline, call save_context function in the timer interrupt routine and resume the context of the task, then return save_context function in extask by hooking the return address of the task. In this situation, the return value of save_context function is WINDUP so that the task executes its wind-up part in exec_windup function. After completing its wind-up part, the task calls stopslf function like task function in Figure 10. This sequential operation is the periodic task execution with the extended imprecise computation model in RMTP.

void extask(void *arg) {
  while (1) {
    part = save_context();
    switch (part) {
      case MANDATORY:
        exec_mandatory();
        res = end_mandatory();
        break;
      case OPTIONAL:
        if (res != DISCARD) {
          exec_optional();
          end_optional();
        }
        break;
      case WINDUP:
        exec_windup();
        break;
    }
    stopslf();
  }
}

Figure 11. extask function

6 Experimental Evaluations

In this section, we evaluate the performance of semi-fixed-priority scheduling on RMTP. Table 2 shows the specification of RMTP. The WCET analysis of each task is too pessimistic or under multiple specific conditions such as no use cache [5, 18]. In this evaluation, we do not use cache to measure the more precise WCET of each task. Moreover, we load the binary of RT-Est to SDARM, the read/write cost of which is higher than that of SRAM. We implement RMWP to RT-Est to measure the effectiveness of the prioritized SMT execution and semi-fixed-priority scheduling.
Table 2. Specification of RMTP

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency</td>
<td>31.25MHz</td>
</tr>
<tr>
<td>SDRAM</td>
<td>64MB</td>
</tr>
<tr>
<td>SRAM</td>
<td>256KB</td>
</tr>
<tr>
<td>L-Cache/D-Cache</td>
<td>each 32KB (Harvard)</td>
</tr>
<tr>
<td>Fetch width</td>
<td>8</td>
</tr>
<tr>
<td>Issue width</td>
<td>4</td>
</tr>
<tr>
<td>Integer register</td>
<td>32-bit x 32-entry x 8-set</td>
</tr>
<tr>
<td>Integer renaming</td>
<td>register 32-bit x 64-entry</td>
</tr>
<tr>
<td>FP register</td>
<td>64-bit x 8-entry x 8-set</td>
</tr>
<tr>
<td>FP renaming register</td>
<td>64-bit x 64-entry</td>
</tr>
<tr>
<td>ALU</td>
<td>4 + 1(Divider)</td>
</tr>
<tr>
<td>FPU</td>
<td>2 + 1(Divider)</td>
</tr>
<tr>
<td>64-bit ALU</td>
<td>1</td>
</tr>
<tr>
<td>FP vector unit</td>
<td>1(4FPU x 2line)</td>
</tr>
<tr>
<td>Branch unit</td>
<td>2</td>
</tr>
<tr>
<td>Memory access unit</td>
<td>1</td>
</tr>
</tbody>
</table>

void recursive(int depth)
{
    if (depth <= 0) {
        return;
    }
    recursive(depth - 1);
}

Figure 12. recursive function

There are eight LPs in RMTP so that the thread in LP0 is the dedicated thread for timer interrupt handling. In this evaluation, all periods of tasks are 1 ms which is the shortest period in Table 1. Moreover, examples of 1 ms periodic tasks are multiple control tasks such as motors, servos and actuators, which are important to achieve precise motions in autonomous mobile robots.

There are two operations to evaluate the effectiveness of prioritized SMT processor because these operations are well used in autonomous mobile robots. Figure 12 shows the pseudo code of recursive function. This function is a simple recursive program and makes use of many stacks. Figure 13 shows the pseudo code of inverse function. This function is an implementation of Gaussian elimination method to calculate inverse matrix, where the dimension is 3, assumed 3-D image processing.

6.1 Prioritized SMT Execution

We measure the effectiveness of prioritized SMT execution in RMTP. We execute seven tasks in LP1-LP7 with 100 times and measure both average and maximum finishing times of each task in both recursive and inverse functions.

Figure 14 shows the finishing time of recursive function, where the depth is 10. Both average and maximum finishing times of the task in LP1 are the earliest in LP1-LP7. The priority of each task is lower and lower, both average and maximum finishing times of each task are approximately earlier and earlier. The prioritized SMT execution is effective because the ratio of the finishing time of the task in LP1 to that in LP7 is approximately 2.27.

Figure 15 shows the finishing time of inverse function. The finishing time of each task is similar trend with that in Figure 14. Unfortunately, the effectiveness of prioritized SMT execution drops unlike recursive function because the ratio of the finishing time of the task in LP1 to that in LP7 is approximately 1.06. In order to achieve fine-grained prioritized SMT execution, the IPC control method was presented [20]. In the current version of RMTP, the IPC control method is only register-transfer-level implementation and is not a real-chip implementation. If we evaluate the performance of prioritized SMT execution with the IPC control method, the ratio of finishing time of the task in LP1 to that in LP7 is dramatically more.

#define NR_DIMS 3

void inverse(int dst[NR_DIMS][NR_DIMS],
            int src[NR_DIMS][NR_DIMS])
{
    int buf;
    int i, j, k;
    int n = NR_DIMS;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            inv_matrix[i][j] = (i == j);
        }
    }
    for (i = 0; i < n; i++) {
        buf = 1;
        for (j = 0; j < n; j++) {
            matrix[i][j] *= buf;
            inv_matrix[i][j] *= buf;
        }
        for (j = 0; j < n; j++) {
            if (i != j) {
                buf = matrix[j][i];
                for (k = 0; k < n; k++) {
                    matrix[j][k] *= buf;
                    inv_matrix[j][k] *= buf;
                }
            }
        }
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                matrix[i][j] = inv_matrix[i][j];
            }
        }
    }
}

Figure 13. inverse function
6.2 Reward

We evaluate the reward of optional part in semi-fixed-priority scheduling. In mandatory part, we assume that each task executes the data read operation from devices in exec_mandatory function, the execution time of which is very few. In optional part, each task executes the infinite loop with counting the number of iterations in exec_optional function to measure the reward. In wind-up part, each task executes one of the following processings in exec_windup function: nop, recursive or inverse, where nop is no operation, recursive is recursive function in Figure 12 and inverse is inverse function in Figure 13.

Now we explain how to calculate the optional deadline of each task in RMTP. We make use of the finishing times of both recursive and inverse functions in Figure 14 and 15 respectively. We consider that the WCET of wind-up part \( w_i \) executing recursive and inverse functions are 1.2 times (safety margin) as same as the maximum finishing time of those. However, RTA-OODH in theorem 4 is not well suited to prioritized SMT processor so that we set the optional deadline as follows: \( OD_i = D_i - w_i \). This equation can improve more reward than RTA-OODH in theorem 4.

Figure 16 shows the number of iterations in each LP. The priority of each thread is higher and higher, the number of iterations is larger and larger. The number of iterations in recursive drops slightly because the WCET of recursive is from 26μs to 59μs in Figure 14 so that the CPU utilization of each task with 1ms period is approximately 2.6% to 5.9%. In contrast, the number of iterations in inverse drops dramatically because the WCET of inverse is from 579μs to 616μs in Figure 15 so that the CPU utilization of each task with 1ms period is approximately 57.9% to 61.6%. In RMTP, tasks except the highest priority task can be executed simultaneously unlike uniprocessors so that semi-fixed-priority scheduling can make use of the prioritized SMT execution to improve reward.

7 Related Work

In this section, we compare our work with related work.

Direct Priority Mapping (DPM) [7] and Shorter Period Upper (SPU) [7] are fixed-priority scheduling algorithms on prioritized SMT processors. DPM and SPU require guaranteeing execution efficiencies of tasks in LPs except the highest priority LP. The control of execution efficiency of each task is difficult so that we have developed the IPC control method [20]. However, the current version of RMTP does not implement the IPC control method in real-chip level.

Mandatory-First with Wind-up Part (M-FWP) [8, 10] and Slack Stealer for Optional Parts (SS-OP) [9] are EDF-based dynamic-priority scheduling algorithms with the extended imprecise computation model on uniprocessors. M-FWP and SS-OP calculate the assignable time of each optional part dynamically. However, guaranteeing execution efficiencies of tasks in LPs except the highest priority LP is difficult. In order to guarantee the schedulability of each task, the assignable time of each task in LPs except the highest LP must be set to 0. That is to say, the assignable time of each optional part on prioritized SMT processors is equal to that on uniprocessors. Therefore, M-FWP and SS-OP cannot make use of the prioritized SMT execution.
RMWP [3] is a semi-fixed-priority scheduling algorithm and does not calculate the assignable time of each optional part dynamically, thanks to optional deadline. Unlike M-FWP and SS-OP, RMWP can make use of the prioritized SMT execution.

8 Concluding Remarks

This paper evaluates the performance of semi-fixed-priority scheduling on RMTP. Experimental evaluations show that prioritized SMT execution using the same tasks can achieve earlier finishing time in higher priority tasks. Moreover, semi-fixed-priority scheduling can improve more reward to make use of the prioritized SMT execution so that semi-fixed-priority scheduling is well suited to prioritized SMT processors.

In future work, we will implement the IPC control method in RT-Est to achieve fine-grained prioritized SMT execution. Moreover, we will evaluate the actual tasks in autonomous mobile robots, as shown in Table 1. In addition, we will consider the effectiveness of cache in RMTP, because RMTP has the prioritized cache replacement policy, which is more suitable than LRU replacement policy in real-time systems. We believe that the combination of prioritized cache replacement policy and prioritized SMT execution in RMTP is more effective to improve reward.

Acknowledgement

This research was supported 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.

References


