Regular Paper

Multiprocessor Semi-fixed-priority Scheduling

Hiroyuki Chishiro[1,†1,a)]

Received: May 25, 2017, Accepted: November 7, 2017

Abstract: Optimal multiprocessor real-time scheduling can achieve full system utilization with implicit-deadline periodic task sets. However, worst case execution time (WCET) analysis is difficult on state-of-the-art hardware/software platforms due to the complex hierarchy of shared caches and multiprogramming. The actual case execution time of each task is usually shorter than its WCET and imprecise computation is an effective method to make better use of the remaining processor time. Semi-fixed-priority scheduling is real-time scheduling that supports imprecise computation and multiprocessors but conventional semi-fixed-priority scheduling algorithms are not optimal. This paper proposes an optimal multiprocessor semi-fixed-priority scheduling algorithm that supports imprecise computation. The proposed algorithm, which integrates Reduction to Uniprocessor (RUN) for Rate Monotonic with Wind-up Part (RMWP), called RUN-RMWP, is superior to Partitioned RMWP algorithm in terms of schedulability analysis. Simulation studies show that RUN-RMWP has a few more preemptions/migrations compared to RUN but confirms its optimality even though conventional semi-fixed priority scheduling algorithms are not optimal.

Keywords: optimal multiprocessor real-time scheduling, semi-fixed-priority scheduling, schedulability analysis

1. Introduction

Multiprocessors have been increasingly used in state-of-the-art real-time applications such as humanoid robots [25], [27]. These robots usually perform periodic real-time tasks with harmonic relationships (harmonic periodic task sets) where task periods are integer multiples of each other. Harmonic periodic task sets improve the utilization bound of real-time scheduling [19] and the precision of schedulability test [9] compared with general periodic task sets that do not have a relationship among the task periods.

There are generally two main multiprocessor real-time scheduling categories: partitioned scheduling and global scheduling. Partitioned scheduling assigns tasks to processors offline but guarantees only 50% processor utilization in the worst case [2]. In contrast, global scheduling can achieve 100% processor utilization by migrating tasks among processors online but increases run-time overhead. This paper is interested in global scheduling and especially optimal multiprocessor real-time scheduling that can achieve 100% processor utilization with implicit-deadline periodic task sets (i.e., all relative deadlines of tasks are equal to their periods). Several optimal multiprocessor real-time scheduling algorithms have been proposed [3], [7], [18], [22] and Reduction to Uniprocessor (RUN) [26] outperforms other algorithms with respect to the number of preemptions/migrations, and hence this paper focuses on RUN.

RUN transforms the multiprocessor real-time scheduling problem into an equivalent set of uniprocessor problems using the DUAL and PACK operations (details of these operations are given in Section 4). Earliest Deadline First (EDF) [24] is optimal for implicit-deadline task sets on uniprocessors, and hence RUN uses it to transform uniprocessor scheduling into multiprocessor scheduling online. Using these operations, RUN achieves its optimality with the small number of preemptions/migrations.

Real-time scheduling analyzes the schedulability using the worst case execution time (WCET) of each task. However, WCET analysis is difficult on state-of-the-art hardware/software platforms due to the complex hierarchy of shared caches and multiprogramming. Since humanoid robots run in dynamic environments, the actual case execution time (ACET) of each task fluctuates and is usually shorter than its WCET in these real-time applications. Imprecise computation [23] is an effective method to make better use of the remaining processor time (e.g., WCET - ACET). The imprecise computation model has a mandatory real-time part and an optional non-real-time part. By terminating the optional part, each task avoids missing its deadline. The imprecise computation model does not work well in actual systems because it does not take into account the processing required to terminate or complete the optional part of each task. Therefore, this paper uses an extended imprecise computation model [20] that has a wind-up (second mandatory) part after the optional part.

Only two multiprocessor real-time scheduling algorithms support the extended imprecise computation model: Global Rate Monotonic with Wind-up Part (G-RMWP) [13] and Partitioned Rate Monotonic with Wind-up Part (P-RMWP) [15]. These RMWP-based algorithms are semi-fixed-priority scheduling [12] in the extended imprecise computation model and have an original parameter, called *optional deadline*. An optional deadline is the time when an optional part must be terminated to avoid the deadline miss of a wind-up part. Thanks to the optional deadline,

1 Graduate School of Industrial Technology, Advanced Institute of Industrial Technology, Shinagawa, Tokyo 140–0011, Japan
†1 Presently with Graduate School of Information Science and Technology, The University of Tokyo
a) chishiro@aiit.ac.jp

G-RMWP and P-RMWP are superior to Global Rate Monotonic (G-RM) [6] and Partitioned Rate Monotonic (P-RM) [24] in terms of schedulability analysis, respectively. Unfortunately, G-RMWP and P-RMWP are not optimal because they cannot achieve 100% processor utilization. Compared to optimal multiprocessor real-time scheduling, G-RMWP and P-RMWP degrade schedulability, and hence humanoid robots fall due to missing deadlines of real-time tasks.

This paper proposes Reduction to Uniprocessor for Rate Monotonic with Wind-up Part (RUN-RMWP), which is optimal multiprocessor real-time scheduling based on RUN for imprecise computation with harmonic periodic task sets. RUN-RMWP supports the extended imprecise computation model to improve the Quality of Service (QoS) and achieves optimal full system utilization. In addition, RUN-RMWP manages the *optional deadline timer* that terminates the optional part of each task. RUN-RMWP migrates tasks among processors online and is global scheduling. Simulation studies show that RUN-RMWP has a few more preemptions/migrations compared to RUN but confirms its optimality even though conventional semi-fixed priority scheduling algorithms are not optimal.

This paper uses *optimal* in two meanings: Meaning of Optimal (1) and Meaning of Optimal (2). The Meaning of Optimal (1) is that 100% processor utilization of mandatory and wind-up parts can be achieved without deadline miss. The Meaning of Optimal (2) is explained in detail in Section 3.

The remainder of this paper is organized as follows. Section 2 introduces the extended imprecise computation model. Section 3 explains semi-fixed-priority scheduling in the extended imprecise computation model and RMWP algorithm. Section 4 introduces RUN and gives a scheduling example. Section 5 proposes the RUN-RMWP algorithm to achieve optimal multiprocessor semi-fixed-priority scheduling. Section 6 evaluates the effectiveness of RUN-RMWP through simulation. Section 7 compares this work with related one, and Section 8 concludes this paper.

2. System Model

This section introduces the system model that supports the extended imprecise computation model [20] as well as RUN's specific model [26].

Figure 1 shows the extended imprecise computation model [20], which adds a wind-up part to the imprecise computation model [23]. The imprecise computation model assumes that the processing to terminate or complete the optional part is not required. However, image processing tasks in robots require a mandatory part prior to the task completion in order to output the results. To guarantee the schedulability of these tasks, the imprecise computation model is extended with a wind-up part as a second mandatory part.

This paper assumes that a task set Γ has n periodic independent tasks $\tau_1, \ldots, \tau_n$ on M identical processors $P_1, \ldots, P_M$. The task set is synchronous (i.e., all tasks are initially released at time $t = 0$). Each task τ_i has its WCET C_i, period T_i, and relative deadline D_i. Harmonic periodic task sets where task periods are integer multiples of each other are used. Each task set has an implicit-deadline; that is to say, the relative deadline D_i of task τ_i is equal to its period T_i. The priority of each task is represented as p_i. The utilization of each task is represented as $U_i = C_i/T_i$ and the system utilization is $U = \frac{1}{M}\sum_{i=1}^{n} U_i$. Each instance of a task is called a job.

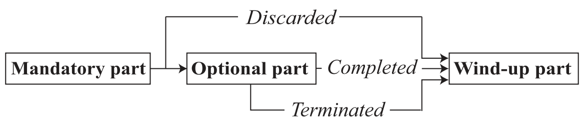


Fig. 1 Extended imprecise computation model.

The extended imprecise computation model adds the wind-up part as a second mandatory part. Therefore, the WCET of each task is $C_i = m_i + w_i$, where m_i is the WCET of the mandatory part and w_i is the WCET of the wind-up part. The Required Execution Time (RET) of the optional part of each task is represented as o_i and its utilization is $U_i^o = o_i/T_i$. $o_{i,j}$ is the time to be actually executed in the j^{th} job of task τ_i. The RET of the optional part of each task fluctuates and its WCET is unknown. The reason why U_i does not include the RET of optional part o_i is because the optional part of each task is a non-real-time part, and hence completing it is not relevant to successfully scheduling the task set.

The relative optional deadline OD_i of task τ_i is defined as the time when an optional part is terminated and a wind-up part is released [12]. Each wind-up part is ready for execution after each optional deadline and can be completed if each mandatory part is completed by the optional deadline. If the mandatory part of each task is not completed by its optional deadline, the corresponding wind-up part may miss its deadline. Note that the corresponding wind-up part may complete its execution by its deadline in such case.

Figure 2 shows the optional deadline of each task. Each solid up-arrow, solid down-arrow, and dotted down-arrow represents the release time, deadline, and optional deadline, respectively. Task τ_1 completes its mandatory part before optional deadline OD_1 and then executes its optional part until OD_1. After OD_1, task τ_1 executes its wind-up part. In contrast, task τ_2 does not complete its mandatory part by optional deadline OD_2. As a result, when τ_2 completes its mandatory part, it executes its wind-up part and does not execute its optional part.

3. Semi-fixed-priority Scheduling

Semi-fixed-priority scheduling [12] is defined as part-level fixed-priority scheduling in the extended imprecise computation model [20]. 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 in just two cases: (1) when the extended imprecise task completes its manda-

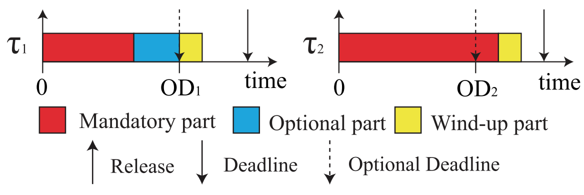


Fig. 2 Optional deadline.

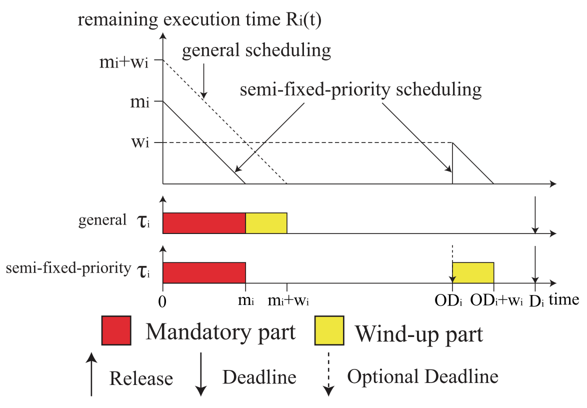


Fig. 3 General scheduling and semi-fixed-priority scheduling.

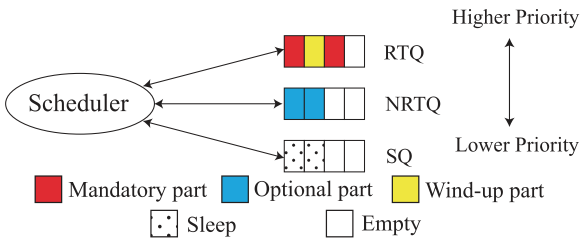


Fig. 4 Task queue.

tory part and executes its optional part, and (2) 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 [24] and semi-fixed-priority scheduling in the extended imprecise computation model. In this case, task τ_i is not interfered with by higher priority tasks. In general scheduling, when task τ_i is released at time 0, then the remaining execution time $R_i(t)$ is set to $m_i + w_i$ and is monotonically decreased until $R_i(t)$ becomes 0 at time $m_i + w_i$. In semi-fixed-priority scheduling, when task τ_i is released at time 0, then $R_i(t)$ is set to m_i and is monotonically decreased until $R_i(t)$ becomes 0 at time m_i. When $R_i(t)$ is 0 at time m_i, then τ_i sleeps until time OD_i. When τ_i is released at time OD_i, then $R_i(t)$ is set to w_i and is monotonically decreased until $R_i(t)$ becomes 0 at time $OD_i + w_i$. If τ_i does not complete its mandatory part by time OD_i, then $R_i(t)$ is set to w_i at the time when τ_i completes its mandatory part. In general scheduling as well as semi-fixed-priority scheduling, τ_i completes its wind-up part by time D_i.

RMWP [12] is a semi-fixed-priority scheduling algorithm that uses the extended imprecise computation model on uniprocessors. As shown in **Fig. 4**, RMWP manages three task queues: Real-Time Queue (RTQ), Non-Real-Time Queue (NRTQ), and Sleep Queue (SQ). RTQ holds tasks that are ready to execute their mandatory or wind-up parts in Rate Monotonic (RM) order [24]. A task is not allowed to execute its mandatory and wind-up parts simultaneously. NRTQ holds tasks that are ready to execute their optional parts in RM order. Every task in RTQ has higher priority than that in NRTQ. SQ holds tasks that have completed their optional parts by their optional deadlines or their wind-up parts by their deadlines. The calculation of each optional deadline in RMWP is shown in Ref. [12]. The relative optimal optional deadline of each task in RMWP is calculated by using Response Time Analysis for Optimal Optional Deadline with Harmonic periodic task sets (RTA-OODH) [12].

This paper uses *optimal* in two meanings.

- Meaning of Optimal (1): Optimal multiprocessor real-time scheduling means that 100% processor utilization of mandatory and wind-up parts can be achieved without deadline miss.
- Meaning of Optimal (2): The relative optimal optional deadline of task τ_i means that $D_i - OD_i$ is equal to the sum of the WCET of its wind-up part w_i and the worst case interference time from higher priority tasks. In any case, this optimality seems directly related to the fact that the later optional deadline is, the longer the optional part can be executed.

This paper explicitly describes *Meaning of Optimal* (*1*) and *Meaning of Optimal* (*2*) when using the first and second meanings, respectively.

First, the following theorems are introduced for the proposed algorithm in this paper.

Theorem 1 (From Theorem 1 in Ref. [12])**.** *The worst case interference time I_k^i ($\forall i : p_i > p_k$), which is the upper bound of the time when τ_i interferes with τ_k in RMWP on uniprocessors, is*

$$I_k^i = \left\lceil \frac{T_k}{T_i} \right\rceil (m_i + w_i).$$

Note that Theorem 1 can be adapted to harmonic periodic task sets as well as general ones. In the case of harmonic periodic task sets, $\left\lceil \frac{T_k}{T_i} \right\rceil = \frac{T_k}{T_i}$.

Theorem 2 (From Theorem 5 in Ref. [12])**.** *The assignable time of task τ_k except w_k in RMWP on uniprocessors with harmonic periodic task sets is*

$$A_k = D_k - w_k - \sum_{\forall i : p_i > p_k} I_k^i.$$

This paper next introduces the worst case interference time of each task.

Theorem 3 (From Theorem 6 in Ref. [12])**.** *The worst case interference time I_k of task τ_k in RMWP on uniprocessors with harmonic periodic task sets is*

$$I_k = \sum_{\forall i : p_i > p_k} \left(\left\lceil \frac{OD_k}{T_i} \right\rceil m_i + \left\lceil \frac{OD_k - OD_i}{T_i} \right\rceil w_i \right).$$

Using these theorems, this paper explains RTA-OODH for calculating the relative optimal optional deadline of each task in RMWP on uniprocessors.

Theorem 4 (From Theorem 7 in Ref. [12])**.** [*Meaning of Optimal* (*2*)] *The relative optimal optional deadline OD_k of task τ_k in RMWP by RTA-OODH on uniprocessors with harmonic periodic task sets is*

$$OD_k = A_k + I_k,$$

where A_k and I_k are in Theorems 2 and 3, respectively.

RTA-OODH is similar to Response Time Analysis (RTA) [4]. RTA calculates the worst case response time of each task. In contrast, RTA-OODH calculates the relative optimal optional deadline of each task.

4. RUN Algorithm

This paper reviews RUN [26], which is optimal multiprocessor real-time scheduling with a small number of preemptions/migrations. The model specific to RUN [26] is introduced because RUN has many original parameters and assumptions to explain itself. The goal of RUN is full system utilization and idle tasks are inserted in order to achieve it. The total utilization of idle tasks is $U_{idle} = M - \sum_i U_i$. Note that each idle task depends on the utilization parameter alone and does not depend on other parameters such as period and WCET.

RUN transforms multiprocessor scheduling into uniprocessor scheduling by aggregating tasks into servers. This paper defines servers as tasks with sequences of jobs but these are not actual tasks in the system; each server is a proxy for a collection of client tasks. When a server is running, the processor time is used by one of its clients. Server clients are scheduled via an internal scheduling mechanism. The utilization of each server S_l is $U_l^{srv} = \sum_{\tau_i \in S_l} U_i$, where $\tau_i \in S_l$ indicates that task τ_i is assigned to server S_l. Note that the utilization of each server does not exceed one (100%). The details of RUN in both offline and online phases are explained as follows.

4.1 Offline Phase

In the offline phase, RUN reduces multiprocessor scheduling to uniprocessor scheduling by the DUAL and PACK operations. RUN is based on EDF because EDF is optimal for implicit-deadline task sets on uniprocessors.

The DUAL operation transforms task τ_i into dual task τ_i^*, whose execution time represents the idle time of τ_i (i.e., $C_i^* = T_i - C_i$). The relative deadline of dual task τ_i^* is equal to that of task τ_i. The DUAL operation reduces the number of processors whenever $n - M < M$.

The PACK operation packs dual servers into packed servers whose utilizations do not exceed one. When $n - M \geq M$, the number of servers can be reduced by aggregating them into fewer servers using the PACK operation. The scheme for packing servers to fewer servers is heuristic. That is to say, the PACK operation is similar to the partitioning schemes (e.g., first-, next-, best-, and worst-fit). Note that if assigning tasks to processors is successful, RUN generates the same schedule as P-EDF and does not perform the DUAL and PACK operations. Otherwise, the DUAL and PACK operations generate the reduction tree offline, which is then used to make server scheduling decisions online. Details on how to make scheduling decisions in the reduction tree are given in the next subsection.

In order to explain the reduction tree, this paper defines the following terms with respect to servers as follows.

- unit server: the utilization of the server is one
- null server: the utilization of the server is zero
- root server: the last packed server whose utilization is one (unit server)

Packing the dual servers of packed servers can reduce the number of servers by nearly half. RUN performs the DUAL and PACK operations repeatedly until all packed servers become unit servers. The REDUCE operation including these operations is defined as follows.

Definition 1 (From Definition IV.6 in Ref. [26]). *Given a set of servers* Γ *and a packing* π *of* Γ*, a* REDUCE *operation on a server* S *in* Γ*, denoted by* $\psi(S)$*, is the composition of the* DUAL *operation* φ *with the* PACK *operation* σ *for* π *(i.e.,* $\psi(S) = \varphi(\sigma(S))$*).*

In addition, this paper defines the reduction level/sequence to explain the reduction tree as follows.

Definition 2 (From Definition IV.7 in Ref. [26]). *Let* $i \geq 1$ *be an integer,* Γ *be a set of servers, and* S *be a server in* Γ*. The operator* ψ^i *is recursively defined by* $\psi^0(S) = S$ *and* $\psi^i(S) = \psi \circ \psi^{i-1}(S)$*. Then* $\{\psi^i\}_i$ *is a* reduction sequence*, and the server system* $\psi^i(\Gamma)$ *is* reduction level *i that represents the number of performing REDUCE operations* (*denoted as i*).

Note that the number of servers at reduction level 0 is the same as that at reduction level 1, if these servers exist.

Figure 5 shows the reduction tree on three processors at time 5. This paper gives further details of the following tuple (C_i, T_i) to explain WCET and task period of τ_i as $\tau_i^{(C_i,T_i)}$, and hence all periods and WCETs of tasks are $\tau_1^{(2,5)}$, $\tau_2^{(4,10)}$, $\tau_3^{(8,20)}$, $\tau_4^{(4,10)}$, and $\tau_5^{(2,5)}$. Tasks τ_1, τ_2, τ_3, τ_4, and τ_5 are assigned to servers S_1, S_2, S_3, S_4, and S_5 at reduction level 0, respectively. The total utilization of idle tasks is $U_{idle} = M - \sum_i^n U_i = 3 - 5 * 0.4 = 1$. In this example, idle tasks are uniformly assigned at reduction level 0, and hence the utilization of each server is added to $U_{idle}/n = 1/5 = 0.2$, respectively.

This paper represents a server as $S_l^{(U_l^{srv}),\{\mathbb{D}_l\}}$, where U_l^{srv} is the utilization of server S_l and $\mathbb{D}_l$ is the deadline set of server S_l. The deadline set includes all absolute task deadlines in the server. Each server sets the earliest deadline in the deadline set when the server is released. This paper assigns deadline sets $5N$, $10N$, $20N$, $10N$, and $5N$ to servers at reduction level 0, respectively, where N indicates a natural number. For example, $5N$ represents all deadlines of the tasks whose relative deadlines are all 5. Servers S_6, S_7, S_8, S_9, and S_{10} are generated by the DUAL operation at reduction level 1 and their utilizations are all 0.4. This is because these servers are dual servers of servers S_1, S_2, S_3, S_4, and S_5 at reduction level 0, respectively. In this example, servers S_6 and S_7 are packed into server S_{11}, servers S_8 and S_9 are packed into server S_{12}, and server S_{10} is packed into server S_{13} by the PACK

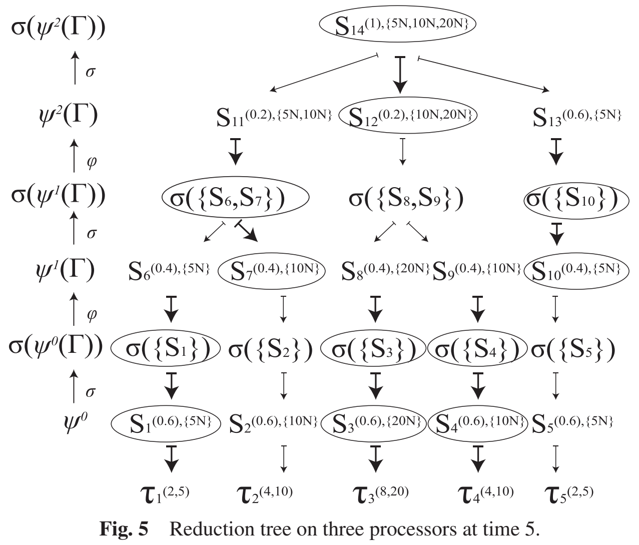


Fig. 5 Reduction tree on three processors at time 5.

operation. Servers S_{11}, S_{12}, and S_{13} are generated by the DUAL operation at reduction level 2. Finally, server S_{14} is generated by the PACK operation at reduction level 2 and its utilization is one. The REDUCE operation is finished and the reduction tree is completely generated.

Note that the number of root servers may become larger than one because when all servers are unit servers at the highest reduction level, the REDUCE operation is finished. If one server is a unit server, then its dual server is a null server that is packed into another server when the next PACK operation is performed.

4.2 Online Phase

In an online phase, RUN schedules servers according to the following rules from Ref. [26] using Fig. 5 for reference.

Rule 1 (From Rule IV.2 in Ref. [26])**.** *If a packed server is running* (*circled*)*, execute the child node with the earliest deadline among those children with work remaining; if a packed server is not running* (*not circled*)*, execute none of its children.*

Rule 2 (From Rule IV.3 in Ref. [26])**.** *Execute* (*circle*) *the child* (*packed server*) *of a dual server if and only if the dual server is not running* (*not circled*).

In the reduction tree, a thick arrow represents a scheduled server and a thin arrow represents a non-scheduled server according to each parent server. If a thick arrow from a server points to a task, the server schedules the task.

In Fig. 5, root server S_{14} is always running regardless of these rules, because a root server is always a unit server. Next, S_{14} makes scheduling decisions in EDF order. Server S_{12} is running at time 5 because of work remaining. Since server S_{12} is running, S_8 and S_9 are not running by Rule 1. Since servers S_{11} and S_{13} are not running, servers S_7 and S_{10} are running by Rule 2. Servers S_6, S_8, and S_9 are not running, and hence servers S_1, S_3, and S_4 are running by Rule 2.

Figure 6 shows an example of RUN scheduling on three processors. Each server is executed on virtual processor $VP_{L,v}$, where L represents the reduction level and v represents the virtual processor ID at each reduction level. The task set is shown in Fig. 5; this example shows the scheduling decisions at time 5. This system has three processors P_1, P_2, and P_3. Reduction level 0 has three virtual processors $VP_{0,1}$, $VP_{0,2}$, and $VP_{0,3}$, reduction level 1 has two virtual processors $VP_{1,1}$ and $VP_{1,2}$, and reduction level 2 has one virtual processor $VP_{2,1}$. The hyperperiod of all tasks (least common multiple of $T_1, T_2, \ldots, T_n$) is 20 and an example of RUN scheduling is shown in the interval $[0, 10)$. Note that the example of RUN scheduling in the interval $[10, 20)$ is the same as that in the interval $[0, 10)$.

RUN uses the following task-to-processor assignment scheme: (1) leave executing tasks on their current processors, (2) assign idle tasks to their last-used processor when available, in order to avoid unnecessary migrations, and (3) assign remaining tasks to free processors arbitrarily. According to this scheme, each server assigns tasks to processors P_1, P_2, or P_3 in Fig. 6. When each task completes its execution on one processor, the processor becomes idle until the remaining execution time of server of each task becomes zero. For example, server S_5, running on $VP_{0,1}$, completes task τ_5 on processor P_1 at time 3 and P_1 becomes idle (executes idle task) during the time interval $[3, 4)$.

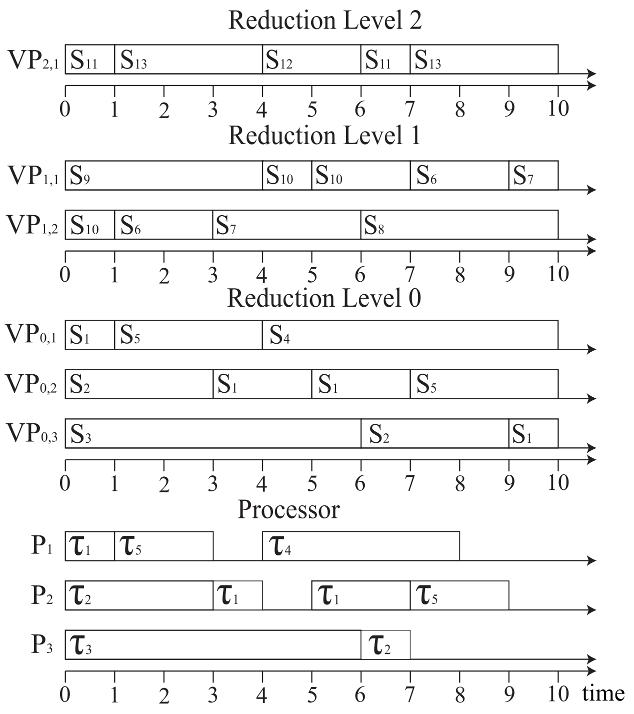


Fig. 6 Example of RUN scheduling on three processors.

(1) When server S_l becomes ready: if the running server with the lowest priority has a lower priority than S_l, preempt the running server.
(2) When server S_l starts running on virtual processor $VP_{L,v}$:
  (a) If S_l is at reduction level 0, schedule ready tasks in S_l in RMWP order.
  (b) Otherwise, S_l schedules the child servers of reduction tree by Rules 1 and 2.
(3) When server S_l goes to sleep: set the release time to the next earliest deadline of all tasks in S_l and put S_l to sleep until the S_l release time.

Fig. 7 RUN-RMWP algorithm.

5. The RUN-RMWP Algorithm

This paper proposes the RUN-RMWP algorithm to achieve optimal multiprocessor semi-fixed-priority scheduling with harmonic periodic task sets. As well as RUN, RUN-RMWP makes server schedules in EDF order, and hence RUN-RMWP can use Rules 1 and 2. In addition, RUN-RMWP makes task scheduling decisions in RMWP order [12]. Using the idea of combination of server and task scheduling, RUN-RMWP achieves optimal multiprocessor real-time scheduling in the extended imprecise computation model.

Figure 7 shows the RUN-RMWP algorithm. RUN-RMWP makes server scheduling decisions under the following conditions: (1) server S_l becomes ready, (2) server S_l starts running on virtual processor $VP_{L,v}$, and (3) server S_l goes to sleep. Conditions (1) and (3) in RUN-RMWP are the same as those in RUN, and hence RUN-RMWP and RUN generate the same server scheduling. In order to make task scheduling decisions under condition (2) in RUN-RMWP, this paper extends the technique to calculate the relative optimal optional deadline of each task in RMWP for RUN-RMWP. RMWP schedules tasks running on a processor while RUN-RMWP schedules servers running on a virtual processor, where the utilization of each server may be less than one (an important difference between processors and servers).

5.1 Optional Deadline

This paper now calculates the relative optional deadline of each task in RUN-RMWP using RTA-OODH [12]. The relative optional deadline of each task in RUN-RMWP depends on the execution time of the server, whose utilization may be less than one. That is to say, the optional deadline of each task is not fixed against the processor time because a server (except for the root server) might not always be running.

First of all, this paper analyzes the assignable time A_k of task τ_k except w_k in server S_l.

Theorem 5 (Assignable Time in RUN-RMWP)**.** *The assignable time A_k of task τ_k except w_k in server S_l in RUN-RMWP on multiprocessors is*

$$A_k = D_k \cdot U_l^{srv} - w_k - \sum_{\forall i:\tau_i,\tau_k \in S_l \wedge p_i > p_k} I_k^i, \qquad (1)$$

where I_k^i is from Theorem 1.

Proof. The differences between this theorem and Theorem 2 are that (1) the first parameter D_k is transformed into $D_k \cdot U_l^{srv}$ and (2) higher priority tasks than τ_k in server S_l interfere with task τ_k. The least common multiple of periods of task τ_k and higher priority tasks than τ_k is equal to T_k (D_k) with harmonic periodic task sets. Next, D_k is changed into $D_k \cdot U_l^{srv}$ because this theorem considers that the utilization of server S_k is less than or equal to one. The worst case interference time of task τ_k considers only higher priority tasks than τ_k in server S_l. Note that the worst case interference time of each job is constant with harmonic periodic task sets. Since the assignable time A_k of task τ_k except w_k is equal to Eq. (1), this theorem holds. □

Theorem 6 (Worst Case Interference Time in $[0, OD_k)$ in RUN-RMWP)**.** *The worst case interference time I_k of each task τ_k in server S_l in RUN-RMWP on multiprocessors is*

$$I_k = \sum_{\forall i:\tau_i,\tau_k \in S_l \wedge p_i > p_k} \left(\left\lceil \frac{OD_k}{T_i} \right\rceil m_i + \left\lceil \frac{OD_k - OD_i}{T_i} \right\rceil w_i \right).$$

Proof. The difference between this theorem and Theorem 3 is that this theorem considers only tasks assigned to each server. As well as the proof of Theorem 3 (found in Ref. [12]), one task τ_i is split into two general tasks τ_i^m and τ_i^w. Task τ_i^m releases the first job at time 0 and its period is T_i. Task τ_i^w releases the first job at time OD_i and its period is T_i. Hence, in $[0, OD_k)$, task τ_k is interfered with by τ_i^m in $\lceil OD_k/T_i \rceil$ times and by τ_i^w in $\lceil (OD_k - OD_i)/T_i \rceil$ times. □

Theorem 7 (Optional Deadline in RUN-RMWP)**.** [*Meaning of Optimal* (2)] *The relative optimal optional deadline OD_k of task τ_k in server S_l in RUN-RMWP on multiprocessors according to RTA-OODH is*

$$OD_k = A_k + I_k, \qquad (2)$$

where A_k and I_k are from Theorems 5 and 6, respectively.

Proof. In Eq. (2), the relative optional deadline OD_k and assignable time A_k of task τ_k are the response time and WCET in RTA [4], respectively. The relative optional deadline of task τ_k by Eq. (2) means that $D_k - OD_k$ is equal to the sum of the WCET of its wind-up part w_k and the worst case interference time from higher priority tasks. The assignable time of each job is constant with harmonic periodic task sets. The assignable time A_k of task τ_k except w_k in $[OD_k, D_k)$ is equal to w_k, and hence the relative optional deadline by Eq. (2) is optimal. □

```
$RTA - OODH(\Gamma)$ {
    while ($\tau_k \in \Gamma$) {
        $A_k = D_k \cdot U_l^{srv} - w_k - \sum_{\forall i:\tau_i,\tau_k \in S_l \wedge p_i > p_k} I_k^i$;
        $I_k = 0$;
        do {
            $OD_k = A_k + I_k$;
            $I_k = \sum_{\forall i:\tau_i,\tau_k \in S_l \wedge p_i > p_k} \left( \left\lceil \frac{OD_k}{T_i} \right\rceil m_i + \left\lceil \frac{OD_k - OD_i}{T_i} \right\rceil w_i \right)$;
        } while ($A_k + I_k > OD_k$);
    }
}
```

Fig. 8 Pseudo code of RTA-OODH in RUN-RMWP.

Figure 8 shows the pseudo code of RTA-OODH in RUN-RMWP. This pseudo code calculates the relative optimal optional deadline by iteration, similarly to RTA-OODH in RMWP [12]. Using this calculation offline, RUN-RMWP avoids missing the deadline due to the overrun of the optional part online.

5.2 Optional Deadline Timer

The optional deadline in RUN-RMWP by Theorem 7 depends on the utilization of the server, which may be less than one. In order to terminate the optional part and release the wind-up part at the optional deadline, RUN-RMWP must manage the optional deadline timer, which generates the timer interrupt at the optional deadline. This paper defines the current execution time and completing time of server S_l at reduction level 0 as $ET(S_l)$ and $CT(S_l)$, respectively. The completing time $CT(S_l)$ is the time to require completing server S_l. This paper explains how to manage the optional deadline timer in RUN-RMWP as follows:

- When server S_l is released, set $ET(S_l) = 0$.
- When server S_l starts running at time t, start all optional deadline timers of tasks in server S_l (if their optional deadlines do not expire), set the optional deadline timer of task τ_i to $t + OD_i - ET(S_l)$.
- When server S_l is preempted after running in time interval ti, stop all optional deadline timers of tasks in server S_l (if started) and set $ET(S_l) = ET(S_l) + ti$. When $ET(S_l) = CT(S_l)$, server S_l is completed and goes to sleep.

RMWP-based algorithms except RUN-RMWP set the optional deadline timer of each task if it becomes ready because each processor is always running and the optional deadline is fixed against the processor time. However, RUN-RMWP sets the optional deadline timer of each task if its server starts running because each server is not always running and the optional deadline is not fixed against the processor time.

5.3 Example

Figure 9 shows an example of RUN-RMWP scheduling on three processors using the task set listed in **Table 1**. In addition, the utilization of each task is equal to that shown in Fig. 5

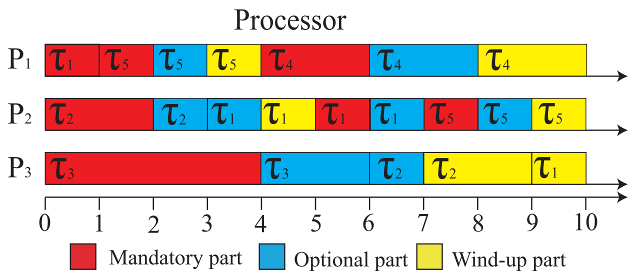


Fig. 9 Example of RUN-RMWP scheduling on three processors.

Table 1 Task set.

Task	m_i	o_i	w_i	OD_i	D_i	T_i
τ_1	1	2	1	2	5	5
τ_2	2	4	2	4	10	10
τ_3	4	8	4	8	20	20
τ_4	2	4	2	4	10	10
τ_5	1	2	1	2	5	5

because $C_i = m_i + w_i$ and all relative deadlines and periods are the same. Tasks τ_1, τ_2, τ_3, τ_4, and τ_5 are uniformly assigned at reduction level 0, respectively. The utilization of each task is 0.4 ($U_i = C_i/T_i = (m_i + w_i)/T_i$) and the total utilization of idle task is $U_{idle} = M - \sum_i^n U_i = 3 - 5 * 0.4 = 1$. In this example, idle tasks are uniformly assigned at reduction level 0 and the utilization of each idle task is $U_{idle}/n = 1/5 = 0.2$, respectively. Therefore, the utilization of each server is 0.6 (= 0.4 + 0.2). Making server scheduling decisions in RUN-RMWP is the same as that in RUN. The optional deadline of each task is calculated by Theorem 7. Note that the optional deadline timer of each task may be restarted because the utilization of each server is 0.6 (less than 1.0). Now how to manage the optional deadline timer of task τ_2 in server S_2 is explained by using Fig. 6 for reference.

- At time 0, server S_2 is released and $ET(S_2)$ is set to zero. At the same time, server S_2 starts running and the optional deadline timer for τ_2 is set to $t + OD_2 - ET(S_2) = 0 + 4 - 0 = 4$. Note that $CT(S_2) = D_2 * U_2^{srv} = 10 * 0.6 = 6$.
- At time 3, server S_2 is preempted after running in time interval 3, the optional deadline timer for τ_2 is stopped and $ET(S_2) = 3$.
- At time 6, server S_2 starts running and the optional deadline timer for τ_2 is set to $t + OD_2 - ET(S_2) = 6 + 4 - 3 = 7$.
- At time 7, the optional deadline timer for task τ_2 expires, and hence task τ_2 terminates its optional part and starts its wind-up part.
- At time 9, task τ_2 completes its wind-up part. At the same time, server S_2 is also completed and goes to sleep after running in time interval 3 because $ET(S_2) = 6$ (i.e., $ET(S_2) = CT(S_2)$).

Each task executes its optional part without deadline miss, thanks to its optional deadline timer, and hence RUN-RMWP can achieve its optimality and improve the QoS.

5.4 Schedulability Analysis

The optimality of RUN-RMWP is analyzed by using the following theorems.

Theorem 8 (From Theorem IV.3 in Ref. [26]). [*Meaning of Optimal (1)*] *RUN is an optimal multiprocessor real-time scheduling algorithm.*

Theorem 9 (From Theorem 8 in Ref. [12]). [*Meaning of Optimal (1)*] *RMWP is an optimal uniprocessor real-time scheduling algorithm with harmonic periodic task sets.*

Using these theorems, this paper next analyzes the optimality of RUN-RMWP with harmonic periodic task sets.

Theorem 10 (Optimality of RUN-RMWP). [*Meaning of Optimal (1)*] *RUN-RMWP is an optimal multiprocessor real-time scheduling algorithm with harmonic periodic task sets.*

Proof. RUN-RMWP and RUN generate the same reduction tree and server scheduling, and hence making server scheduling decisions in RUN-RMWP is optimal. Here, RUN transforms uniprocessor EDF scheduling into multiprocessor scheduling. Since EDF is an optimal uniprocessor real-time scheduling algorithm, RUN is optimal by Theorem 8. RMWP is an optimal uniprocessor scheduling algorithm with harmonic periodic task sets by Theorem 9, and hence making task scheduling decisions in RUN-RMWP is also optimal. Hence, this theorem holds. □

By Theorem 10, RUN-RMWP achieves optimal multiprocessor real-time scheduling and reveals how to apply RUN for imprecise computation with harmonic periodic task sets.

6. Simulation Studies

6.1 Simulation Setups

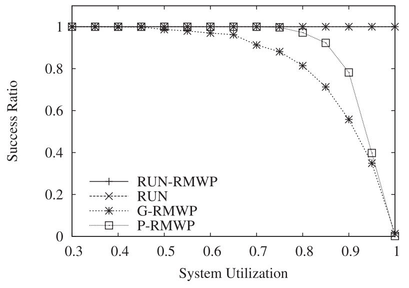
This simulation uses 1,000 task sets in each system utilization. The system utilization U is selected within $[0.3, 0.35, 0.4, \ldots, 1.0]$. RUN-RMWP, RUN, G-RMWP, and P-RMWP algorithms are evaluated. In simulation environments, the number of processors M is 4. Each U_i is selected within $[0.02, 0.03, 0.04, \ldots, 1.0]$ and is split into two utilizations that are assigned to m_i and w_i, respectively, for imprecise-based algorithms (i.e., RUN-RMWP, G-RMWP, and P-RMWP). The splitting algorithm is such that m_i is first selected within $[0.01, 0.02, \ldots, U_i - 0.01]$ and w_i is next set to $U_i - m_i$. In contrast, RUN does not perform the above operation. This paper assumes that the ACET of each task fluctuates and is usually shorter than its WCET. The ratio of ACET/WCET is set to the ranges of 1.0, [0.75, 1.0], and [0.5, 1.0]. The period T_i of each task τ_i is selected within [100, 200, 400, 800, 1600]. Each U_i^o is selected within [0.01, 1.0], represented as RUN-RMWP-OP [*1] if the evaluated algorithm is RUN-RMWP. If U_i^o is always equal to zero, the result is represented as RUN-RMWP. The simulation length of each task set is the hyperperiod of all tasks.

All tasks are assigned to (1) servers in RUN-RMWP and RUN or (2) processors in P-RMWP, by the worst-fit decreasing utilization heuristic. In order to achieve full system utilization, RUN-RMWP and RUN uniformly assign idle tasks at reduction level 0, as in the task set shown in Fig. 5. If the utilization of server S_l at reduction level 0 exceeds one, the overutilization of server (i.e., $U_l^{srv} - 1$) is uniformly reassigned to other servers at reduction level 0 until the remaining utilization of idle tasks becomes zero.

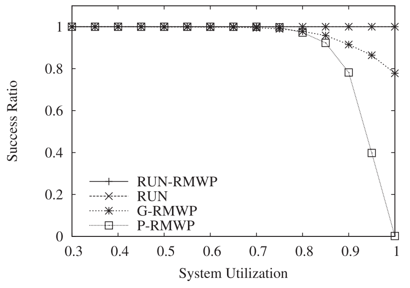
The performance metrics are defined by the following equations.

$$\text{Success Ratio} = \frac{\text{\# of successfully scheduled task sets}}{\text{\# of scheduled task sets}}$$

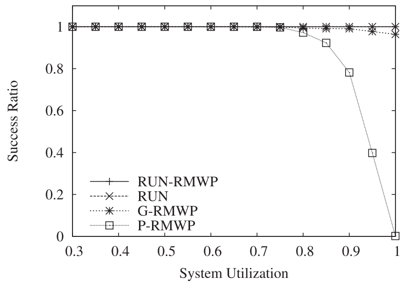
[*1] OP represents optional part.



(a) ACET/WCET = 1.0

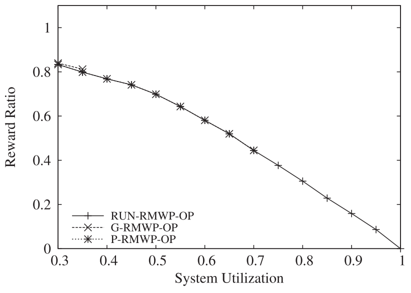


(b) ACET/WCET = [0.75,1.0]

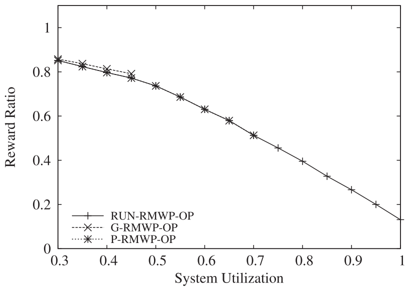


(c) ACET/WCET = [0.5,1.0]

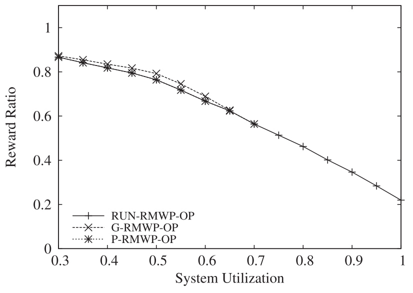
Fig. 10 Success ratio.



(a) ACET/WCET = 1.0

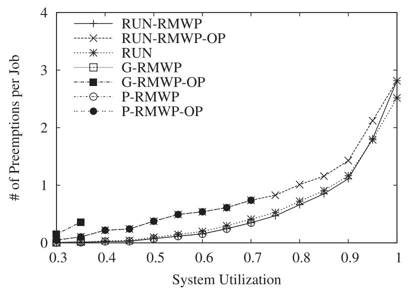


(b) ACET/WCET = [0.75,1.0]

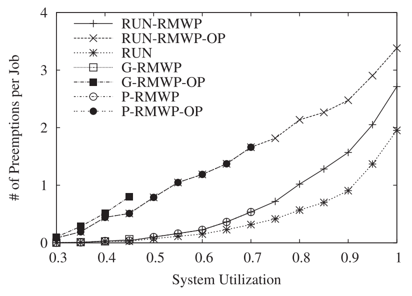


(c) ACET/WCET = [0.5,1.0]

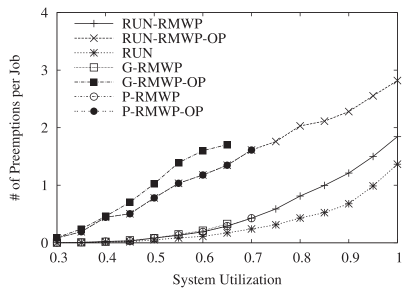
Fig. 11 Reward ratio.



(a) ACET/WCET = 1.0



(b) ACET/WCET = [0.75,1.0]



(c) ACET/WCET = [0.5,1.0]

Fig. 12 Number of preemptions per job.

$$\text{Reward Ratio} = \frac{1}{n}\sum_i \frac{1}{\text{\# of jobs of task } \tau_i}\sum_j \frac{o_{i,j}}{o_i}$$

$$\text{\# of Preemptions per Job} = \frac{1}{n}\sum_i \frac{\text{\# of preemptions of task } \tau_i}{\text{\# of jobs of task } \tau_i}$$

$$\text{\# of Migrations per Job} = \frac{1}{n}\sum_i \frac{\text{\# of migrations of task } \tau_i}{\text{\# of jobs of task } \tau_i}$$

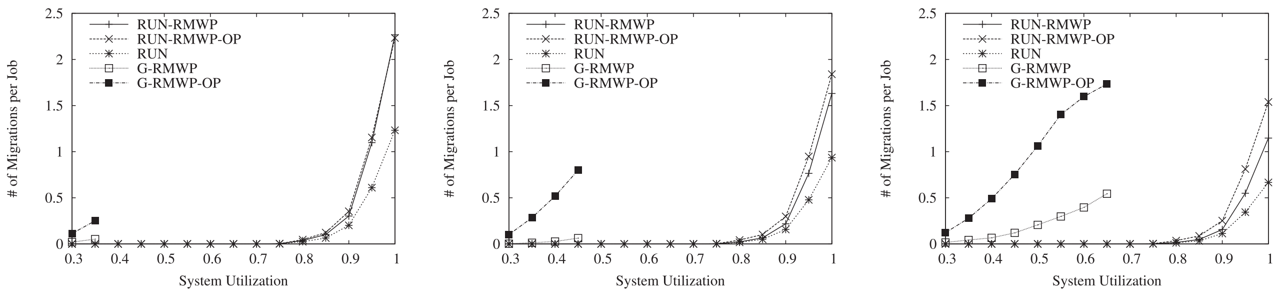
Optimal target full system utilization is a success ratio and satisfies Meaning of Optimal (1). If the success ratio of each system utilization is less than one, the results of the system utilization evaluated by other performance metrics are omitted. The reward ratio is a metric to measure the QoS of each task for imprecise computation. The more the QoS of each task is increased, the higher the reward ratio becomes. Depending on the evaluated algorithms, $o_{i,j}$, the time to be actually executed in the j^{th} job of task τ_i, is different.

6.2 Simulation Results

Figure 10 shows the simulation results of the success ratio. The success ratios of RUN-RMWP and RUN for all results are always one because they are optimal. G-RMWP lowers the success ratio when the system utilization is high. If the ACET of each task is shorter than its WCET, G-RMWP can improve the success ratio thanks to global scheduling. In contrast, P-RMWP generates the same results and always lowers the success ratio when the system utilization exceeds 0.7.

Figure 11 shows the simulation result of reward ratio. G-RMWP-OP slightly outperforms RUN-RMWP-OP and P-RMWP-OP when the system utilization is low, thanks to global scheduling. However, G-RMWP-OP does not show the results when the system utilization is high because G-RMWP-OP lowers the success ratio, as shown in Fig. 10. RUN-RMWP-OP has the same results as P-RMWP-OP when the system utilization is low. When the system utilization is high, P-RMWP-OP does not show the results and only RUN-RMWP-OP shows results.

Figure 12 shows the simulation results of the number of preemptions per job. RUN-RMWP has similar results to RUN if the optional part of each task is not executed. RUN-RMWP-OP has slightly more preemptions than RUN because the optional part of each task is executed, and hence there is a trade-off between preemption and QoS. The number of preemptions per job in RUN-RMWP, RUN-RMWP-OP, and RUN is small and at most 3.4. Therefore, RUN-RMWP and RUN-RMWP-OP inherit the advantages of RUN with respect to the small number of preemp-



(a) ACET/WCET = 1.0 (b) ACET/WCET = [0.75,1.0] (c) ACET/WCET = [0.5,1.0]

Fig. 13 Number of migrations per job.

tions per job. P-RMWP performs as well as RUN-RMWP when the system utilization does not exceed 0.7. However, when the system utilization exceeds 0.7, P-RMWP lowers the success ratio.

The number of preemptions per job in RUN-RMWP-OP is the largest because there are many opportunities for terminating the optional part of each task, which incurs many preemptions. Interestingly, the number of preemptions per job in RUN-RMWP-OP when the ratio of ACET/WCET is $[0.5, 1.0]$ is smaller than that when the ratio of ACET/WCET is $[0.75, 1.0]$. This is because there are many opportunities to complete the optional part of each task before it is preempted by higher priority tasks.

Figure 13 shows the simulation results of the number of migrations per job. When the system utilization does not exceed 0.65, RUN-RMWP, RUN-RMWP-OP, and RUN are zero because assigning tasks to processors is successful. When the system utilization exceeds 0.65, the number of migrations per job is increased but is at most 2.3. Therefore, RUN-RMWP and RUN-RMWP-OP also inherit the advantages of RUN with respect to the small number of migrations per job.

When the system utilization is low, G-RMWP and G-RMWP-OP show results and have a larger number of migrations per job than other algorithms. In particular, G-RMWP-OP is the largest in evaluated algorithms because the optional part of each task is migrated and executed on different processors frequently due to global scheduling. The number of migrations per job in RUN-RMWP-OP when the ratio of ACET/WCET is $[0.5, 1.0]$ is smaller than that when the ratio of ACET/WCET is $[0.75, 1.0]$. This is because there are many opportunities to complete the mandatory and wind-up parts of each task well before it is migrated thanks to the shorter ACET.

From these results, RUN-RMWP outperforms G-RMWP and P-RMWP. In addition, it has a slightly larger number of preemptions/migrations per job than RUN because it supports the extended imprecise computation model. In actual systems, the additional overheads of imprecise computation using G-RMWP and P-RMWP were investigated and found to be low and comparable to the overheads of G-RM and P-RM, respectively [15]. Therefore, this paper believes that the overhead of RUN-RMWP will be sufficient in actual systems.

7. Related Work

Optimal multiprocessor real-time scheduling has been achieved by Pfair algorithm [7] that keeps execution times close to fluid scheduling. Pfair incurs significant run-time overhead due to the quantum-based scheduling approach. The practicality of Pfair is investigated by Brandenburg to evaluate PD^2 [1], which is an extension of Pfair to reduce the number of preemptions/migrations. These algorithms are compared with other non-optimal multiprocessor real-time scheduling algorithms on Intel's 24-core processors [10]. Experimental results show that PD^2 has the worst schedulability of all evaluated algorithms, and hence Pfair does not work well in actual systems.

There are some other optimal multiprocessor real-time scheduling algorithms: Largest Local Remaining Execution time First (LLREF) [18], EDF with task splitting and K processors in a group (EKG) [3], Deadline Partitioning Wrap (DP-Wrap) [22], and RUN [26]. Simulation results by Regnier et al. show that RUN outperforms LLREF, EKG, and DP-Wrap in the number of preemptions/migrations per job and scales well as the number of tasks/processors is increased [26].

There are imprecise-based real-time scheduling algorithms on uniprocessors including Mandatory-First with Earliest Deadline [8] and Optimization with Least-Utilization [5]. However, they support the imprecise computation model [23] and do not support the extended imprecise computation model [20].

Mandatory-First with Wind-up Part [20] and Slack Stealer for Optional Parts [21] are proposed to support the extended imprecise computation model on uniprocessors but they do not support multiprocessors. G-RMWP [13] and P-RMWP [15] support multiprocessors in the extended imprecise computation model but they are not optimal. In contrast, RUN-RMWP is optimal and supports the extended imprecise computation model, and hence RUN-RMWP has the advantage over G-RMWP and P-RMWP.

8. Conclusion

This paper proposes the new algorithm RUN-RMWP to achieve optimal multiprocessor semi-fixed-priority scheduling with harmonic periodic task sets (Meaning of Optimal (1)). RUN-RMWP integrates RUN and RMWP to inherit the advantages of these algorithms that achieve a small number of preemptions/migrations per job and support the extended imprecise computation model. RUN-RMWP calculates the optional deadline of each task by extending RTA-OODH. In addition, RUN-RMWP manages an optional deadline timer to terminate the optional part and release the wind-up part of each task in each server. Simulation results show that RUN-RMWP outperforms other non-optimal multiprocessor real-time scheduling algorithms including G-RMWP and P-RMWP relative to the success ratio and the number of preemptions/migrations per job. RUN-RMWP has a few more preemptions/migrations than RUN. However, RUN-RMWP supports the extended imprecise computation model, and hence RUN-RMWP is well suited to imprecise real-time applications such as humanoid robots.

In future work, RUN-RMWP will be implemented in RT-Est [14], which is a real-time operating system for semi-fixed-priority scheduling algorithms in the extended imprecise computation model. The overhead-aware schedulability of RUN-RMWP will be analyzed using the preemption-aware interrupt accounting method [10]. The integration of RMWP++ [17] and RUN-RMWP is interesting. In addition, RUN-RMWP will be adapted to the multiple mandatory parts [16] and the parallel-extended imprecise computation model [11].

References

[1] Anderson, J.H. and Srinivasan, A.: Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks, *J. Comput. Syst. Sci.*, Vol.68, No.1, pp.157–204 (2004).

[2] Andersson, B. and Jonsson, J.: The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors are 50%, *Proc. 15th Euromicro Conference on Real-Time Systems*, pp.33–40 (2003).

[3] Andersson, B. and Tovar, E.: Multiprocessor Scheduling with Few Preemptions, *Proc. 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp.322–334 (2006).

[4] Audsley, N.C., Burns, A., Richardson, M.F., Tindell, K. and Wellings, A.J.: Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling, *Software Engineering Journal*, Vol.8, No.5, pp.284–292 (1993).

[5] Aydin, H., Melhem, R., Mosse, D. and Mejfa-Alvarez, P.: Optimal Reward-Based Scheduling of Periodic Real-Time Tasks, *Proc. 20th IEEE Real-Time Systems Symposium*, pp.79–89 (1999).

[6] Baker, T.P.: An Analysis of Fixed-Priority Schedulability on a Multiprocessor, *Real-Time Systems*, Vol.32, No.1-2, pp.49–71 (2006).

[7] Baruah, S.K., Cohen, N.K., Plaxton, C.G. and Varvel, D.A.: Proportionate Progress: A Notion of Fairness in Resource Allocation, *Algorithmica*, Vol.15, No.6, pp.600–625 (1996).

[8] Baruah, S.K. and Hickey, M.E.: Competitive On-line Scheduling of Imprecise Computations, *IEEE Trans. Comput.*, Vol.47, No.9, pp.1027–1032 (1998).

[9] Bonifaci, V., Marchetti-Spaccamela, A., Megow, N. and Wiese, A.: Polynomial-Time Exact Schedulability Tests for Harmonic Real-Time Tasks, *Proc. 34th IEEE Real-Time Systems Symposium*, pp.236–245 (2013).

[10] Brandenburg, B.B.: Scheduling and Locking in Multiprocessor Real-time Operating Systems, Ph.D. Thesis, The University of North Carolina at Chapel Hill (2011).

[11] Chishiro, H.: RT-Seed: Real-Time Middleware for Semi-Fixed-Priority Scheduling, *Proc. 19th IEEE International Symposium on Real-Time Computing*, pp.124–133 (2016).

[12] Chishiro, H., Takeda, A., Funaoka, K. and Yamasaki, N.: Semi-Fixed-Priority Scheduling: New Priority Assignment Policy for Practical Imprecise Computation, *Proc. 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp.339–348 (2010).

[13] Chishiro, H. and Yamasaki, N.: Global Semi-Fixed-Priority Scheduling on Multiprocessors, *Proc. 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp.218–223 (2011).

[14] Chishiro, H. and Yamasaki, N.: RT-Est: Real-Time Operating System for Semi-Fixed-Priority Scheduling Algorithms, *Proc. 2011 International Symposium on Embedded and Pervasive Systems*, pp.358–365 (2011).

[15] Chishiro, H. and Yamasaki, N.: Experimental Evaluation of Global and Partitioned Semi-Fixed-Priority Scheduling Algorithms on Multicore Systems, *Proc. 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp.127–134 (2012).

[16] Chishiro, H. and Yamasaki, N.: Semi-Fixed-Priority Scheduling with Multiple Mandatory Parts, *Proc. 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp.1–8 (2013).

[17] Chishiro, H. and Yamasaki, N.: Zero-Jitter Semi-Fixed-Priority Scheduling with Harmonic Periodic Task Sets, *International Journal of Computers and Their Applications*, Vol.22, No.3, pp.118–126 (2015).

[18] Cho, H., Ravindran, B. and Jensen, E.D.: An Optimal Real-Time Scheduling Algorithm for Multiprocessors, *Proc. 27th IEEE Real-Time Systems Symposium*, pp.101–110 (2006).

[19] Fan, M. and Quan, G.: Harmonic Semi-Partitioned Scheduling For Fixed-Priority Real-Time Tasks On Multi-Core Platform, *2012 Design, Automation & Test in Europe*, pp.503–508 (2012).

[20] Kobayashi, H. and Yamasaki, N.: An Integrated Approach for Implementing Imprecise Computations, *IEICE Trans. Inf. Syst.*, Vol.86, No.10, pp.2040–2048 (2003).

[21] Kobayashi, H. and Yamasaki, N.: RT-Frontier: A Real-Time Operating System for Practical Imprecise Computation, *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp.255–264 (2004).

[22] Levin, G., Funk, S., Sadowski, C., Pye, I. and Brandt, S.: DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling, *Proc. 22nd Euromicro Conference on Real-Time Systems*, pp.3–13 (2010).

[23] Lin, K., Natarajan, S. and Liu, J.: Imprecise Results: Utilizing Partial Computations in Real-Time Systems, *Proc. 8th IEEE Real-Time Systems Symposium*, pp.210–217 (1987).

[24] Liu, C.L. and Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *J. ACM*, Vol.20, No.1, pp.46–61 (1973).

[25] Mizuuchi, I., Nakanishi, Y., Sodeyama, Y., Namiki, Y., Nishino, T., Muramatsu, N., Urata, J., Hongo, K., Yoshikai, T. and Inaba, M.: Advanced Musculoskeletal Humanoid Kojiro, *Proc. 2007 IEEE-RAS International Conference on Humanoid Robots*, pp.294–299 (2007).

[26] Regnier, P., Lima, G., Massa, E., Levin, G. and Brandt, S.: RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor, *Proc. 32nd IEEE Real-Time Systems Symposium*, pp.104–115 (2011).

[27] Taira, T., Kamata, N. and Yamasaki, N.: Design and Implementation of Reconfigurable Modular Robot Architecture, *Proc. 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp.3566–3571 (2005).

Hiroyuki Chishiro was born in 1985. He received his B.S., M.S., and Ph.D. degrees from Keio University in 2008, 2010, and 2012, respectively. He joined the Information Processing Society of Japan in 2009. He became a research fellow of the Japan Society for the Promotion of Science (PD) in 2012, a research associate at Keio University in 2014, and an assistant professor at Advanced Institute of Industrial Technology in 2016. He is presently a project lecturer at The University of Tokyo in 2017. His research interests are real-time systems, operating systems, middleware, and trading systems. He is a member of IEEE and ACM.