Volume 82, issue 1, January 2009
ISSN 0164-1212
ELSEVIER
The Journal of Systems and Software
Special Issue: Software Performance – Modeling and Analysis

The Journal of Systems and Software 82 (2009) 144–154

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

An anomaly prevention approach for real-time task scheduling ☆

Ya-Shu Chen [a], Li-Pin Chang [b], Tei-Wei Kuo [c,*], Aloysius K. Mok [d]

[a] Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan
[b] Department of Computer Science, National Chiao-Tung University, Hsin-Chu 300, Taiwan
[c] Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan
[d] Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA

ARTICLE INFO

Article history:
Received 8 June 2006
Received in revised form 2 July 2008
Accepted 21 July 2008
Available online 9 August 2008

Keywords:
Scheduling anomaly
Real-time task scheduling
Process synchronization
Scheduler stability

ABSTRACT

This research responds to practical requirements in the porting of embedded software over platforms and the well-known multiprocessor anomaly. In particular, we consider the task scheduling problem when the system configuration changes. With mutual-exclusive resource accessing, we show that new violations of the timing constraints of tasks might occur even when a more powerful processor or device is adopted. The concept of scheduler stability and rules are then proposed to prevent scheduling anomaly from occurring in task executions that might be involved with task synchronization or I/O access. Finally, we explore policies for bounding the duration of scheduling anomalies.

1. Introduction

One of the key factors in the development of successful embedded-system products is reusing software and hardware IP's to shorten the time to the market, and maximize the reconfigurability in product development. The issues in software-IP reuse are very different from those in hardware-IP reuse, especially since many applications today must have good response time and high reliability. Such requirements in system implementations often imply the need for software portability over platforms, not just for functionality but also in terms of the timing behaviors of the target software (Buttazzo, 2002). This observation underlies the motivation of this study which the explores real-time task scheduling methodology that could retain selected timing behaviors of task executions when system platforms change.

The real-time resource allocation problem has been an active research topic in the past decades. Optimal scheduling algorithms such as the rate-monotonic scheduling algorithm, the earliest-deadline-first algorithm, and the least-slack-first algorithm (Liu and Layland, 1973; Mok, 1993) have been proposed for different contexts. Real-time task scheduling was also explored for different system architecture assumptions (such as those for multiprocessor scheduling (Baker, 2003; Dhall, 1977), scheduling with end-to-end deadlines (Chen et al., 2000; Kao and Garcia-Molina, 1997), imprecise computation (Lin et al., 1987), and probabilistic performance guarantee (Abeni and Buttazzo, 1998; Abeni and Buttazzo, 1999). Many excellent scheduling algorithms have been proposed (such as those for independent task scheduling (Liu and Layland, 1973; Mok, 1993), task synchronization (Sha et al., 1990), multiframe scheduling (Mok and Chen, 1997), and rate-based scheduling (Jeffay and Goddard, 1999; Liu and Goddard, 2003; Spuri et al., 1995). However, there has been little work addressing the scheduling anomaly problem for platform changing in real-time task scheduling.

This study is motivated by the practical needs of embedded-system implementations when system platforms change, and the well-known multiprocessor anomaly (Manimaran and Siva Ram Murthy, 1997; Mok, 2000; Graham, 1976; Shen et al., 1990; Shen et al., 1993). We consider the task scheduling problem for software portability, in which performance requirements might be violated. We show that new violations of the timing constraints of tasks might occur even when a more powerful processor or device is adopted. The occurrence of these new violations is referred to as scheduling anomaly in this paper. We propose the concept of scheduler stability and anomaly-prevention rules to avoid scheduling anomaly. We consider task executions that might be involved with task synchronization or I/O access. Finally, we explore policies for bounding the duration time of scheduling anomaly. A series of experiments was conducted to evaluate the capability of the proposed anomaly-prevention rules, for which we present very encouraging results.

☆ This paper is an extended version of the paper that appeared in Chen et al. (2005).
* Corresponding author.
E-mail addresses: yschen@mail.ntust.edu.tw (Y.-S. Chen), lpchang@cs.nctu.edu.tw (L.-P. Chang), ktw@csie.ntu.edu.tw (T.-W. Kuo), mok@cs.utexas.edu (A.K. Mok).

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.07.038

The rest of this paper is organized as follows: Section 2 presents the research motivation and problem definition. Section 3 shows the non-existence of greedy scheduling algorithms in avoiding scheduling anomalies and defines a necessary condition for any occurrence of a scheduling anomaly. Section 4 first presents two rules for anomaly prevention and then a methodology to bound the duration time of a scheduling anomaly. Section 5 presents the performance evaluation for our algorithms. Finally, Section 6 summarizes the paper and indicates directions for future research.

2. Problem definition

2.1. Motivation

The motivation of this research could be illustrated by an example schedule over a uniprocessor system in which a violation of timing constraints might occur, due to the upgrading of the processor:

Suppose that there are two tasks τ_1 and τ_2 in this system, where the priority of τ_1 is higher than that of τ_2, and both need to access the same resource R exclusively. In the original schedule, as shown in Fig. 1a, τ_2 first runs without acquiring R (referred to as $\tau_{2,1}$). Later τ_1 arrives and preempts τ_2 (referred to as $\tau_{1,1}$). Then τ_1 locks R (referred to as $\tau_{1,2}$) and later unlocks it and executes until its completion (referred to as $\tau_{1,3}$). When τ_1 is completed τ_2 resumes. τ_2 requests R and executes until τ_2 unlocks R (referred to as $\tau_{2,2}$), at which point, τ_2 executes and completes its execution (referred to as $\tau_{2,3}$). Suppose that the processor is now upgraded such that the duration of each subtask $\tau_{i,j}$ is reduced by some amount. As shown in Fig. 1b, τ_1 now completes its execution later because the upgrading of the processor lets $\tau_{2,2}$ start before τ_1 arrives.

Now let us consider the same task executions with the exclusive resource access on R being replaced with that on an I/O device (also referred to as a resource in this paper). The schedule, as shown in Fig. 2a, is similar to its counter-part in Fig. 1a, except that the durations of $\tau_{1,2}$ and $\tau_{2,2}$ are longer because of I/O access. Fig. 2b shows the revised schedule after the upgrading of the processor. Similar to the previous discussions, τ_1 also completes later in a system with a more powerful processor even though it has a higher priority. In addition, the delay in τ_1's execution is more significant because of I/O access (even though parallelism is observed in this example).

Considerations of the scheduling anomaly should not be based simply on the fact that processor speed nearly doubles in every 18 months. Instead, we should consider the variety of embedded system products and their platforms when softwares are ported among platforms/systems with different processor/device speeds. Technical problems should be on the reasons behind the anomaly and the way to avoid it. It underlies the motivation of this research.

2.2. Problem formulation

We first define some terminology and then formally define the scheduling problem.

Definition 2.1. Passive Resources: A resource is passive if the resource requires the consumption of the processor power during its the access.

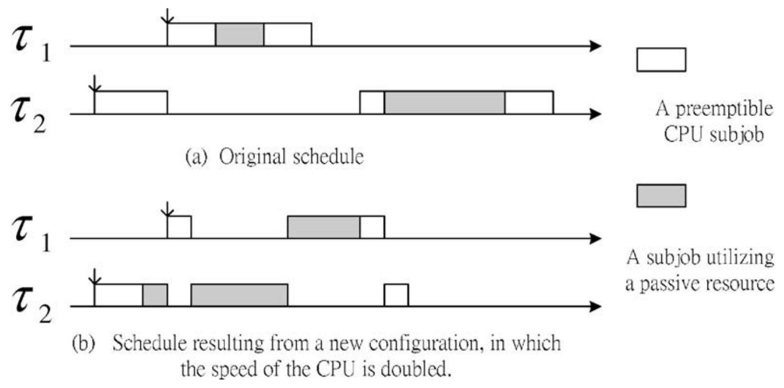

τ_1
τ_2
(a) Original schedule
τ_1
τ_2
(b) Schedule resulting from a new configuration, in which the speed of the CPU is doubled.
A preemptible CPU subjob
A subjob utilizing a passive resource

Fig. 1. An anomaly of task executions due to hardware upgradings.

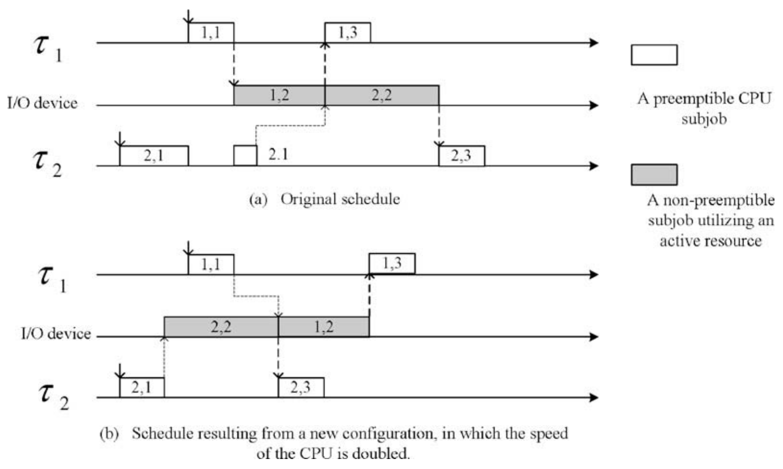

τ_1
1,1
1,3
I/O device
1,2
2,2
τ_2
2,1
2.1
2,3
(a) Original schedule
τ_1
1,1
1,3
I/O device
2,2
1,2
τ_2
2,1
2,3
(b) Schedule resulting from a new configuration, in which the speed of the CPU is doubled.
A preemptible CPU subjob
A non-preemptible subjob utilizing an active resource

Fig. 2. An anomaly of task executions over a uniprocessor system with an I/O device.

Good examples of passive resources include semaphores, mutex locks, event objects, and database locks. Passive resources could be accessed without locks or with exclusive or shared locks, depending on the characteristics of the resources and application logics. A resource is *active* if it is not passive. Good examples of active resources include disks, printers, network adaptors, and transceivers. A task might issue a request on an active resource and resume its execution if the request is asynchronous and granted. If the request is synchronous, then the task is suspended until the request is fulfilled. In this paper, we are interested in non-preemptible active resources, such as disks, with synchronous requests. We assume an access request to be serviced immediately on the corresponding active resource once it is granted and available, and we do not consider I/O buffering or caching.

We denote the *j*th invocation of task τ_i as $J_{i,j}$ (referred to as a *job*), where a task is periodic. A job $J_{i,j}$ comprises a sequence of sub-jobs to be executed in order. For example, a sub-job can be a subroutine of an application. $J_{i,j,k}$ denotes the *k*th sub-job of $J_{i,j}$. Let $C = \{(\mathrm{CPU}, \mathrm{Speed}_{\mathrm{CPU}}), (\mathrm{Dev}_1, \mathrm{Speed}_{\mathrm{Dev}_1}), \cdots, (\mathrm{Dev}_n, \mathrm{Speed}_{\mathrm{Dev}_n})\}$ denote a system configuration, in which the processor runs at a speed $\mathrm{Speed}_{\mathrm{CPU}}$, and each active resource Dev_i operates at a speed $\mathrm{Speed}_{\mathrm{Dev}_i}$. A system configuration $C' = \{(\mathrm{CPU}, \mathrm{Speed}'_{\mathrm{CPU}}), (\mathrm{Dev}_1, \mathrm{Speed}'_{\mathrm{Dev}_1}), \cdots\cdots, (\mathrm{Dev}_n, \mathrm{Speed}'_{\mathrm{Dev}_n})\}$ is greater than or equal to C (denoted as $C' \geqslant C$) if and only if $\mathrm{Speed}'_{\mathrm{CPU}}$ is no less than $\mathrm{Speed}_{\mathrm{CPU}}$, and each $\mathrm{Speed}'_{\mathrm{Dev}_i}$ of C' is no less than the corresponding $\mathrm{Speed}_{\mathrm{Dev}_i}$ of C. We assume that the time required to complete any sub-job J over C' is no more than that of J over C if $C' \geqslant C$.

Given a set T of jobs in an n-task set $\{\tau_1, \tau_2, \ldots, \tau_n\}$ between a given interval P, let Π be a given real-time scheduler, and $S = \Pi_C(T)$ be the schedule resulting from the scheduling of jobs in T by the scheduler Π based on a given system configuration C. Note that jobs in T have their arrival times fixed for a given T. Let $\theta^C_\Pi(J_{i,j,k})$ denote the completion-time of $J_{i,j,k}$ under a scheduler Π based on a system configuration C. We say that a scheduler is *stable* according to the following definition:

Definition 2.2. Stability: A real-time scheduler Π is stable if and only if $\forall_{i,j,k}, \theta^C_\Pi(J_{i,j,k}) \geqslant \theta^{C'}_\Pi(J_{i,j,k})$ for any task set T and any two system configurations C and C' when $C' \geqslant C$.

A real-time scheduler is *unstable* if it is not stable. An unstable scheduler may or may not result in an anomaly for any two given system configurations C and C' and a given task set T. An *anomaly* occurs for a real-time scheduler Π with two given system configurations C and C' and a given task set T if and only if $\exists_{i,j,k}, \theta^{C'}_\Pi(J_{i,j,k}) > \theta^C_\Pi(J_{i,j,k})$ when $C' \geqslant C$. The objective of this paper is to explore the stability property of real-time schedulers.

In the rest of this paper, a scheduling algorithm (e.g., rate-monotonic scheduling (RM) (Liu and Layland, 1973) or earliest-deadline-first scheduling (EDF) (Liu and Layland, 1973)) with a resource synchronization protocol (e.g., non-preemptible critical section protocol (NCSP) (Mok, 1993), priority-ceiling protocol (PCP) (Sha et al., 1990), or stack-resource policy (SRP) (Baker, 1990) are referred to as a *scheduler*. For example, a scheduler can be RM with NCSP. A set of anomaly-prevention *rules* to be proposed are to revise the protocol of a scheduler for anomaly prevention.

3. Scheduler stability – greediness versus stability

It was shown in Mok (2000) that there does not exist a totally on-line optimal scheduler in the presence of non-preemptive resource access, where a *totally on-line scheduler* makes a scheduling decision without the knowledge of the future arrivals of tasks and their timing constraints. Schedulers considered in this case are greedy, and a *greedy scheduler* always grants a resource request among the pending requests if the resource is available. A scheduler is greedy if and only if both its scheduling algorithm and its resource synchronization protocol are greedy. For example, the scheduler of rate-monotonic scheduling with NCSP is greedy. Our discussions first begin with the stability of greedy schedulers because they are widely adopted in many real-time operating systems. We shall first show that no greedy scheduler is stable in the presence of non-preemptive resources and then explore anomaly-free resource synchronization issues.

Theorem 1. *No greedy real-time scheduler is stable in the presence of non-preemptive resources.*

Proof. The correctness of this theorem can be proved by examples similar to those in Figs. 1 and 2: Consider the executions of two tasks τ_1 and τ_2 with passive resource sharing on R (Please see Fig. 1), where τ_2 has a lower priority but arrives earlier than τ_1. Let τ_1 arrive before τ_2 locks R non-preemptively in a given system configuration C. Now suppose that τ_1 arrives after τ_2 attempts to lock R in another system configuration C', where $C' > C$. Since R is available, any greedy scheduler would grant the request of τ_2 on R. As a result, τ_1 would be blocked by τ_2 for its request to R. Similar to the condition in Fig. 1, the completion-time of τ_1 would be delayed. The correctness of this theorem on active resource sharing could be proved in a way similar to that in Fig. 2. □

How to avoid anomaly could be a difficult problem in system implementations unless proper constraints are given for the domain of interest. The discussions of greedy schedulers are to show that they are vulnerable to anomalies. That no greedy schedulers can be stable has been proven in Theorem 1. On the other hand, system upgrades are not beneficial if the adopted scheduler is static. This work considers anomaly-prevention rules for greedy schedulers to consider not only anomaly prevention but also resource utilization. Contrast to a totally clairvoyant scheduler, the technical issue is how to conduct anomaly prevention with limited future knowledge. *In this paper, we focus our discussion on the stability of real-time schedulers in adopting a more powerful system configuration C', with respect to a given baseline system configuration C* (i.e., $C' \geqslant C$). We first present a necessary condition for the stability of a real-time scheduler with respect to a system configuration C and then explore the stability issues of existing resource synchronization protocols in later sections.

Theorem 2. *Given a system configuration C, a real-time scheduler Π is unstable with respect to C only if there exists a sub-job in some given task set T that experiences a preemption or blocking in $S' = \Pi_{C'}(T)$ but not in $S = \Pi_C(T)$, where C' is another system configuration, and $C' \geqslant C$.*

Proof. Suppose that there exists an anomaly for some sub-job $J_{i,j,k}$ in T, i.e., $\theta^{C'}_\Pi(J_{i,j,k}) > \theta^C_\Pi(J_{i,j,k})$. The occurrence of the anomaly is due to one of the following two cases: (1) $J_{i,j,k}$'s beginning time is delayed in S' compared to that in S, because $J_{i,j,k}$ is blocked in S' due to resource contention. (2) $J_{i,j,k}$ is preempted by some higher-priority sub-job in S', and the preemption does not happen in S. □

Theorem 2 could serve as a necessary condition to retain the stability of a given real-time scheduler with respect to a system configuration. We can always prevent the occurrence of any anomaly by negating the necessary condition. The technical question, however, is how to maintain the preemption and blocking relationship in a given system configuration when a more powerful system configuration is adopted.

4. Anomaly prevention

In this section, anomaly-prevention rules are introduced. How the proposed anomaly-prevention rules can be integrated into a

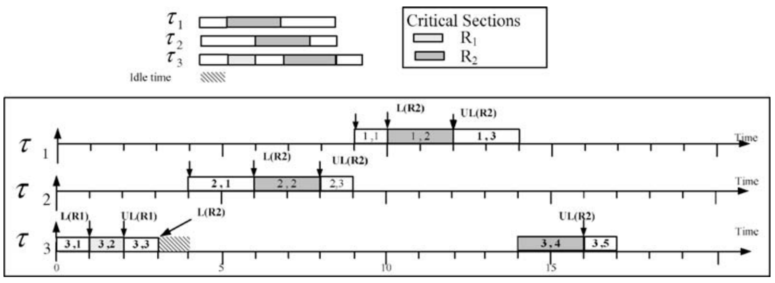


Fig. 3. An NCSP-IDI schedule.

greedy scheduler of rate-monotonic scheduling algorithm with non-preemptible critical section protocol (NCSP) (Mok, 1993) is demonstrated.[1]

4.1. Anomaly prevention – passive resources

In this section, we first discuss the handling of passive resources and present two naive rules for anomaly prevention.

4.1.1. Insertions of idle time

Given the arrival times of tasks and the duration time of each resource access, an existing protocol could be revised and be stable based on the following rule: *Whenever a task τ_L requests a passive resource R at time t and it might access R for D time units, the request is held pending if $(t + D)$ is larger than the arrival time of some higher-priority task τ_H*. We refer to this as the *idle-time-insertion* (IDI) rule. Note that when only a fixed set of periodic tasks is considered, the arrival time of each task could be derived based on its period and initial phase.

We shall revise the well-known non-preemptible critical section protocol (NCSP) (Mok, 1993) according to the IDI rule (referred to as NCSP-IDI) to better illustrate the concept of the IDI rule, where NCSP is combined with a priority-driven scheduling algorithm, except that no preemption/context-switch is allowed in a critical section. Consider the executions of three periodic tasks, τ_1, τ_2 and τ_3, where the period of τ_1, τ_2, and τ_3 are 20, 25, and 35, respectively. Let the two passive resources R_1 and R_2 be shared among the three tasks, where τ_1 has the highest priority, and τ_3 has the lowest priority. Fig. 3 shows their executions under NCSP-IDI. Suppose that τ_1, τ_2, and τ_3 arrive at time 9, time 4, and time 0, respectively. The request for R_2 by τ_3 is held pending at time 3 because the corresponding access could not be completed before time 4 (referred to as $\tau_{3,4}$), i.e., the arrival time of τ_2. As a result, the processor is idle from time 3 to time 4. The request of R_2 by τ_2 is granted at time 6 because the corresponding access would be completed before time 9 (referred to as $\tau_{2,2}$), i.e., the arrival time of τ_1. At time 9, τ_2 completes its execution, and τ_1 is dispatched. When τ_1 completes its execution at time 14, τ_3 resumes its execution, and its pending request is granted because the corresponding access would be completed before time 16, where the next arrival time of higher-priority tasks is 29.

As shown in the previous example, the definition of NCSP remains the same, except that a resource request could be pending when the request might result in blocking any higher-priority task. Clearly, the scheduling decision could be done very efficiently. The time complexity of the revised NCSP protocol is O(n), where n is the number of tasks in a fixed set of periodic tasks. However, such a rule is very restrictive since lower-priority tasks could be repeatedly delayed due to pending requests. Given a taskset $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$, by letting a task be idle if any higher-priority tasks is currently idle, the extra time overheads imposed on task τ_i by using the IDI rule for anomaly prevention can be simply computed as:

$$l_i = \sum_{k=1}^{i} \left(s_k \sum_{j<k} \left\lceil \frac{p_k}{p_j} \right\rceil \right)$$

where s_k is the duration of the longest critical section of tasks τ_k.

When task set T is scheduled by rate-monotonic scheduling with IDI, a sufficient condition of the schedulability of task set T is:

$$\forall \tau_i \in T, \sum_{j \neq i} \frac{c_j}{p_j} + \frac{c_i + l_i}{p_i} \leqslant i \cdot (2^{i-1} - 1)$$

The schedulability test takes into consideration all the potential overheads of idle time insertion for anomaly prevention. An admitted task set can suffer from neither deadline violations nor anomalies.

Consider a task set T, a system configuration C, and the upgraded system configuration C'. Let a preemption or a blocking be *new* if it exists in $S' = \Pi_{C'}(T)$ but not in $S = \Pi_C(T)$. With the IDI rule, anomaly prevention is enforced because (1) there are no new preemptions and (2) in both S and S' a task is never blocked by a low-priority task.

Consider a low-priority task in $S = \Pi_C(T)$ that is preempted by a high-priority task. In $S' = \Pi_{C'}(T)$, the low-priority task may or may not be preempted by the high-priority task, because the low-priority task's execution time in $S' = \Pi_{C'}(T)$ is reduced. However, the low-priority task in $S' = \Pi_{C'}(T)$ may lock a resource earlier than it does in $S = \Pi_C(T)$. Without the IDI rule, in $S' = \Pi_{C'}(T)$, the low-priority task may introduce a new blocking to the high-priority task. Because of this new blocking, the high-priority task in $S' = \Pi_{C'}(T)$ may complete later than it does in $S = \Pi_C(T)$, and it may experience new preemptions. However, the IDI rule avoids any blocking, so the new preemptions are impossible. On the other hand, a high-priority task never introduces new preemptions, because it preempts low-priority tasks only on its arrivals (i.e., task periods), which are independent of system configurations.

4.1.2. Preservation of access order

The idle-time-insertion rule is conservative in task synchronization. Another naive rule is to maintain the resource access order of resources in $S = \Pi_C(T)$ when another system configuration C' is given. We could avoid any anomaly in task scheduling by referencing either a pre-run schedule or an on-line generated schedule $S = \Pi_C(T)$.

Given a schedule $S = \Pi_C(T)$ for a system configuration C (generated either in an on-line or off-line fashion), we can order all of the resource requests in S according to their granted time, and they will be sequentially ordered. We define the *order-preservation*

[1] The scheduler is sufficiently simple to demonstrate how the anomaly-prevention rules can be applied. Note that, since the correctness of the proposed anomaly-prevention rules is independent of any resource synchronization protocols, the rules are applicable to other schedulers, such as a rate-monotonic scheduling algorithm with PCP (Sha et al., 1990) or SRP (Baker, 1990).

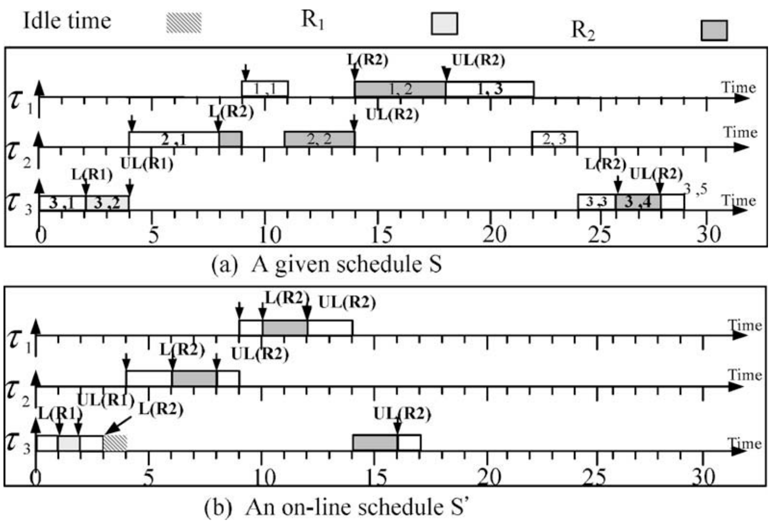


Fig. 4. An NCSP-OP schedule.

(OP) rule as follows: *The granting of resource requests from tasks executing over another system configuration, C', must be consistent with their corresponding granting order in S*. That is, a resource request R will be pending for a system configuration C' if there exists some resource request that is granted before R in the referenced schedule S but is not yet granted in S'.[2] When there is no ambiguity, we use the symbol of a sub-job to denote the corresponding resource request made in the beginning of the sub-job for the rest of this paper.

NCSP can be revised with the OP rule (referred to as NCSP-OP) to illustrate this concept: Consider the executions of the same task set, denoted as T, in Fig. 3, except that the execution time of each task is doubled. Fig. 4a shows a schedule $S = \Pi_C(T)$ of the task set under NCSP for a given system configuration C. The granting order of resource requests in S is $O_C = (\tau_{3,2}, \tau_{2,2}, \tau_{1,2}, \tau_{3,4})$. Suppose that we run T over another system configuration C', where the computing power of C' is twice that of C. Fig. 4b shows the resulting schedule $S' = \Pi_{C'}(T)$ over C'. At time 1, the request on R_1 issued by τ_3 is granted, since it is the first granted request in O_C. At time 3, the request on R_2 issued by τ_3 will be pending because the granting of the request will violate the order in O_C. The request on R_2 issued by τ_2 is granted at time 6, and the request on R_2 by τ_1 is then granted at time 10. When τ_1 completes at time 14, τ_3's request on R_2 is granted at time 14, and that is the last granted request in O_C.

As mentioned above, the OP rule can be conducted by referencing either an off-line generated pre-run schedule or an on-line emulated schedule. The analysis of time and space complexity of the two alternatives are as follows:

The space complexity of generating a pre-run schedule is $O(\sum_{i=1}^{n} n_i * (\text{LCM}/p_i) * m)$, where n_i and p_i are the numbers of sub-tasks and the period of task τ_i, respectively. LCM and m are the hyper-period of the task set and the total number of passive resources in the system, respectively. A pre-schedule contains a complete schedule within a hyper-period with respect to the original system configuration C. With the access order in the pre-run schedule, to check whether a request to a resource can be granted takes constant time (i.e., O(1) time).

At any time instant, each task would have no more than one pending resource request. In other words, to enforce the access-order rule, at any time instant, with respect to system configuration C', we need to maintain the original precedence of no more than n pending requests in the access order of resources in $S = \Pi_C(T)$. This can be accomplished by emulating schedule $S = \Pi_C(T)$ to see which one among the n currently pending requests should be granted first. The schedule $S = \Pi_C(T)$ is progressively emulated on the arrival of a task, and it takes $O(\sum_{i=1}^{n} n_i * m)$ time to emulate how the newly arriving task is scheduled in $S = \Pi_C(T)$. With on-line emulation, to grant one among the n pending requests takes no more than $O(n)$ time.

The correctness of the OP rule is based on Theorem 2. We shall show that the OP rule allows no extra blocking and preemption in $S' = \Pi_{C'}(T)$ with respect to $S = \Pi_C(T)$. We shall first show that no extra blocking in $S' = \Pi_{C'}(T)$ with respect to $S = \Pi_C(T)$ can occur. Let r_h and r_l be requests to a passive resource R from a high-priority task and a low-priority task, respectively. In $S = \Pi_C(T)$, there can be two cases: r_h precedes r_l or r_l precedes r_h. For the case of r_h preceding r_l, r_l naturally is never "blocked" by r_h because r_h is from a high-priority task. Furthermore, as long as the OP rule is enforced, r_l can never be granted before r_h and no new blocking exists. For the case of r_l preceding r_h, if r_h is blocked by r_l in $S' = \Pi_{C'}(T)$, then r_h is also blocked by r_l in $S = \Pi_C(T)$. That is because the low-priority task has successfully locked resource R before the arrival of the high-priority task and thus r_h is blocked by r_l. Since the arrivals of tasks (i.e., task periods) are not affected by upgrades to system configurations, the low-priority task always successfully locks resource R before the high-priority task. As to extra preemption, the reason that the OP rule introduces no extra preemption is similar to that of the IDI rule. The only difference is that a low-priority task will not block a high-priority task if the blocking is not in $S = \Pi_C(T)$.

4.2. Anomaly prevention – active and passive resources

The purpose of this section is to extend the idea in anomaly prevention to the handling of active resources and a more flexible way in the bounding of the anomaly duration.

4.2.1. IDI and OP rules – active resources

This section revisits Theorem 2 on anomaly prevention in terms of the IDI and OP rules when active non-preemptible resources are considered. We shall first show that the speedup of active resources could introduce new preemptions and might result in anomaly. We will then address the revision of the IDI and OP rules.

[2] When an off-line schedule is used, the job arrival times of each task must be known and fixed. The duration time of each resource access must also be known and fixed.

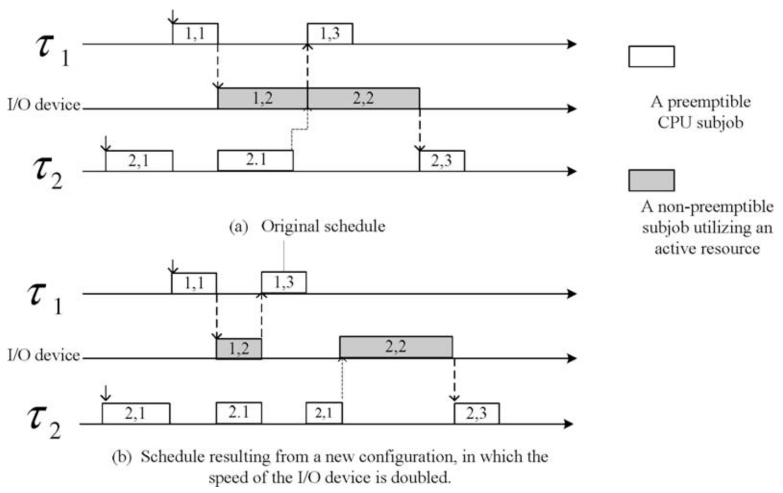

τ 1
1,1
1,3
I/O device
1,2
2,2
τ 2
2,1
2,1
2,3
(a) Original schedule
1,1
1,3
1,2
2,2
2,1
2.1
2,1
2,3
(b) Schedule resulting from a new configuration, in which the speed of the I/O device is doubled.
A preemptible CPU subjob
A non-preemptible subjob utilizing an active resource

Fig. 5. Scheduling anomaly involving with I/O device access.

Fig. 5a shows the executions of tasks for a system configuration *C*, as shown in Fig. 2a. Suppose that the performance of the active resource (AR) is improved such that $\tau_{1,2}$ is completed earlier, and $\tau_{1,3}$ preempts $\tau_{2,1}$, as shown in 5b. As a result, the completion-time of $\tau_{2,1}$ is delayed because the parallelism of task executions is reduced.

We can extend the OP rule by preserving the preemption relationship of sub-jobs of tasks. Similar to the informal proof in Section 4.1.2, we could show that any priority-driven scheduler revised with the new OP rule is stable. However, the IDI rule could not be extended in a similar way because active resources would introduce new task preemptions, where the rationale behind the IDI rule is highly dependent on the predictablity of preemption times, which are only due to the arrivals of higher-priority tasks in Section 4.1.1. Note that new preemptions caused by the speedup of active resources would simply result in the delay of the completion-time of lower-priority tasks. One way to avoid this in the application of the IDI rule is to avoid such preemptions when the related active resource access is completed too early, compared to that in the original system configuration.

4.2.2. Anomaly with a bounded duration time

This section proposes rules to limit bound the number of occurrences of scheduling anomalies and, thus, the anomaly duration for a task. The main idea is to manage the number of priority inversions per task such that the occurrences of anomaly is under control. We first use the well-known priority ceiling protocol (PCP) (Sha et al., 1990) and NCSP as examples to illustrate the priority inversion problem due to active-resource access.

Under PCP, each resource is given a ceiling equal to the maximum priority of tasks that might access the resource, and a resource request of a task is granted if the task priority is higher than the maximum ceiling of resources locked by other tasks. When active and passive resources are considered together in the enforcement of the ceiling rule, PCP guarantees that the maximum number of priority inversions for each task is one.[3] Task scheduling under NCSP is similar to that under PCP. Any task scheduled under NCSP is guaranteed one priority inversion in the worse case because any access to an active or passive resource is in a critical section, and no context switch is allowed. The price paid for the one priority inversion in the above two cases is the reduction of the execution parallelism. Now suppose that active and passive resources are separately considered in task synchronization. That is, the granting of a request on a passive/active resource considers only the maximum ceiling of passive/active resources locked by other tasks. The maximum number of priority inversions for each task under PCP becomes one plus the number of accesses to active resources from the task. This is because a higher-priority task might suffer from one extra priority inversion whenever it resumes from the access of an active resource. Better parallelism is achieved but with the cost of priority inversion. A similar phenomenon could also be observed for tasks scheduled under NCSP. As astute readers might point out, with a larger maximum number of priority inversions, there can be more scheduling anomalies. Such observations underlie the motivation of the following rule in anomaly management, referred to as the *anomaly control* (AC) rule: We consider only fixed-priority assignment in task scheduling in this section.

Let each table entry AC[i] *denote the number of blocking tolerable to a task* τ_i. (1) *Whenever a job of* τ_i *completes its execution*, AC[i] *is reset to the initial setting*. (2) *Whenever a blocking occurs to* τ_i, AC[i] *is decremented by one*. (3) *Any request to a resource from* τ_i *is blocked if any* AC[j] *of some higher-priority task is no more than* 0. (4) *Active resources should be managed independently of passive resources*.

Consider the executions of three tasks τ_H, τ_M and τ_L being scheduled by NCSP revised with the AC rule (referred to as NCSP-AC), where τ_H is the highest-priority task, and τ_L is the lowest-priority task. As shown in Fig. 6, AC[τ_i] for each task τ_i is initially set as 2. τ_M arrives at time 0 and is granted a request on an active resource (AR) at time 1 because AC[τ_H] > 0. When τ_H that arrives at time 1 and makes a request to AR at time 2, the request is blocked because AR is accessed non-preemptively now (or some critical section in accessing an active resource is active now). As a result, AC[τ_H] is decremented by 1. When τ_L arrives at time 2, it starts its execution because τ_H is blocked on a pending request on AR, and τ_M is waiting for the completion of the access on AR. The lock request by τ_L on a passive resource R_1 is granted at time 3 because no task is yet in a critical section accessing a passive resource (the separated consideration of active and passive resources), and AC[τ_M], AC[τ_H] > 0. At time 4, τ_M completes its access of AR, and the request of τ_H on AR is granted. But τ_M could not resume at time 4 because τ_L is executing in a critical section, AC[τ_M] is decremented by 1 because of the blocking. When τ_L unlocks R_1 and leaves its critical section at time 5, τ_M resumes. τ_M then enters a

[3] This claim holds only if a pending request of a lower-priority task on an active resource could not be granted at the time moment when a higher-priority task resumes its execution from an access on the active resource. Otherwise, the maximum number of priority inversions for each task is one plus the number of active resource access by the task. The statement is true even when active and passive resources are considered together in ceiling enforcement.

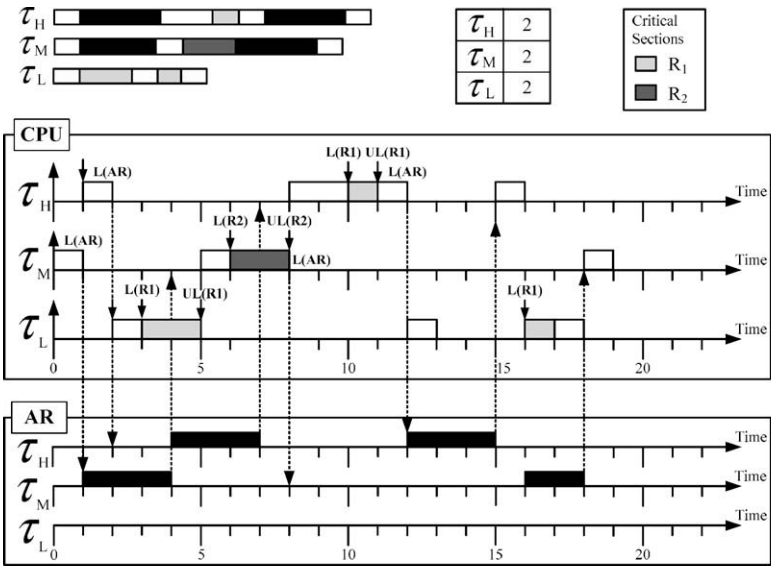


Fig. 6. An NCSP-AC schedule.

critical section by successfully locking R_2 (where $\mathrm{AC}[\tau_H] > 0$) at time 6 and later blocks τ_H a time 7 when τ_H completes its access on AR. $\mathrm{AC}[\tau_H]$ becomes 0 at time 7 and forbids τ_M and τ_L from locking any active or passive resource again until its completion.

As readers might point out, the maximum number of priority inversions for a task, indeed, implies the maximum occurrence number of scheduling anomalies (with respective to a system configuration with the worst-case consideration). We conclude that the maximum number of scheduling anomalies for a task τ_i under NCSP revised with the AC rule is the maximum number of priority inversions, i.e., the initial value of AC[i]. With a setting of AC[], the maximum duration of an anomaly is thus bounded, given the maximum access duration of active/passive resources.

5. Performance evaluation

5.1. Experimental setup and performance metrics

In this section, the proposed anomaly-prevention rules IDI together with OP and the proposed anomaly-bounding rule AC were evaluated. A scheduler of rate-monotonic scheduling with NCSP is adopted as the baseline scheduler. NCSP-IDI, NCSP-OP, and NCSP-AC refer to the baseline schedulers revised by the IDI rule, the OP rule, and the AC rule, respectively. Two schedulers, rate-monotonic scheduling with priority ceiling protocol (referred as PCP) (Sha et al., 1990) and rate-monotonic scheduling with stack resource policy (referred as SRP) (Baker, 1990), were used for performance comparisons. All the schedulers were evaluated for the anomaly problem and the speedup in the completion-time of each task.

The primary metric was the number of anomaly occurrences over the total number of task instances, referred to as the *anomaly ratio*. Let X and Y be the number of anomaly occurrences and the total number of task instances in an experiment, respectively. The anomaly ratio was defined as X/Y. The second performance metric was the *completion ratio*, which stands for the ratio of the total number of fulfilled task instances (i.e., task instances complete before their deadlines) to the total number of all task instances. Note that all task sets in the experiments are randomly generated without testing their schedulability in advance. In each experiment, a baseline system configuration C was adopted for comparisons. Let ct and ct′ denote the completion times of a task instance scheduled by an experimented scheduler under the baseline system configuration and a given system configuration C', respectively. Suppose that the task instance was ready at time t. The *completion-time ratio* for the task instance under a given scheduler and C' was defined as $\frac{ct'-t}{ct-t}$. The completion-time ratio for a given scheduler under C' was defined as the average completion-time ratio of all task instances scheduled by the scheduler in the experiment.

In the experiments, the number of tasks per task set ranged from 5 to 20 and the number was randomly picked for each experiment. Each task was periodic, and the deadline of a task instance was equal to the arrival time of its following task instance. The number of fundamental frequencies for a task set in our experiments ranged from 2 to 4 and was randomly assigned to tasks (Kamenoff and Weiderman, 1991; Kim et al., 1996; Locke et al., 1991; Molini et al., 1990; Kuo et al., 2003). Fundamental frequencies are the common factors of all the task periods generated for experiments. For example, the two fundamental frequencies of periods 5, 15, 45 are 3 and 5. Let each task τ in the experiments be denoted as $\tau = (c^{\mathrm{CPU}} \cdot \epsilon, c^{\mathrm{active}} \cdot \epsilon, p \cdot \epsilon)$, where $c^{\mathrm{CPU}} \cdot \epsilon$, $c^{\mathrm{active}} \cdot \epsilon$, and $p \cdot \epsilon$ were the CPU computation time, the active-resource execution time, and the period of τ, respectively. Since ϵ was used in the experiments as the baseline time unit, ϵ would be omitted for the simplicity of presentation when temporal parameters were presented in the rest of this paper (e.g., τ would be abbreviated as $(c^{\mathrm{CPU}}, c^{\mathrm{active}}, p)$). Note that ϵ could be any positive constant time interval, e.g., 1 ms. The periods of tasks were randomly picked between 150 and 3000 time units under the constraints for the selected number of fundamental frequencies. The CPU utilization of each task ranged from 5% to 30%. When an active resource was adopted in the experiment, the utilization of the active resource for the task was between 20% and 40% of its CPU utilization. The number of passive resources for each task set ranged from 3 to 6 and was randomly picked. Which task would use which passive resource was determined by a random distribution function. However, a task that might consume more CPU time tended to use more passive resources. In the experiments, 1000 task sets was generated for each system configuration. Each run of the simulation for a task set was done for the least common multiple of all of the task periods. Each experiment was given a system configuration $C = \{(\mathrm{CPU}, \mathrm{Speed_{CPU}}), (\mathrm{Dev}_1, \mathrm{Speed_{Dev1}})\}$, the system configuration $\{(\mathrm{CPU},1), (\mathrm{Dev}_1, 1)\}$ served as the baseline in the experiments. In other words, if a task instance $\tau = (c^{\mathrm{CPU}}, c^{\mathrm{active}}, p)$

needed $c \cdot \epsilon$ time units to complete its execution on the CPU for a configuration $\{(\text{CPU},1),(\text{Dev}_1,1)\}$, then τ needed $(c \cdot \epsilon)/k$ time units to be completed over the new CPU for a configuration $\{(\text{CPU},k),(\text{Dev}_1,1)\}$.

5.2. Experimental results

5.2.1. Systems with passive resources only

In this section, NCSP-IDI and NCSP-OP were evaluated in, comparison to PCP, SRP, and NCSP, where only passive resources were shared among tasks. The anomaly ratio, completion ratio, and completion-time ratio for the simulated schedulers were reported for comparisons.

Fig. 7 shows the anomaly ratio of all of the simulated schedulers. The *X*-axis denotes the subsets of task sets, where “First *X* Priority” stands for the first *X* priority tasks in the task set. The *Y*-axis denotes the anomaly ratio. As shown in Fig. 7, PCP, SRP, and NCSP all suffered from the anomaly problem. For example, each task instance under PCP might suffer from an average of 0.12 times of anomaly occurrences, when the CPU speed was doubled. Because of anomaly prevention, NCSP-IDI and NCSP-OP did not have any anomaly problem.

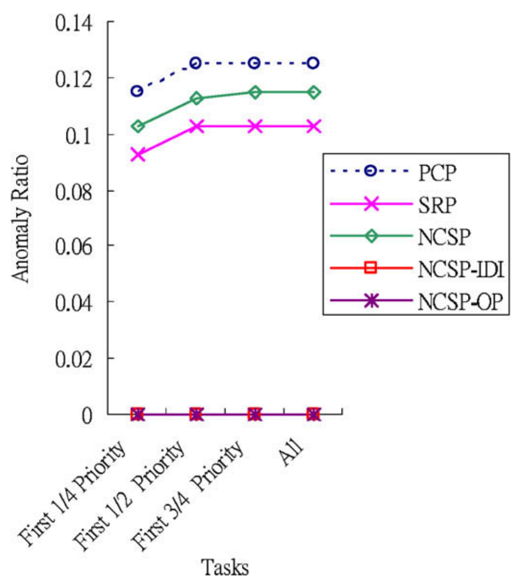
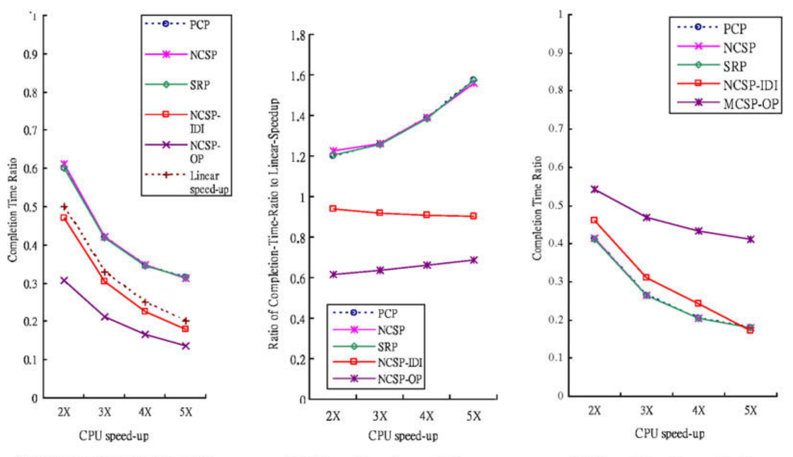


Fig. 7. Anomaly ratios for schedulers when only passive resources were shared among tasks.

Fig. 8 shows results for the completion-time ratios of the simulated schedulers, where the *X*-axis denotes the speedup ratio of the CPU, and the *Y*-axis denotes the completion-time ratios. That is, when the speedup ratio was 2*X*, the CPU speed was doubled. The first 1/4 priority tasks refers to a group of high-priority tasks in rate-monotonic scheduling. By separately observing the effects of anomaly prevention on high-priority tasks and all the tasks, it can compare the effects of anomaly prevention on tasks of different priorities. Fig. 8a shows the completion-time ratios of the first 1/4 priority tasks in the task set. NCSP-IDI and NCSP-OP provided fairly stable improvement over the completion-time of each task. Although PCP, SRP, and NCSP also seemed to provide stable improvement over the completion times of the first 1/4 priority tasks, Fig. 8b shows the results from a different perspective. In particular, it shows the ratio of the completion-time ratio to the inverse of CPU speedups (i.e., the linear-speedup curve in Fig. 8a) for the first 1/4 priority task in the task set. This provides a better view in observing the completion-time ratio when the CPU speedup varies. Note that the ratios of the completion-time ratio to the inverse of the CPU speedup for NCSP-IDI and NCSP-OP were almost the same for different increases of CPU speed. On the other hand, the ratios for SRP, PCP, and NCSP changed slightly and were higher than those for NCSP-IDI and NCSP-OP. Note that with smaller ratios, the completion-time ratio was better. NCSP-OP outperformed NCSP-IDI with respect to high-priority tasks, because NCSP-IDI pessimistically inserts idle time to avoid any blocking for anomaly prevention.

However, lower-priority tasks were sacrificed to permit the scheduling of higher-priority tasks under NCSP-OP. This is be shown by Fig. 8c, in which reported the completion-time ratios of all tasks. As shown in Fig. 8c, the completion-time ratios of NCSP-OP is higher than that of Fig. 8a. Fig. 8a shows the completion ratios of first 1/4 priority tasks, including that the response of lower-priority tasks was little improved under NCSP-OP as system configurations were upgraded. This was the price paid for the anomaly prevention and the benefits of higher-priority tasks. With the OP rule, although the completion-time ratio never gets worse, it does not linearly scale with the CPU speedup. One can consider applying the IDI rule together with the OP rule: A request for an available passive resource is immediately granted if the resource can be released before the arrival of any higher-priority tasks.



(a) Completion time ratio for the first ¼ priority tasks

(b) The ratios of completion-time-ratios to linear-speedup for the first 1/4 priority tasks

(c) Completion time ratio for all tasks

Fig. 8. Completion-time ratios for schedulers when only passive resources were shared among tasks.

Otherwise, the request should be granted to conform to the OP rule. Based on the proposed guidelines, smart heuristics can be proposed to better match the characteristics of schedulers and platforms. It should be noted that to optimally order the granting of resource accessing in terms of completion-time ratios is intractable (Mok, 2000). Most importantly, we must emphasize that NCSP-IDI and NCSP-OP were both free from any anomaly, which is our design objective.

Fig. 9 shows the completion ratios for the simulated schedulers, where the *X*-axis denotes the increased CPU speed, and the *Y*-axis denotes the completion ratios. Note that all the task sets in the experiments are randomly generated without testing their schedulability in advance. The completion ratios in Fig. 9 show how tasks of different priorities are benefited by upgrades to system configurations. With larger the completion ratios, the tasks benefit more from the upgrades. Fig. 9a shows that the completion ratios of the first 1/4 priority tasks was increased for each scheduler as the CPU was speeded up. NCSP-IDI showed the benefit in the favor of high-priority tasks since their completion ratios were the largest among those of others. However, anomaly-prevention schedulers tended to sacrifice the scheduling of lower-priority tasks, as shown in Fig. 9b. Note that the completion ratio improvement achieved by NCSP-OP did not seemed significant, which was because NCSP-OP postponed resource accesses issued by low-priority tasks to preserve orders in resource accessing.

5.2.2. Systems with passive resources and active resources

In this section, NCSP-AC was evaluated and compared against NCSP, where both active resources and passive resources were shared among tasks. As shown in Section 4.2.2, NCSP-AC would introduce parallelism to the executions over the CPU and active resources, while the number of priority inversions encountered by each task could still be controlled. In this part of the experiments, the completion ratios for the simulated schedulers were reported for comparison. Note that the number of priority inversions that could be encountered by each task instance was set between 3 and 5 as a parameter for NCSP-AC, and one active resource (e.g., Dev_1 in system configurations) was shared among tasks.

Fig. 10 shows the completion ratios for the simulated schedulers, where the *X*-axis denotes different system configurations. For

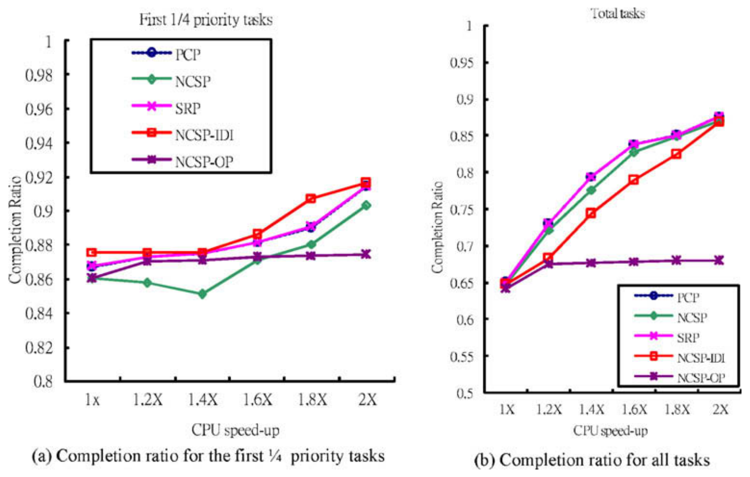


Fig. 9. Completion ratios for schedulers when only passive resources were shared among tasks.

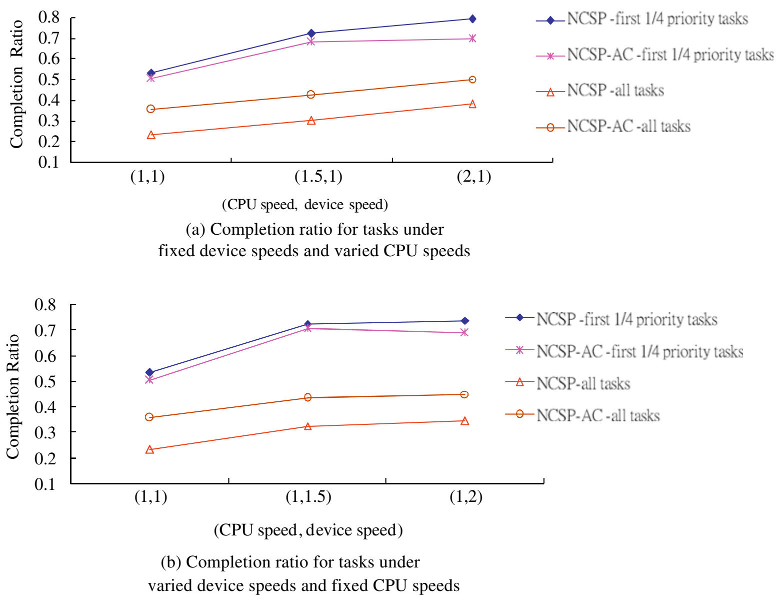


Fig. 10. Completion ratio for schedulers when both passive resources and active resources were shared among tasks.

example, (1,1.5) denotes a system configuration $\{(\text{CPU},1), (\text{Dev}_1, 1.5)\}$. The Y-axis denotes the completion ratios. Figs. 10a and b showed the completion ratios for different schedulers when the CPU or the active resource was upgraded, respectively. It can be seen that NCSP-AC greatly outperformed NCSP in terms of completion ratios when all tasks were considered, though NCSP was slightly better than NCSP-AC when only the first 1/4 priority tasks were considered. That was because NCSP-AC is intended to exchange a controllable number of priority inversions imposed on high-priority tasks for a higher degree of system parallelism. As may be noted, any upgrade to the active resource did not result in a significant improvement over the completion ratios, compared to upgrading the CPU. That was because the upgrading of the active resource might reduce potential parallelism in many cases.

6. Conclusion

As more and more software programs with timing constraints are ported among different platforms, the considerations of the timing behaviors of software become a critical issue. This research addresses important scheduling anomaly issues that are motivated by the practical needs in embedded-system implementations. We propose the concept of scheduler stability when system platforms change. We show that new violations of the timing constraints of tasks might occur even when a more powerful processor or device is adopted. Anomaly-prevention rules are first proposed to avoid scheduling anomalies in handling passive resources, such as semaphores. We then extend the idea to the management of active and passive resources. Finally, we present a rule to limit the number of maximum occurrences and the duration of scheduling anomalies for each task by an anomaly control rule based on a counting-table. A series of experiments was conducted to evaluate the capability of the proposed anomaly control rules, which present very encouraging results.

In future research, we will further explore scheduling anomalies under different system architectures, such as hyper-threading platforms and embedded I/O subsystems. Since the variety of embedded systems products has made software portability an important issue, more research in this direction should prove very rewarding.

References

Abeni, L., Buttazzo, G., 1998. Integrating multimedia applications in hard real-time systems. In: Proceedings of the IEEE Real Time System Symposium, pp. 4–13.

Abeni, L., Buttazzo, G., 1999. Qos guarantee using probabilistic deadlines. In: Proceedings of the Euromicro Conference on Real-Time Systems.

Baker, T.P., 1990. A stack-based resource allocation policy for real-time process. In: Proceedings of the IEEE Real-Time System Symposium.

Baker, T.P., 2003. Multiprocessor edf and deadline monotonic schedulability analysis. In: Proceedings of the IEEE Real-Time Systems Symposium.

Buttazzo, G.C., 2002. Scalable applications for energy-aware processors. In: Proceedings of the International Conference on Embedded Software.

Chen, D., Mok, A.K., Baruah, S.K., 2000. Scheduling distributed real-time tasks in the DGMF model. In: Proceedings of the IEEE Real Time Technology and Applications Symposium, pp. 14–22.

Chen, Y.-S., Chang, L.-P., Kuo, T.-W., Mok, A.K., 2005. Real-time task scheduling anomaly: observation and prevention. In: Proceedings of the ACM Symposium on Applied Computing.

Dhall, S.K., 1977. Scheduling Periodic-Time-Critical Jobs on Single Processor and Multiprocessor Computing Systems. Ph.D. thesis, University of Illinois, Urbana.

Graham, R.L., 1976. Computer and job-shop scheduling theory. In: Coffman, E.G., Bruno, J.L. (Eds.), Bounds on the Performance of Scheduling Algorithm. Wiley-Interscience, New York, pp. 165–227.

Jeffay, K., Goddard, S., 1999. A theory of rate-based execution. In: Proceedings of the IEEE Real-Time Systems Symposium, December, pp. 304–314.

Kamenoff, N.I., Weiderman, N.H., 1991. Hartstone distributed benchmark: requirements and definitions. In: Proceedings of the IEEE Real-Time Systems Symposium.

Kao, B., Garcia-Molina, H., 1997. Deadline assignment in a distributed soft real-time system. IEEE Transactions on Parallel and Distributed Systems 8 (12), 1268–1274.

Kim, N., Ryu, M., Hong, S., Saksena, M., Choi, C., Shin, H., 1996. Visual assessment of a real-time system design: a case study on a CNC controller. In: Proceedings of the IEEE Real-Time Systems Symposium.

Kuo, T.-W., Chang, L.-P., Liu, Y.-H., Lin, K.-J., 2003. Efficient on-line schedulability tests for real-time systems. IEEE Transaction on Software Engineering 29 (8).

Lin, K.J., Natarajan, S., Liu, J.W.-S., 1987. Imprecise results: utilizing partial computations in real-time systems. In: Proceedings of the IEEE Real-Time Systems Symposium, December, pp. 210–217.

Liu, C.L., Layland, J.W., 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM 20 (1), 46–61.

Liu, X., Goddard, S,. 2003. Resource sharing in an enhanced rate-based execution model. In: Proceedings of the Euromicro Conference on Real-Time Systems.

Locke, C.D., Vogel, D., Mesler, T., 1991. Building a predictable avionics platform in ada: a case study. In: Proceedings of the IEEE Real-Time Systems Symposium.

Manimaran, C., Siva Ram Murthy, G., 1997. Dynamic scheduling of parallelizable tasks and resource reclaiming in real-time multiprocessor systems. In: Proceedings of the High Performance Computing, December, pp. 18-21.

Mok, A., Chen, D., 1997. A multiframe model for real-time tasks. IEEE Transactions on Software Engineering 23 (10), 635–645.

Mok, A.K., 1993. Fundamental Design Problem of Distributed System for Hard-Real Time Environment. Ph.D. thesis, Department of Electrical Engineering and Computer Science, MIT, May.

Mok, A.K., 2000. Tracking real-time systems requirements. In: Proceedings of the Workshop on Modelling Software System Structures in a Fastly Moving Scenario.

Molini, J.J., Maimon, S.K., Watson, P.H., 1990. Real-time system scenarios. In: Proceedings of the IEEE Real-Time Systems Symposium.

Sha, L., Rajkumar, R., Lehoczky, J., 1990. Priority inheritance protocols: an approach to real-time synchronization. IEEE Transactions on Computers 39 (9).

Shen, C., Ramamritham, K., Stankovic, J., 1990. Resource reclaiming in real time. In: Proceedings of the IEEE Real-Time System Symposium.

Shen, C., Ramamritham, K., Stankovic, J., 1993. Resource reclaiming in multiprocessor real-time systems. IEEE Transactions on Parallel and Distributed Computing 4 (4), 382–397.

Spuri, M., Buttazzo, G., Sensini., 1995. Scheduling aperiodic tasks in dynamic scheduling environment. In: Proceedings of the IEEE Real-Time Systems Symposium.

Ya-Shu Chen joined Department of Electronic Engineering, National Taiwan University Science and Technology, at August 2007. She currently serves as an Assistant Professor. Ya-Shu Chen earned her BS degree in computer information and science at National Chiao-Tung University in 2001. Then, she studied in Department of Computer Science and Information Engineering, National Taiwan University, and was supervised by Prof. Tei-Wei Kuo. She successfully defended her master thesis and doctoral dissertation at 2003 and 2007, respectively. Her research interest includes operating systems, embedded storage systems, and hardware/software co-design.

Li-Pin Chang received BS degree in Computer Science from I-Shou University, and MS/Ph.D. degrees in Engineering from Department of Computer Science and Information Engineering at Taiwan University. At May 2005, he joined the faculty of Department of Computer Science, National Chiao-Tung University. He currently serves as Assistant Professor. He has served on the editorial board of Journal of Signal Processing Systems (Springer, SCI-E), and technical committee of international conferences including ACM Symposium on Applied Computing (SAC), IEEE Real-Time Systems Symposium (RTSS), IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), and IEEE/IFIP International Conference On Embedded and Ubiquitous Computing (EUC). Prof. Li-Pin Chang's research interest includes operating systems, real-time systems, and embedded storage systems.

Tei-Wei Kuo received the B.S.E. degree in Computer Science and Information Engineering from National Taiwan University in Taipei, Taiwan, in 1986. He received the M.S. and Ph.D. degrees in Computer Sciences from the University of Texas at Austin in 1990 and 1994, respectively. He is currently a Professor of the Department of Computer Science and Information Engineering, National Taiwan University. Starting from Feb 2006, he also serves as a Deputy Dean of the College of Electrical Engineering and Computer Science, National Taiwan University.

Prof. Kuo has served in the editorial board of many journals, including the Journal of Real-Time Systems (SCI) and IEEE Transactions on Industrial Informatics. He is also the General Chair of the IEEE Real-Time Systems Symposium (RTSS) in Barcelona, Spain, in 2008, and the Program Chair in Tucson, Arizona, USA, in 2007, where RTSS is the flagship conference in real-time systems. Since 2005, Prof. Kuo has served as the Steering Committee Chair of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), the best real-time systems conference in the Asian and Pacific Region. Prof. Kuo serves as an Executive Committee member of the IEEE Technical Committee on Real-Time Systems since 2005. He received the Ten Young Outstanding Persons Award of Taiwan in 2004 and a number of research awards, including the Distinguished Research Award from the National Science Council in 2003, the Distinguished

Teaching Award from the National Taiwan University in 2005 (Top 1%), and another two teaching awards from the National Taiwan University in 2003 and 2004 (top 10%). His research interests include embedded systems, real-time task scheduling, real-time operating systems, flash-memory storage systems, and real-time database systems. He has over 140 technical papers published or been accepted in international journals and conferences and more than 10 patents in USA and Taiwan on the designs of flash-memory storage systems.

Aloysius K. Mok received the BS degree in Electrical Engineering, the MS degree in Electrical Engineering and Computer science, and the PhD degree in Computer Science, all from the Massachusetts Institute of Technology. He is the Quincy Lee Centennial Professor in Computer Science at the University of Texas at Austin, where he has been a member of the faculty of the Department of Computer Sciences since 1983. He has performed extensive research on computer software systems and is internationally known for his work in real-time systems. He is a past chairman of the Technical Committee on Real-Time Systems of the IEEE and has served on numerous national and international research and advisory panels. His current interests include real-time and embedded systems, robust and secure network centric computing, and real-time knowledge-based systems. In 2002, Dr. Mok received the IEEE Technical Committee on Real-Time Systems Award for his outstanding technical contributions and leadership achievements in real-time systems. He is a member of the IEEE.