Real-Time Syst (2009) 42: 34–62
DOI 10.1007/s11241-009-9070-0

# A real-time configurable synchronization protocol for self-suspending process sets

**Ya-Shu Chen · Li-Pin Chang**

Published online: 12 March 2009


**Abstract** While several of researchers have proposed excellent protocols on resource synchronization, little work has been done for processes that might suspend themselves for I/O access, especially when they tend to be more tolerant to multiple priority inversions. This paper presents research results extended from the concept of priority ceilings with the objective of satisfying different priority-inversion requirements for different processes. We aim at practical considerations in which processes might voluntarily give up CPU and be willing to receive more blocking time than those in more traditional approaches. Extensions on the proposed scheduling protocols for deadlock prevention are also considered.

**Keywords** Real-time systems · Resource synchronization protocol · Priority ceiling · Deadlock prevention

## 1 Introduction

Real-time resource synchronization has been an important research topic over the past decades. Resolving resource contention with a proper management of priority inversion has usually been the main focus of the research. Among the many proposed synchronization protocols, the Priority Ceiling Protocol (PCP) (Sha et al. 1990) is one of the most well-known protocols in hard real-time task scheduling. It has been

This work is supported by a grant from the NSC program NSC97-2218-E-011-002.

Y.-S. Chen (✉)
Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, 106, ROC
e-mail: yschen@mail.ntust.edu.tw

L.-P. Chang
Department of Computer Sciences, National Chiao Tung University, Hsinchu, Taiwan, 300, ROC

proven that no higher-priority task could be blocked by more than one lower-priority task. The Stack Resource Policy (SRP) (Baker 1990) further extends PCP by allowing multiple units per resource and could adopt dynamic priority assignment algorithms, e.g., the Earliest Deadline First (EDF) algorithm (Liu and Layland 1973), where EDF assigns the task with the closest deadline among ready tasks with the highest priority.

Although many excellent resource synchronization protocols have been proposed, most are either for hard real-time task scheduling with the maximum priority inversion number being one, e.g., (Sha et al. 1990; Baker 1990; Chen and Lin 1990; Rajkumar et al. 1988), or for soft real-time task scheduling without any guarantee on the maximum priority inversion number, e.g. (Liang et al. 2003; Kuo et al. 2001; Han et al. 1996; Hsueh and Lin 1998; Kuo et al. 1999; Kuo et al. 2001; Liang et al. 2003; Sides 1995; Spuri et al. 1995; Stoica et al. 1996; Waldspurger 1995; Waldspurger and Weihl 1995; Wang and Lin 1994; Wu et al. 1999). Further, research on hard or soft real-time task scheduling often considers computation-intensive tasks only. When I/O operations are considered, research on real-time resource synchronization is often based on heuristics without any schedulability guarantee. Nevertheless, tasks executing over modern computer systems often consist of CPU and I/O bursts. While a task is pending on the completion of an I/O operation, the scheduler usually performs a context switch to execute another task. Resource synchronization which involves concurrent I/O operations and CPU executions is a difficult problem (Stankovic et al. 1995). Tasks which might suspend themselves voluntarily for I/O operations could suffer from a large number of priority inversions under many popular real-time resource synchronization protocols. This is because a lower-priority task might lock a resource that later blocks a suspended higher-priority task. Such a task model with I/O operations is not considered in many existing protocols. On the other hand, one priority inversion seems overly conservative for many applications. In reality, tasks could take different numbers of priority inversions, depending on their natures. These observations motivate this research.

This paper proposes configurable resource synchronization protocols for engineers to adjust the maximum number of priority inversions for each task and considers task suspension. A table-based approach is first proposed to adjust the maximum number of priority inversions for tasks without suspension. We then extend the approach to considerations of task suspension. The ceilings of resources become configurable to allow lower-priority tasks to grab resources while higher-priority tasks suspend themselves to wait for I/O operations. System utilization is traded with the maximum numbers of priority inversions for different tasks. A deadlock-prevention method with low run-time overheads is proposed to avoid system deadlocks. Although PCP is adopted to illustrate the configurable resource synchronization protocols, the idea could be extended to other synchronization protocols.

The rest of this paper is organized as follows: The motivation of this work is illustrated in Sect. 2 based on several examples, with the process mode considered in this paper and the necessary terminologies. A configurable resource synchronization protocol is proposed in Sect. 3. In Sect. 4, we present the idea of deadlock avoidance. An off-line heuristic to resolve deadlocks with its on-line manipulation is then proposed. Section 5 reports the performance evaluation of the proposed approach. This work is concluded in Sect. 6.

## 2 Motivation and problem definitions

The purpose of this section is to provide the motivation for this research. Observations of the behaviors of the well-known Priority Ceiling Protocol (PCP) (Sha et al. 1990) and Stack Resource Policy (SRP) (Baker 1990), when tasks might suspend themselves, are illustrated. We then define terminologies and definitions for this paper.

### 2.1 Motivation

We are interested in uniprocessor scheduling with I/O considerations, and every resource has only one instance. Let $\tau_H$, $\tau_M$, and $\tau_L$ be three tasks scheduled by a fixed-priority scheduling algorithm, where $\tau_H$ and $\tau_L$ are the highest-priority task and the lowest-priority task, respectively. Three resources $R_1$, $R_2$, and $R^a$ are shared among the tasks. $R_1$, $R_2$ are semaphores, and $R^a$ is an I/O device which could operate independently when a particular task is running over the CPU. Let operations on $R^a$ be non-preemptible. Suppose that $\tau_H$ might access $R_1$, $R_2$, and $R^a$, $\tau_M$ might access $R^a$, and $\tau_L$ might access $R_1$ and $R_2$. We use the following two scheduling examples to serve as the motivation for this research: Note that tasks could be periodic or aperiodic. We call each instance of a task as a *job*, where a task is a template of its jobs. For example, a periodic task has a corresponding job ready for each period.

Suppose that the task set is scheduled by PCP (Sha et al. 1990). The ceilings of $R1$ and $R_2$ are both equal to the priority of $\tau_H$, where the ceiling of a resource is equal to the maximum priority of all tasks that might access the resource. Since $R^a$ denotes an I/O device, there are two cases to be considered: (1) There is no resource locking needed for $R^a$ so any task could access $R^a$ whenever it is available. (2) A semaphore is defined for the access synchronization of $R^a$.

Consider the first case, where no ceiling rule is applied to $R^a$. Suppose that $\tau_H$, $\tau_M$ and $\tau_L$ are all ready at time 0. At time 1, $\tau_H$ successfully issues a request on $R^a$ and suspends to wait for the I/O completion. Let $\tau_M$ be dispatched at time 1 because it is the ready task with the highest priority. At time 2, $\tau_M$ issues an I/O request on $R^a$ and suspends its execution, where the suspension will continue until completing requests from $\tau_H$ and $\tau_M$ because $R^a$ is non-preemptive. Because $\tau_L$ is the only ready task, it is dispatched at time 2. At time 3, $\tau_L$ locks $R_1$ successfully because no other resource is currently locked. Note that no ceiling rule is applied to $R^a$. At time 4, $\tau_H$ resumes for execution because the request on $R^a$ is completed. As a result, the request from $\tau_M$ starts on $R^a$, and $\tau_H$ preempts $\tau_L$ at time 4. When $\tau_H$ issues a lock request on $R_1$ at time 5, $\tau_H$ is blocked by $\tau_L$ because the ceiling of $R_1$ (that is locked by $\tau_L$) is not lower than the priority of $\tau_H$. *It is the first priority inversion suffered by* $\tau_H$. $\tau_L$ inherits $\tau_H$'s priority and resumes its execution. At time 6, $\tau_L$ unlocks $R_1$, and $\tau_H$ preempts $\tau_L$ and locks $R_1$ successfully. $\tau_H$ unlocks $R_1$ at time 7. The I/O request on $R^a$ at time 8 is blocked because the request from $\tau_M$ is still under processing. *It is another priority inversion for* $\tau_H$. The blocking of a higher-priority task could happen not only on the executions over the CPU but also on the requests serviced by an I/O device.

We must point out that the blocking of $\tau_H$ occurs at time 5 because $\tau_H$ voluntarily surrenders the CPU at time 1 such that $\tau_L$ has a chance to execute and block $\tau_H$.

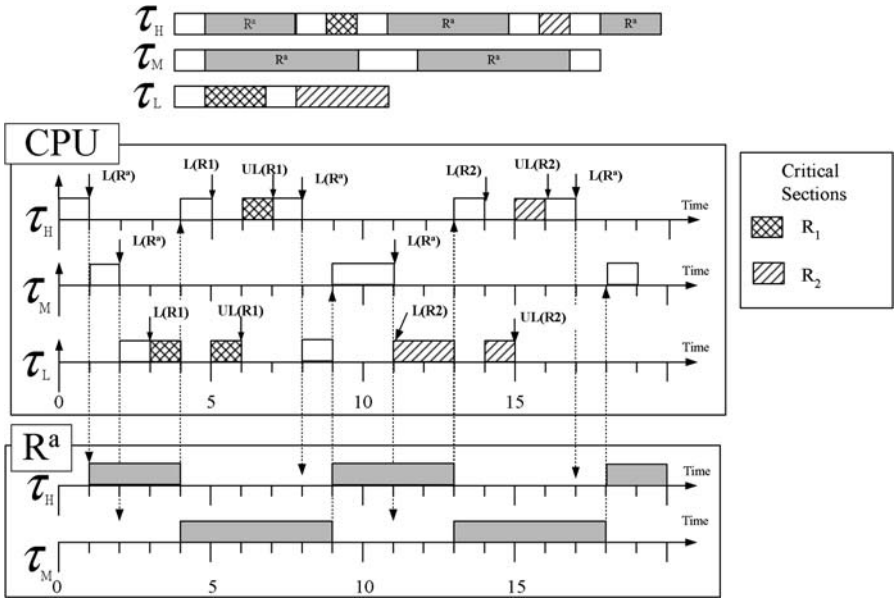



**Fig. 1** PCP not applied in $R^a$

Such a blocking could theoretically happen repeatedly without a bound (over CPU and I/O devices). For example, the I/O request of $\tau_H$ at time 8 causes the suspension of $\tau_H$ again such that $\tau_L$ successfully locks $R_2$ at time 11. Such a lock on $R_2$ let $\tau_L$ later block $\tau_H$ again at time 14, as shown in Fig. 1. We conclude that a task can experience a priority inversion every time when it resumes on the CPU, if PCP is directly applied without a proper synchronization rule on $R^a$. However, we must point out that one advantage for the absence of synchronization for I/O devices is a higher system utilization.

Another scheduling alternative is to have a semaphore for the access synchronization on $R^a$. For the purpose of discussions, we adopt another (maybe more) restrictive algorithm in access synchronization. Let SRP be adopted for the scheduling of the task set, and a semaphore is adopted for the access synchronization of $R^a$. For simplicity of presentation, the semaphore for $R^a$ is referred to as $R^a$ when there is no ambiguity. Since there is only one unit per resource, and the number of units per request is only one, we only need to define the preemption levels for resources $R_1$, $R_2$, and $R^a$ when there is no resource available. When there is not a single unit for resource $R_1$ ($R_2/R^a$) available, the preemption level of $R_1$ ($R_2/R^a$) is equal to the priority of the task $\tau_H$ (i.e., $\lceil R_1 \rceil^0 = \lceil R_2 \rceil^0 = \lceil R^a \rceil^0 = \text{priority}(\tau_H)$). SRP requires no task being scheduled unless its priority is higher than the maximum preemption level of resources in the system, i.e., the *system preemption level*. Figure 2 shows the schedule under SRP. It is observed that $\tau_H$ suffers no priority inversion when it requests any lock on $R_1$, $R_2$, or $R^a$. Such a strong synchronization requirement results in the delay of the executions of $\tau_M$ and $\tau_L$ until time 16. Before time 16, either the CPU or the I/O device is idle.

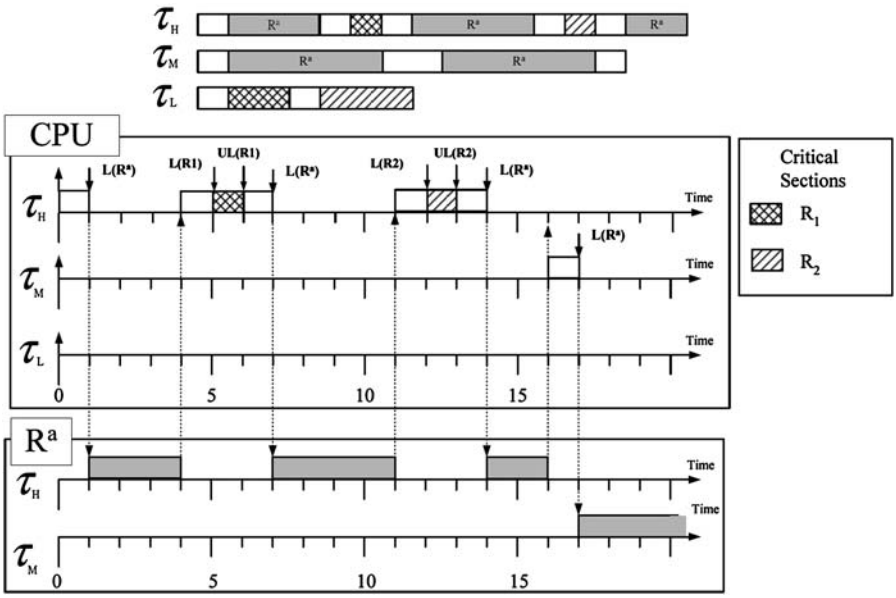



**Fig. 2** SRP applied in $R^a$

We must point out that I/O devices are very different from common resources for synchronization, such as semaphores, in which their access requires the running of the CPU. A tradeoff does exist between the system utilization and the maximum number of priority inversions (or priority inversion time):

- Suppose that no PCP ceiling rule is adopted to manage an I/O device. Each lock request of a task to the I/O device might introduce at most one priority inversion when the task tries to lock a semaphore later (after the I/O request is satisfied). Please see Fig. 1.
- Suppose that each I/O device is considered as a resource managed by SRP. The maximum number of priority inversions per task is one. Please see Fig. 2.
- Suppose that each I/O device is considered as a resource managed by PCP. The maximum number of priority inversions of a task is equal to one plus the number of lock requests to I/O devices.[1]

The above observations motivate the design of a resource synchronization protocol in which system engineers could trade the priority inversion time with the system utilization.

[1] A lower-priority task could execute over the CPU when a higher-priority task suspends itself to access an I/O device. Note that when the higher-priority task resumes from the I/O access, the lower-priority task could access the I/O device and later block the higher-priority task.

## 2.2 Process model, definitions, and terminologies

This section defines the process model and terminologies for this paper. We first classify resources as active or passive as follows:

**Definition 1** (Passive resources) A resource is passive if any accessing of the resource requires consumption of the CPU.

We say that a resource is *active* if it is not passive. Good examples of passive resources include semaphores, mutex locks, event objects, and database locks. A task must be executing while it is accessing a passive resource. Access over passive resources could be either exclusive or shared among tasks, depending on the characteristics of the resources and application logics, e.g., read and write locks on a piece of data. Examples of active resources are such as disks, printers, network adaptors, and transceivers. A task might issue a request on an active resource and resumes its execution if the request is asynchronous and granted. If the request is synchronous and granted, then the task is suspended until the request is fulfilled. Access over active resources could be preemptible or non-preemptible.

In this paper, we are interested in non-preemptible active resources, such as disks (that are observed in most cases). All active resources in this paper are non-preemptible unless they are explicitly identified as being preemptible. Let all active resources in this paper be accessed synchronously, and no I/O buffering be considered. In other words, when an access request for an active resource is granted, it is executed immediately. In this paper, we assume that a task voluntarily suspends its CPU execution when it accesses an active resource. By the definition of passive resources, a task can not lock any passive resources, before it can start accessing an active resource. Note that an active resource can never be involved in a deadlock. That is because (1) a task cannot lock another active resource when it is currently locking an active resource and (2) a task must release any passive resource before it suspends its CPU execution. Notably, for clearly explaining our protocol, we assume that task priorities are distinct, and we are interested in uniprocessor scheduling.

Tasks could be periodic or aperiodic. We call each instance of a task as a *job*, where a task is a template of its jobs. A task $\tau_i$ is a sequence of subtasks $\tau_{i,j}$. A subtask could be either a CPU execution or a period of time in accessing an active resource. If a subtask is a CPU execution, then it might lock or unlock any passive resources. The duration of a subtask $\tau_{i,j}$ is denoted as $c_{i,j}$, and the total execution time $c_i$ of $\tau_i$ is the sum of the CPU executions and the periods of time in accessing active resources of all subtasks $\tau_{i,j}$. Suppose that task $\tau_1$ first executes some computation-intensive code (i.e., $\tau_{1,1}$) and then accesses active resource $R^a$ (i.e., $\tau_{1,2}$). After the access of $R^a$, $\tau_1$ executes some other computation-intensive code (i.e., $\tau_{1,3}$) and then accesses $R^a$ again (i.e., $\tau_{1,4}$). Finally, $\tau_1$ executes some computation-intensive code (i.e., $\tau_{1,5}$) and completes, as shown in Fig. 3. The times required for subtasks $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}, \tau_{1,4}$, and $\tau_{1,5}$ are denoted as $c_{1,1}$,$c_{1,2}$,$c_{1,3}$,$c_{1,4}$, and $c_{1,5}$, where $c_1 = c_{1,1} + c_{1,2} + c_{1,3} + c_{1,4} + c_{1,5}$. Note that active resources are accessed synchronously. While $\tau_1$ is accessing an active resource, it must suspend its CPU execution until the access completes. No new lock could be obtained for a suspending task.

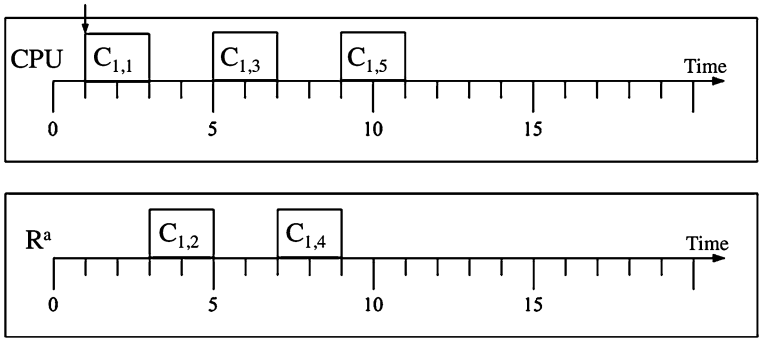



**Fig. 3** A task execution which involves active resource access

All (passive or active) resources must be locked before they are accessed. An active resource is *released* by a task when the access on the active resource by the task completes, and the corresponding lock is released. When a task accesses and locks an active resource several times in a period, the time point for the last releasing of the resource is called the *dismissing point* of the resource in the period. As shown in Fig. 3, an active resource $R^a$ is locked and released by task $\tau_1$ at time 3 and 5, respectively. It is locked and released again at time 7 and 9, respectively. The dismissing point of the active resource $R^a$ for task $\tau_1$ is at time 9.

For the rest of this paper, we shall propose a resource synchronization protocol to trade the priority inversion time with the system utilization.

## 3 A configurable synchronization protocol

### 3.1 Overview

Existing research results on resource synchronization are mainly concerned with minimizing priority inversion. PCP guarantees at most one priority inversion for any higher-priority task in uniprocessor fixed-priority systems. SRP later extends the idea to manage multiple resources and dynamic priority scheduling. Although various excellent resource synchronization protocols have been proposed, little work has been done to adjust the numbers of priority inversions for tasks.

In this section, we shall propose two resource synchronization protocols with an adjustment mechanism for priority inversion management. The basic protocol extends the ceiling rules of PCP so that higher-priority tasks could receive a larger number of priority inversions to trade for the schedulability of lower-priority tasks. A ceiling table is proposed to set up the maximum numbers of priority inversions for tasks. Note that the table of preemption levels in SRP is proposed for a purpose very different from that of the ceiling table proposed in this table.[2] In the basic protocol, only passive resources are considered, and we are interested in uniprocessor

[2]The table of preemption levels in SRP is to check for the availability of resources before a task begins its execution.

task scheduling in this paper. We then extend the basic protocol in the considerations of active resources such that tasks might suspend themselves for I/O operations (on active resources). Guidelines are also proposed for the setup of the ceiling table based on the priority-inversion requirements of tasks.

## 3.2 The basic configurable ceiling protocol

The basic protocol extends the ceiling rules of PCP to have a tradeoff between the number of priority inversions of higher-priority tasks and the schedulability of lower-priority tasks. A ceiling table is first defined for adjusting the maximum priority inversion time for tasks. Protocol rules are then proposed for task scheduling based on the ceiling table.

### *3.2.1 The ceiling table*

Under PCP, the ceiling of a resource is defined as the maximum priority of tasks which might access the resource. The lock request of a task could not be granted unless the priority of the task is higher than the maximum ceiling of resources locked by other tasks. Ceiling($R_p$) denotes the ceiling of resource $R_p$. With a ceiling table, the resource synchronization rule will be modified.

The purpose of a ceiling table is for adjusting the maximum priority inversion time for tasks. CT($\tau_i$, $R_p$) is an entry in the ceiling table to represent the way in which task $\tau_i$ might access resource $R_p$. When CT($\tau_i$, $R_p$) = 0, $\tau_i$ will not access $R_p$ in any way. If CT($\tau_i$, $R_p$) = 1, then $\tau_i$ might lock and access $R_p$. If CT($\tau_i$, $R_p$) = $*$, then $\tau_i$ could tolerate priority inversion resulting from the access conflict of $R_p$. Guidelines for the setup of the ceiling table are in Sect. 3.4.

With a ceiling table, the ceiling of each resource $R_p$ (Ceiling($R_p$)) is revised as follows: Ceiling($R_p$) is the maximum priority of tasks $\tau_i$ with CT($\tau_i$, $R_p$) = 1. The lock request of a task $\tau_i$ on resource $R_q$ is granted if the priority of $\tau_i$ is higher than the maximum ceiling of resources locked by other tasks. Otherwise, $\tau_i$ is blocked. Let $R_p$ be a locked resource owning the maximum ceiling such that $\tau_i$ is blocked, and task $\tau_j$ currently lock $R_p$. We say that $\tau_i$ is blocked by $\tau_j$. We must point out that the revised definition of Ceiling($R_p$) does not consider the priorities of tasks $\tau_i$ when CT($\tau_i$, $R_p$) = $*$. Such a modification virtually gives up some privileges of those tasks with CT($\tau_i$, $R_p$) = $*$ when $R_p$ is locked. That is, when $R_p$ is locked, other tasks might still have a possibility to lock other resources even though their priorities are lower than the priorities of tasks with CT($\tau_i$, $R_p$) = $*$.

Consider a ceiling table in Table 1, based on the definition of Ceiling($R_p$) in PCP, Ceiling($R_1$) = Ceiling($R_2$) = Ceiling($R_3$) = the priority of $\tau_1$, and Ceiling($R_4$) =

**Table 1** An example ceiling table

| Task/Resource | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | 1 | 1 | * | 0 | 0 |
| $\tau_2$ | 0 | 0 | * | * | 1 |
| $\tau_3$ | 0 | 1 | 1 | * | 1 |
| $\tau_4$ | 0 | 1 | 1 | 1 | 1 |

Ceiling($R_5$) = the priority of $\tau_2$. Different from PCP, the revised ceiling definitions for the ceiling table (i.e., Table 1) are as follow: Ceiling($R_1$) = the priority of $\tau_1$, Ceiling($R_2$) = the priority of $\tau_1$, Ceiling($R_3$) = the priority of $\tau_3$, Ceiling($R_4$) = the priority of $\tau_4$, Ceiling($R_5$) = the priority of $\tau_2$.

In the following section, we shall propose a resource synchronization protocol based on the concept of the ceiling table.

### *3.2.2 Resource synchronization protocol*

The purpose of this section is to propose a resource synchronization protocol called the *basic configurable ceiling protocol* (BCCP) based on the concept of ceiling tables. We are interested in uniprocessor scheduling with only access over passive resources in the basic protocol. Since PCP is adopted to illustrate the idea, we adopt a fixed-priority assignment policy, such as the Rate Monotonic Scheduling (RMS) algorithm (Liu and Layland 1973), where RMS assigns a higher priority to a task with a smaller period. The objective is to provide a tradeoff between the numbers of priority inversions for higher-priority tasks and the deadline satisfaction of lower-priority tasks. The ceiling table is to provide a way to adjust the maximum numbers of priority inversions for tasks.

The scheduling protocol is defined as follows: The ceilings of all resources are defined based on the given ceiling table, as described in Sect. 3.2.1. The ready task with the highest priority is dispatched for execution. The lock request of a task $\tau_i$ on resource $R_p$ is granted if the priority of $\tau_i$ is higher than the maximum ceiling of resources locked by other tasks, and $R_p$ is free. Otherwise, $\tau_i$ is blocked. Two conditions must be considered for the occurrence of a blocking: One possibility is that the priority of $\tau_i$ is no higher than the maximum ceiling of resources locked by other tasks. Let $R_q$ be the locked resource owning the maximum ceiling such that $\tau_i$ is blocked, and $\tau_j$ currently locks $R_q$. We say that $\tau_i$ is blocked by $\tau_j$. Another possibility is that $R_p$ is locked by another task $\tau_j$ although the priority of $\tau_i$ is higher than the maximum ceiling of resources locked by other tasks. In this case, $\tau_i$ is *directly blocked* by $\tau_j$. The occurrence of such direct blocking results from the lowering of the priority ceiling of $R_p$ (because CT($\tau_i$, $R_p$) = $*$). When $\tau_i$ is blocked by $\tau_j$, $\tau_j$ inherits the priority of $\tau_i$. The priority inheritance is done transitively. When $\tau_i$ is no longer blocked by $\tau_j$, the priority of $\tau_j$ resumes at the priority when the priority inheritance occurs.

The detailed definition of the protocol is as follows. Note that $\Pi(\tau_i)$ denotes the maximum ceiling of resources locked by other tasks except $\tau_i$.

**Task Scheduling:**

(a) The current priority $\pi(\tau_i)$ of task $\tau_i$ equals to the priority assigned to $\tau_i$ when $\tau_i$ arrives. The current priority $\pi(\tau_i)$ of task $\tau_i$ remains unless priority inheritance is applied.

(b) Tasks are scheduled preemptively in a priority-driven manner according to their current priorities.

**Resource Allocation:**

Let $\Pi(\tau_i)$ denote the maximum ceiling of resources currently locked by tasks other than task $\tau_i$. Whenever a task $\tau_i$ requests a resource $R_p$, the following two conditions are considered:

(a) If $\tau_i$'s current priority $\pi(\tau_i)$ is higher than $\Pi(\tau_i)$

(1) If $R_p$ is free, then $R_p$ is allocated to $\tau_i$.
(2) If $R_p$ is held by another task $\tau_j$, then $\tau_i$ is blocked by $\tau_j$
(b) If the current priority $\pi(\tau_i)$ of $\tau_i$ is not higher than $\Pi(\tau_i)$ of the system, $R_p$ is allocated to $\tau_i$,
only if $\tau_i$ is the task holding the resources whose priority ceiling equal to $\Pi(\tau_i)$;
otherwise, $\tau_i$ is blocked by task $\tau_j$ which holds resource $R_q$,
where the ceiling of resource $R_q$ equals to $\Pi(\tau_i)$.

**Priority Inheritance:**
If $\tau_i$ is blocked by task $\tau_j$, then $\tau_j$ inherits the current priority $\pi(\tau_i)$ of task $\tau_i$.
When $\tau_j$ no longer blocks $\tau_i$, the priority inheritance ceases the existence.
That is, $\tau_j$ resumes the priority that it has right before it inherits a priority from $\tau_i$
Priority inheritance is transitive.

### *3.2.3 Properties and protocol analysis*

We could show the correctness of the following properties for BCCP: Note that no active resources are considered for BCCP. Assume that tasks are sorted by their priorities and $\tau_0$ is the highest priority task.

Before the properties of BCCP are proved, we revise a given ceiling table CT() into a corresponding ceiling table CT′() as follows: $\mathrm{CT}'(\tau_i, R_p) = \mathrm{CT}(\tau_i, R_p)$ for any task $\tau_i$ and any resource $R_p$ except the following cases: (1) $\mathrm{CT}'(\tau_i, R_p) = 1$ if $\mathrm{CT}(\tau_i, R_p) = *$, and there is an entry $\mathrm{CT}(\tau_a, R_p) = 1$ and $i > a$. It is because the ceiling of $R_p$ would be higher than the priority of $\tau_i$ regardless of whether $\mathrm{CT}'(\tau_i, R_p)$ is equal to 1 or $*$. (2) $\mathrm{CT}'(\tau_i, R_p) = 1$ if $\mathrm{CT}(\tau_i, R_p) = *$, and $\mathrm{CT}(\tau_k, R_p) = 0$ for all $k > i$. It is because no lower-priority task will lock $R_p$. In other words, $\tau_i$ would not be blocked by any lower-priority tasks because of $R_p$, regardless of whether $\mathrm{CT}'(\tau_i, R_p)$ is equal to 1 or $*$. As a result, we also set $\mathrm{CT}'(\tau_i, R_p) = 1$ if $\mathrm{CT}(\tau_i, R_p) = *$, in which $\tau_i$ is the lowest-priority task in the system. Actually this is a special case of case (2). CT′ is called the *revised ceiling table* of a given ceiling table CT(). We could show the following lemma:

**Lemma 1** *A task is blocked by another task under BCCP with a given ceiling table* CT() *if and only if the former task is blocked by the later task under BCCP with the revised ceiling table* CT′().

*Proof* The if-part of the lemma can be proved as follows: Suppose that a lock request of a task $\tau_i$ on resource $R_p$ is blocked under CT′(). Such a blocking could only occur when the priority of $\tau_i$ is no higher than the maximum ceiling of resources currently locked by other tasks under CT′(), or when $R_p$ is locked by another task $\tau_k$.

Suppose that $\tau_j$ is the task that locks the resource with the maximum ceiling and blocks $\tau_i$. Two cases are under consideration: (1) As mentioned in the previous paragraph, $\mathrm{CT}'(\tau_i, R_p)$ is revised as 1 if $\mathrm{CT}(\tau_i, R_p) = *$, and there is an entry $\mathrm{CT}(\tau_a, R_p) = 1$, where $a < i$. As a result, Ceiling($R_p$) remains the same for both CT() and CT′(), and it is equal to the priority of $\tau_a$. A lock request of a task $\tau_i$ on resource $R_p$ will not be granted under CT() as well. (2) $\mathrm{CT}'(\tau_i, R_p)$ is revised as 1 if $\mathrm{CT}(\tau_i, R_p) = *$, and there does not exist a non-zero entry $\mathrm{CT}(\tau_k, R_p)$ where $i < k$. In other words, any task which has a priority lower than $\tau_i$ would never lock $R_p$. The revision of $\mathrm{CT}'(\tau_i, R_p)$ would not introduce any new blocking. Even though the ceiling of $R_p$ is raised (i.e., to the priority of $\tau_i$, the lowest priority among all the tasks

that can lock $R_p$), no new blockings are introduced. If a task's resource request can not be immediately granted because of $R_p$'s revised ceiling, then the task's priority is never higher than any task that can lock $R_p$. So the revised ceiling introduces no new priority inversion (i.e., blocking).

Suppose that a lock request of a task $\tau_i$ on resource $R_p$ is not granted under CT′() because $R_p$ is locked by another task $\tau_j$. It is obvious that the lock request of $\tau_i$ on $R_p$ will not be granted under CT() because $R_p$ is locked already. The only-if-part of the lemma can be proved in a similar way. □

Given a task $\tau_i$, let $\phi(\tau_i)$ denote the number of resources $R_p$ with $\mathrm{CT}'(\tau_i, R_p) = *$. Note that there is a $n$ task set, and all tasks are reordered and renamed such that $\tau_i$ has a priority higher than $\tau_{i+1}$ does, for $(n-1) \geq i \geq 0$.

**Theorem 1** *No task $\tau_i$ could be directly blocked by lower-priority tasks for more than $\phi(\tau_i) + 1$ times in each of its period.*

*Proof* The correctness of this theorem follows directly from Lemma 1 and the following observation (based on CT′()): First, no direct blocking would be introduced to $\tau_i$ due to any access on a resource $R_p$ if $\mathrm{CT}'(\tau_i, R_p) = 0$ because $\tau_i$ would not access $R_p$. Each resource $R_p$ with $\mathrm{CT}'(\tau_i, R_p) = *$ could introduce only one direct blocking for $\tau_i$ because Ceiling($R_p$) is lower than the priority of $\tau_i$. Further, when some task accesses a resource $R_p$ with $\mathrm{CT}'(\tau_i, R_p) = 1$, the ceiling of $R_p$ will prevent any other task with a priority lower than that of $\tau_i$ from directly blocking $\tau_i$ again. In other words, only one direct blocking would be possibly introduced to $\tau_i$ for all resources with $\mathrm{CT}'(\tau_i, R_p) = 1$. As a result, the maximum number of direct blocking of $\tau_i$ is no more than $\phi(\tau_i) + 1$. □

Based on Theorem 1, the maximum number of direct blocking for each task could be derived from a given ceiling table. For example, the maximum number of direct blocking for $\tau_1$ under BCCP with the ceiling table, as shown in Table 1, is $1 + 1 = 2$. Those of $\tau_2$ and $\tau_3$ are $2 + 1 = 3$ and $1 + 1 = 2$, respectively. Note that $\tau_4$ will not suffer from any direct blocking because it is the task with the lowest priority. Theorem 1 shows the maximum number of direct blocking suffered by a task in a period. The rest of this section derives a bound on the maximum duration of priority inversion time possibly suffered by a task in a period, where some of the priority inversion time might come from transitive blocking.

Priority inversion could come from direct and/or indirect blocking. Since the ceilings of resources could be lower than their corresponding PCP ceilings, indirect blocking (i.e., transitive blocking) might occur. The possibility of transitive blocking could be observed from a given ceiling table CT(): Let a symbol $+$ denote some value equal to $*$ or 1. Suppose that $\mathrm{CT}(\tau_i, R_p) = +$ and $\mathrm{CT}(\tau_j, R_p) = +$ for some resource $R_p$, where the priority of $\tau_i$ is higher than that of $\tau_j$. Suppose that $\mathrm{CT}(\tau_j, R_q) = *$ and $\mathrm{CT}(\tau_{j+1}, R_q) = +$ for some other resource $R_q$, as shown in Fig. 4(a). Let tasks be sorted in increasing order of their priorities. (We first consider the case in which every task has a distinct priority.) That is, the priority of $\tau_{j+1}$ is lower than that of $\tau_j$. Let $R_q$ be locked by $\tau_{j+1}$ when $\tau_j$ locks $R_p$. The lock request of $\tau_j$ on $R_p$ is successful because $\mathrm{CT}(\tau_j, R_q) = *$. The lock request of $\tau_i$ on $R_p$ later results in a direct blocking

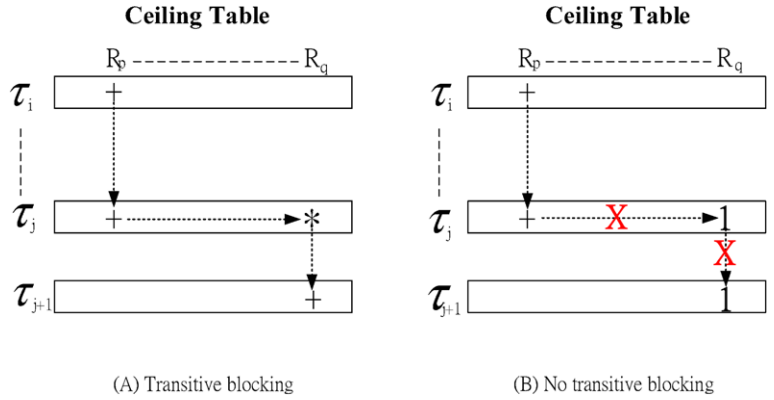



**Fig. 4** The reason of transitive blocking

of $\tau_i$ by $\tau_j$ (i.e., the path $+ \longrightarrow +$). As a result, $\tau_j$ resumes its execution. When $\tau_j$ requests $R_q$, $\tau_j$ is directly blocked by $\tau_{j+1}$ because $R_q$ is already locked by $\tau_{j+1}$ (i.e., the path $* \longrightarrow +$). Such a transitive blocking $\tau_i - \tau_j - \tau_{j+1}$ could occur because $\mathrm{CT}(\tau_j, R_q) = *$. As astute readers might notice, a transitive blocking might occur when one of the tasks in the transitive blocking locks a resource $R_q$ such that a higher-priority task $\tau_j$ in the transitive blocking later request $R_q$, where $\mathrm{CT}(\tau_j, R_q) = *$. On the other hand, when the above observation does not exist, no transitive blocking will occur. We use a counter example, as shown in Fig. 4(b), to provide an explanation: As in the example shown in Fig. 4(a), $\mathrm{CT}(\tau_i, R_p) = +$ and $\mathrm{CT}(\tau_j, R_p) = +$, and the priority of $\tau_i$ is higher than that of $\tau_j$. Now let $\mathrm{CT}(\tau_j, R_q) = 1$, and $\mathrm{CT}(\tau_{j+1}, R_q) = 1$. Let $R_q$ be locked by $\tau_{j+1}$ when $\tau_j$ requests $R_p$. The lock request of $\tau_j$ is blocked because $\mathrm{CT}(\tau_j, R_q) = 1$. As a result, $\tau_i$ is not blocked by $\tau_j$ on $R_p$, when $\tau_i$ later requests $R_p$, compared to the former example (i.e., the path $+ \longrightarrow +$). A transitive blocking $\tau_i - \tau_j - \tau_{j+1}$ does not occur.

A task can be transitively blocked only if it is directly blocked. Indirect blocking also can be caused by priority inheritance and priority ceiling. In PCP, a task can be blocked by another task in terms of priority inheritance only if it can be blocked by the same task in terms of priority ceiling. In our approach, resource ceilings are not higher than they are in PCP. Because each resource is only one instance, a task instance cannot be blocked on a resource because of priority inheritance and priority ceiling. Similarly, a task can be blocked on a resource because of either direct blocking or indirect blocking.

Algorithm BCCP_MAX_BLOCKING_TIME (shown in Appendix) is presented for computing the longest duration a task can be blocked in BCCP. This procedure is in two steps. The first step is to check whether or not a task will be directly/indirectly blocked on a resource. As mentioned in the last paragraph, if one task can be directly or indirectly blocked by another task, then the two tasks can be blocked on the same collection of resources. In the second step, the longest duration that a task instance can be blocked is computed by summing all the longest locking durations of each resource that the task can be blocked on. A task instance cannot be blocked on a resource more than once because every resource has only one unit, and there is no

**Table 2** An example ceiling table for the extended protocol

| Task/Resource | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | 3 | 1 | 3 | 4 | 0 |
| $\tau_2$ | 1 | 0 | 2 | 3 | 1 |
| $\tau_3$ | 1 | 1 | 1 | 2 | 1 |
| $\tau_4$ | 1 | 1 | 1 | 1 | 1 |

active resource in BCCP. The time complexity of this algorithm is $O(n * m)$, where $n$ is the number of tasks and $m$ is the number of passive resources in the system.

Let the maximum duration in the locking of resource $R_p$ of all tasks be denoted as $B_{R_p}$. The maximum blocking time of $\tau_1$ under BCCP with the ceiling table, as shown in Table 1, is $\max(B_{R_1}, B_{R_2}) + B_{R_3} + B_{R_4}$, where $\max(B_{R_1}, B_{R_2})$ comes from "1" 's, and $B_{R_3}$ comes from the asterisk symbols and "0" 's in the table, and $B_{R_4}$ comes from the transitive blocking. Those of $\tau_2$ and $\tau_3$ are $B_{R_5} + B_{R_3} + B_{R_4}$ and $\max(B_{R_2}, B_{R_3}, B_{R_5}) + B_{R_4}$, respectively. Note that $\tau_4$ will not suffer from any blocking because it is the task with the lowest priority.

## 3.3 The extended configurable ceiling protocol

### *3.3.1 The Ceiling Table*

In this section, we shall extend the basic configurable ceiling protocol (BCCP) for systems with active resources. Since a task could voluntarily suspend its CPU execution until completing an active resource request, the tradeoff between the priority inversion management and the system utilization becomes a critical issue. Under the extended protocol, system designers are allowed to fill in the maximum number of priority inversions for each task. Furthermore, each entry in the table denotes the maximum number of priority inversions for the corresponding task caused by any access conflicts over the corresponding resource (Please see Theorem 2). The ceiling table for the extended protocol is called the *extended ceiling table* for the rest of this paper.

During the on-line operations, the table could be used to manage the number of priority inversions for each task and to derive a proper ceiling for each resource. The main idea is as follows: The initial value of $\mathrm{CT}(\tau_i, R_p)$ denotes the maximum number of priority inversions for any access conflicts of resource $R_p$ for task $\tau_i$. When $\mathrm{CT}(\tau_i, R_p) = N$ at some time $t$, it means that $\tau_i$ could tolerate additional $N$ priority inversions over any access conflicts of resource $R_p$. Note that only passive resources have corresponding entries in the extended ceiling table. The idea of the extended ceiling table is to better manage of passive resources when active resources are available in the system.

After setting the initial values for the extended ceiling table, the system dynamically derives the ceiling of each resource in an on-line fashion: When $\mathrm{CT}(\tau_i, R_p) = N$ for some $N > 1$, and a direct blocking occurs for $\tau_i$ on $R_p$, $\mathrm{CT}(\tau_i, R_p)$ is decremented by one. The derivation of the ceiling for resource $R_p$ only considers the entries with $\mathrm{CT}(\tau_i, R_p) = 1$ (for any task $\tau_i$). The rationale behind this rule is that when

$\mathrm{CT}(\tau_i, R_p) > 1$, $\tau_i$ could still tolerate priority inversion resulting from the access conflict of $R_p$. As a result, the setting of the ceiling of $R_p$ does not need to consider the priority of $\tau_i$. It is similar to the case when $\mathrm{CT}(\tau_i, R_p) = *$ under BCCP. In the next subsection, we shall extend BCCP with the extended ceiling table.

The ceiling derivation of passive resources is as presented in the previous paragraph. The ceiling of an active resource is defined as the maximum priority of the tasks that have already locked the resource and have not reached the dismissing point of the resource. If there is no such task in the system, then the ceiling of the active resource is the lowest priority in the system.

### *3.3.2 Resource synchronization protocol*

This section extends BCCP by considering active resources and a more precise management of the number of priority inversions due to each resource.

Given a system with a collection of passive resources $\{\ldots, R_p, \ldots\}$, the extended ceiling table CT(), and a collection of active resources $\{\ldots, R_g^a, \ldots\}$, the system always dispatches the ready task with the highest priority.

The lock request of a task $\tau_i$ on $R_g^a$ is granted if the priority of $\tau_i$ is no less than $\mathrm{Ceiling}(R_g^a)$, and $R_g^a$ is currently not locked. If the lock request of task $\tau_i$ on $R_g^a$ is granted, then $\mathrm{Ceiling}(R_g^a)$ is replaced with the priority of $\tau_i$. Otherwise, the request is blocked. We say that $\tau_i$ is *directly blocked* by $\tau_j$ if $R_g^a$ is currently locked by some other task $\tau_j$, and the priority of $\tau_i$ is no less than $\mathrm{Ceiling}(R_g^a)$ (i.e., the priority of $\tau_j$). Note that if some higher-priority task $\tau_h$ once locks $R_g^a$ and has not reached its dismissing point, then $\mathrm{Ceiling}(R_g^a)$ is larger than the priority of $\tau_i$. We say that $\tau_i$ is *directly obstructed* by $\tau_h$ because the lock request of a task $\tau_i$ on $R_g^a$ is not granted.

Let $\Gamma$ and $\gamma$ denote the maximum priority and the minimal priority of tasks which are currently suspending themselves, respectively. Therefore, $\Gamma$ and $\gamma$ take all active resources into account. The lock request of a task $\tau_i$ on a passive resource $R_p$ is granted if $R_p$ is not locked, the priority of $\tau_i$ is higher than the maximum ceiling of passive resources locked by other tasks, and one of the following two conditions holds: (1) The priority of $\tau_i$ is higher than $\Gamma$ (2) $\mathrm{Ceiling}(R_p)$ is less than $\gamma$. Note that $\mathrm{Ceiling}(R)$ denotes the ceiling of resource $R$. Otherwise, the request is not granted and postponed. The task that blocks $\tau_i$ is determined as follows:

If the priority of $\tau_i$ is not higher than the maximum ceiling of passive resources locked by other tasks, then $\tau_i$ is blocked by the task locking the resource with the maximum ceiling. If the priority of $\tau_i$ is higher than the maximum ceiling of passive resources locked by other tasks, but $R_p$ is locked, then $\tau_i$ is directly blocked by the task locking $R_p$. If the priority of $\tau_i$ is higher than the maximum ceiling of passive resources locked by other tasks, $R_p$ is not locked, but if the two conditions presented in the previous paragraph both fail, then we say that $\tau_i$ suffers from an *active resource obstructing*. $\tau_i$ is said to be *directly obstructed* by some of the tasks that currently suspend for accessing active resources such that the above two conditions both fail. The occurrence of an active resource obstruction is to prevent a higher-priority but suspending task from extra priority inversion when the task resumes from the suspension due to the access of the active resource. The side-effects of an active resource obstructing might result in executing some task with a priority lower than that of $\tau_i$.

When a task $\tau_i$ is blocked by another task $\tau_j$ because of the requesting of a passive resource, $\tau_j$ inherits the priority of $\tau_i$. Priority inheritance is transitive. When a blocking no longer exists, the corresponding task resumes its priority when the priority inheritance occurs. Note that when $\tau_i$ is blocked by $\tau_j$ because of the requesting of an active resource, the priority inheritance is not applied.

The detailed protocol definition of ECCP is summarized as follows.

**Task Scheduling:**
(a) The current priority $\pi(\tau_i)$ of task $\tau_i$ equals to the priority assigned to $\tau_i$ when $\tau_i$ arrives. The current priority $\pi(\tau_i)$ of task $\tau_i$ remains unless priority inheritance is applied.
(b) Tasks are scheduled preemptively in a priority-driven manner according to their current priorities.

**Resource Allocation:**
Let $\Pi(\tau_i)$ denote the maximum ceiling of resources currently locked by tasks other than task $\tau_i$.
Let $\Gamma$ and $\gamma$ denote the maximum priority and the minimal priority of tasks which are currently suspending themselves, respectively.
Whenever a task $\tau_i$ requests a passive resource $R_p$, the following two conditions are considered:
(a) If $\tau_i$'s current priority $\pi(\tau_i)$ is higher than $\Pi(\tau_i)$
  (1) If $R_p$ is free
    (A) If $\pi(\tau_i)$ is higher than $\Gamma$, $R_p$ is allocated to $\tau_i$.
    (B) If $\pi(\tau_i)$ is no higher than $\Gamma$ and $R_p$'s current ceiling is less than $\gamma$, $R_p$ is allocated to $\tau_i$.
    (C) If neither $(A)$ nor $(B)$ holds, the request is not granted and $\tau_i$ is obstructed.
  (2) If $R_p$ is held by another task $\tau_j$, then $\tau_i$ is blocked by $\tau_j$
(b) If $\tau_i$'s priority $\pi(\tau_i)$ is not higher than $\Pi(\tau_i)$ of the system, $R_p$ is allocated to $\tau_i$, only if $\tau_i$ is the task holding the resources whose priority ceiling equal to $\Pi(\tau_i)$; otherwise, $\tau_i$ is blocked by task $\tau_j$ which holds resource $R_q$, where the ceiling of resource $R_q$ equals to $\Pi(\tau_i)$.
Let Ceiling($R_g^a$) denote the current ceiling of the corresponding active resource.
Whenever a task $\tau_i$ requests an active resource $R_g^a$, the following two conditions are considered:
(a) If $\tau_i$'s current priority $\pi(\tau_i)$ is no less than Ceiling($R_g^a$)
  (1)If $R_g^a$ is free, $R_g^a$ is allocated to $\tau_i$.
  (2)If $R_g^a$ is held by another task $\tau_j$, then $\tau_i$ is blocked by $\tau_j$
(b) If $\tau_i$'s current priority $\pi(\tau_i)$ is less than Ceiling($R_g^a$), the request is not granted and $\tau_i$ is obstructed.

**Priority Inheritance:**
If $\tau_i$ is blocked by task $\tau_j$ due to requesting a passive resource $R_p$, then $\tau_j$ inherits the current priority $\pi(\tau_i)$ of task $\tau_i$.
When $\tau_j$ no longer blocks $\tau_i$, the priority inheritance ceases the existence.
That is, $\tau_j$ resumes the priority that it has right before it inherits a priority from $\tau_i$.
Priority inheritance is transitive.

### *3.3.3 Properties and protocol analysis*

The purpose of this section is to derive the maximum number of blocking for each task under ECCP. We shall comment on the value assignment of entry values in the extended ceiling table in a later section.

We shall first show that each extended ceiling table CT() under ECCP has a corresponding revised extended ceiling table CT′() so the schedules of task executions with either extended ceiling table are the same. Let $\theta_i$ denote the total

number of accesses on all active resources by some task $\tau_i$, and $\mu_{i,p}$ denote the number of requests on a passive resource $R_p$ by $\tau_i$ in each period. $\mathrm{CT}'(\tau_i, R_p) = \mathrm{CT}(\tau_i, R_p)$ for any task $\tau_i$ and any resource $R_p$ except the following four cases: (1) $\mathrm{CT}'(\tau_i, R_p) = min(\mu_{i,p}, \theta_i)$ if $\mathrm{CT}(\tau_i, R_p) > \mu_{i,p}$ or $\mathrm{CT}(\tau_i, R_p) > \theta_i$. It is because the number of priority inversions suffered by $\tau_i$ because of the requesting of $R_p$ is bound by $\mu_{i,p}$ and $\theta_i$. (2) $\mathrm{CT}'(\tau_i, R_p) = 1$ if $\mathrm{CT}(\tau_i, R_p) > 1$, and there is an entry $\mathrm{CT}(\tau_a, R_p) = 1$ for $i > a$. It is because the ceiling of $R_p$ would be higher than the priority of $\tau_i$, regardless of whether $\mathrm{CT}'(\tau_i, R_p)$ is no less than 1. (3) $\mathrm{CT}'(\tau_i, R_p) = 1$ if $\mathrm{CT}(\tau_i, R_p) > 1$, and $\mathrm{CT}(\tau_k, R_p) = 0$ for all $k > i$. It is because no lower-priority task will lock $R_p$. In other words, $\tau_i$ would not be blocked by any lower-priority tasks because of $R_p$, regardless of whether $\mathrm{CT}'(\tau_i, R_p)$ is no less than 1. We also set $\mathrm{CT}'(\tau_i, R_p) = 1$ if $\mathrm{CT}(\tau_i, R_p) > 1$, in which $\tau_i$ is the lowest-priority task in the system. It is a special case of case (3). $\mathrm{CT}'$ is called the *revised extended ceiling table* of a given extended ceiling table CT() based on the above revision rules. We could show the following lemma:

**Lemma 2** *A task is blocked by another task under ECCP with a given extended ceiling table* CT() *if and only if the former task is blocked by the later task under ECCP with the revised extended ceiling table* $\mathrm{CT}'()$.

*Proof* The proof is similar to that of Lemma 1. □

**Theorem 2** *Under ECCP, no task $\tau_i$ could be directly blocked by lower-priority tasks for more than*

$$M + 1 + \sum_{\mathrm{CT}'(\tau_i, R_p) > 1} \left(\mathrm{CT}'(\tau_i, R_p) - 1\right)$$

*times in each of its period, with the presence of $M$ active resources.*

*Proof* This theorem could be proved in a similar way to that of Theorem 1: No direct blocking would be introduced to any task $\tau_i$ due to any access on a resource $R_p$ if $\mathrm{CT}'(\tau_i, R_p) = 0$ because $\tau_i$ would not access $R_p$. Only one direct blocking could possibly be introduced to $\tau_i$ for all resources with $\mathrm{CT}'(\tau_i, R_p) = 1$ because the ceiling of $R_p$ will prevent any other task with a priority lower than that of $\tau_i$ from directly blocking $\tau_i$ again. For any resource $R_p$ with $\mathrm{CT}'(\tau_i, R_p) > 1$, a task $\tau_i$ might be directly blocked once whenever $\tau_i$ resumes its execution due to the waiting of the service over some active resource $R_g^a$. Since $\mathrm{CT}'(\tau_i, R_p)$ is decreased by one whenever a direct blocking occurs to $\tau_i$ due to access on $R_p$, there is no more than $(\mathrm{CT}(\tau_i, R_p) - 1)$ direct blocking for $\tau_i$ due to access on $R_p$. Note that when $\mathrm{CT}(\tau_i, R_p)$ becomes one, Ceiling$(R_p)$ is set to the priority of $\tau_i$.

An active resource could also contribute one potential direct blocking to $\tau_i$, because the ceiling of the active resource is raised to the priority of $\tau_i$ until the dismissing point of the period is reached. As a result, no task $\tau_i$ could be directly blocked for more than $M + 1 + \sum_{\mathrm{CT}'(\tau_i, R_p) > 1} (\mathrm{CT}'(\tau_i, R_p) - 1)$ times in each of its period. □

Based on Theorem 2, the maximum number of direct blocking for each task could be derived from a given extended ceiling table. For example, consider that there is

only an active resource, the maximum number of direct blocking for $\tau_1$ under ECCP with the extended ceiling table, as shown in Table 2, is $1+1+(3-1)+(3-1)+(4-1)=9$. Those of $\tau_2$ and $\tau_3$ are $1+1+(2-1)+(3-1)=5$ and $1+1+(2-1)=3$, respectively. Note that $\tau_4$ will not suffer from any direct blocking because it is the task with the lowest priority .

Similar to the basic protocol, some of the priority inversion time might come from indirect blocking. We propose an algorithm ECCP_MAX_BLOCKING_TIME to derive a bound on the maximum duration of priority inversion time possibly suffered by a task in a period. This procedure is in two steps. The first step is to check whether or not a task will be directly/indirectly blocked on a resource. As mentioned in the above section, if one task can be directly or indirectly blocked by another task, then the two tasks can be blocked on the same collection of resources. In the second step, the longest duration that a task instance can be blocked is computed by summing all the longest locking durations of each resource that the task can be blocked on. Note that the potential blocking time contributed by an active resource $R_g^a$ is the longest duration of all sub-jobs in accessing $R_g^a$ rather than the total duration of a job in accessing the active resource. The time complexity of ECCP_MAX_BLOCKING_TIME is $O(n * \max(m, M))$ (Please see the Appendix), where $n$ is the number of tasks, $m$ is the number of passive resources and $M$ is the number of active resources in a given task set.

Based on Algorithm ECCP_MAX_BLOCKING_TIME, the maximum blocking time suffered by each task (in a period) could be derived from a given extended ceiling table. Note that the maximum duration in the locking of resource $R_p$ of all tasks is denoted as $B_{R_p}$, and the longest duration of all sub-jobs in accessing $R_g^a$ is denoted as $B_{R_g^a}$. For example, consider that there is only an active resource $R_1^a$, the maximum blocking time of $\tau_1$ under ECCP with the extended ceiling table, as shown in Table 2, is $B_{R_1^a} + \max(B_{R_1}, B_{R_2}, B_{R_3}, B_{R_4}) + 2 * B_{R_1} + 2 * B_{R_3} + 3 * B_{R_4}$, Those of $\tau_2$ and $\tau_3$ are $B_{R_1^a} + \max(B_{R_1}, B_{R_3}, B_{R_4}, B_{R_5}) + B_{R_3} + 2 * B_{R_4}$ and $B_{R_1^a} + \max(B_{R_1}, B_{R_2}, B_{R_3}, B_{R_4}, B_{R_5}) + B_{R_4}$, respectively. Note that $\tau_4$ will not suffer from any blocking because it is the task with the lowest priority.

### 3.4 Remark: the setting of the BCCP and ECCP ceiling tables

In this section, we shall provide some guidelines to set up the ceiling table CT() for a given system. Similar to the requirements of PCP, tasks scheduled by BCCP and ECCP must have their resource requests known in advance. That is, which resource will be used by which task, and the number of requests on each active or passive resource in a period of each task. The information on the duration of each request could further help in deriving the priority inversion time for each task and providing schedulability analysis.

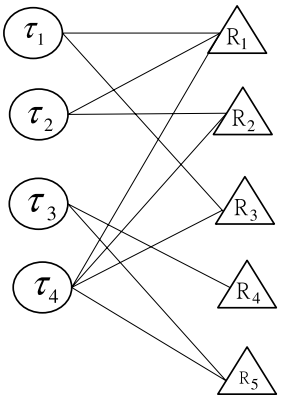
Let $n$ be the number of tasks, $m$ be the number of passive resources and $M$ be the number of active resources in a given task set. An algorithm with time complexity $O(n^2 * m * M)$ is proposed for ECCP to set ceiling tables (Please see the Appendix). The rationales behind the algorithm is: First, we run a schedulability analysis, such as the Rate Monotonic Analysis, to derive the maximum blocking time tolerable to each task. Initially, the maximum number of priority inversions for each task is set

as 1 (i.e., $\mathrm{CT}(\tau_i, R_p) = 0$ if $\tau_i$ would not use $R_p$; otherwise, $\mathrm{CT}(\tau_i, R_p) = 1$). Starting from the task with the highest priority, we try to increase the maximum number of direct blocking due to the access of each resource for the task under consideration. The increasing of the maximum number of direct blocking for a task $\tau_i$ due to access over a resource $R_p$, i.e., $\mathrm{CT}(\tau_i, R_p)$, can be done if the resulting amount for priority inversion (because of direct or transitive blocking) is still no more than the maximum blocking time tolerable to $\tau_i$. Note that the proposed algorithm could be simply modified to set up the ceiling table for BCCP by changing all entry values over one into "*".

## 4 Deadlock prevention

The flexibility of adjusting the maximum priority inversion number for each task introduces potential transitive blocking and deadlocks. This section presents a simple deadlock prevention approach for BCCP or ECCP.

A resource allocation graph (Fig. 5), is used to model the dependencies among tasks. When a task may access a resource, then an edge exists between the corresponding vertices in the graph. Notably, a resource allocation graph is a bipartite graph, where vertices of the same type reside on the same side. A request edge $(\tau_i \rightarrow R_p)$ denotes that task $\tau_i$ is blocked in requesting of resource $R_p$. An allocation edge $(R_p \rightarrow \tau_i)$ indicates that task $\tau_i$ currently locks resource $R_p$. The ceiling table settings of ECCP (and BCCP) did not prevent a run-time waiting cycle, such as $(\tau_1 \rightarrow R_1 \rightarrow \tau_2 \rightarrow R_2 \rightarrow \tau_4 \rightarrow R_3 \rightarrow \tau_1)$, from occurring. No active resource should be involved in the considerations of deadlocks, because a task that self-suspends itself to wait for service completion of some active resource could not issue another request for some other resource. Notably, although this work focus on synchronous I/O, the above characteristics of deadlocks remain even when an asynchronous I/O is presented. An asynchronous I/O automatically releases the involved active resource once the corresponding service is complete.




| Task/Resource | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $\tau_1$ | 3 | 0 | 1 | 0 | 0 |
| $\tau_2$ | 1 | 2 | 0 | 0 | 0 |
| $\tau_3$ | 0 | 0 | 0 | 2 | 3 |
| $\tau_4$ | 1 | 1 | 1 | 0 | 1 |

**Fig. 5** The ceiling table and the corresponding resource allocation graph of an example system

Deadlock avoidance in PCP is achieved by having a sufficiently high ceiling for a resource in a potentially formed wait cycle such that no task could lock the last resource needed in forming the wait cycle. An example shown in Fig. 6 is used to illustrate. Let resource $R_1$ be locked by task $\tau_3$ at time $t$, and $\tau_3$ requests $R3$. Under PCP, task $\tau_1$ has no chance of locking resource $R_2$ successfully after time $t$ (because the ceiling of $R_1$ is not less than the priority of $\tau_1$) and later requests to lock $R_1$. As astute readers might point out, the ceiling of a resource under BCCP or ECCP might be lower than that of the corresponding resource under PCP such that a deadlock might be formed.

To prevent deadlocks, two naive approaches should be considered: (1) a dynamic adjustment mechanism for resource ceilings to prevent any deadlock in an on-line fashion; and (2) a proper and off-line setting for the ceiling tables for "critical resources" such that no deadlocks could occur. The first approach is costly and may not be suitable for BCCP and ECCP because it impacts on manipulations of the ceiling table. This section focus on the second approach as it only involves off-line effort in picking-up certain resources. After revising the ceiling table for selected resources, BCCP and ECCP operate as defined in previous sections, and their properties remain.

The idea is to pick up two resources per cycle in a resource allocation graph (e.g., $R_1$ and $R_3$ in Fig. 6) as critical resources such that these two resources are not locked by two different tasks at the same time. This could be achieved by setting the ceilings of these two resources as the maximum priority of the tasks that might access these two resources. However, two technical issues must be addressed: (1) how to revise a given ceiling table for BCCP/ECCP to achieve the above objective; and (2) how to select a minimum collection of resources such that two resources always appear in a cycle in the graph. Notably, one should not try to identify every cycle in the graph because the number of cycles may be an exponential number of the number of vertices.

**Theorem 3** *If two distinct resources in each cycle are not locked by two different tasks simultaneously, then the system has no deadlock.*

*Proof* Since there would be no wait-for cycle of tasks, no deadlock exists. □

This work first address the first technical issue, i.e., how to revise a given ceiling table for BCCP/ECCP: Let $R_p$ and $R_q$ be two resources selected in a cycle, and $\tau_i$ and $\tau_j$ be the highest-priority tasks with $\mathrm{CT}(\tau_i, R_p) \geq 1$ and $\mathrm{CT}(\tau_j, R_q) \geq 1$, respectively. Without the loss of generality, let the priority of $\tau_i$ be higher than that of $\tau_j$. Revising the ceiling table can be accomplished simply by setting both $\mathrm{CT}(\tau_i, R_p)$ and $\mathrm{CT}(\tau_i, R_q)$ as 1.

The second technical issue is minimizing the number of selected critical resources. This issue could be further refined to consider the effects of selection because the ceiling of each selected resource would be higher, compared to the original BCCP/ECCP settings. An elevated resource ceiling can result in a low concurrency level in the system. This issue can be refined to formally define a technical problem, where a weight is given to each resource serving as a critical resource:

**Definition 2** (WEIGHTED 2-VERTEX FEEDBACK SET) Given an undirected graph $G = (V, E)$, and a positive integer $K$, let $w(v) \in Z^+$ for each $v \in V$. The question is whether a subset $V' \subset V$ exists, for each cycle in $G$ which contains at least two elements in $V'$ and $\sum_{v \in V'} w(v) \leq K$.

**Lemma 3** *WEIGHTED 2-VERTEX FEEDBACK SET is NP-hard.*

*Proof* The NP-hardness is demonstrated by reducing any instance of the VERTEX COVER problem to an instance of the WEIGHTED 2-VERTEX FEEDBACK SET problem. The VERTEX COVER problem is defined as follows (Garey and Johnson 1979): Given a undirected graph $G = (V, E)$ and a positive integer $N$, does a set $V' \in V$ and $|V'| \leq N$ exist such that $V'$ contains at least one vertex for each edge in $E$?

For the graph $G = (V, E)$ which is an instance of VERTEX COVER problem, this study construct a new graph $G^* = (V^*, E^*)$ for the instance of WEIGHTED 2-VERTEX FEEDBACK SET problem as follows. One new vertices $v^*_{i,j}$ is added with two new edge to each edge $(v_i, v_j) \in E$. Two new edges $(v^*_{i,j}, v_i)$ and $(v^*_{i,j}, v_j)$ are added and let $w(v_i)$, $w(v_j)$, and $w(v^*_{i,j})$ be $(|E| + 1)$, $(|E| + 1)$, and 1, respectively. In other words, each edge in $E$ is now transformed into a triangle in $E^*$. Let the cost constraint $K$ for the WEIGHTED 2-VERTEX FEEDBACK SET problem be $N * (|E| + 1) + |E|$. Notably, $N$ denotes the maximum number of vertices that can be selected for the VERTEX COVER problem.[3]

Suppose that the algorithm can solve any problem instance of the WEIGHTED 2-VERTEX FEEDBACK SET (i.e., two vertices are selected for each cycle in $G^*$ and the total weight of the selected vertices does not exceed $K$). Each triangle in $G^*$ must then have at least two vertices selected. Thus, every edge in $G$ will have at least one vertex selected, and any cycle of length no shorter than 2 in $G$ will have at least 2 vertices selected. Among all vertices in $G^*$ of weight $|E| + 1$ (which also present in $G$), at most $N$ can be selected because the cost constraint of the WEIGHTED 2-VERTEX FEEDBACK SET is $K = N * (|E| + 1) + |E|$. Thus, the corresponding instance of VERTEX COVER with $G$ and $N$ is solved.

Conversely, if the algorithm can not find a set $V^{*\prime} \in V^*$ such that each cycle in $G^*$ has at least two vertices selected and the total cost of the selected vertices in $V^{*\prime}$ exceeds $K = N * (|E| + 1) + |E|$, then all newly added vertices for each triangle in $G^*$ must be already included in $V^{*\prime}$ because their cost is 1 (and their total cost is $|E|$). Since every newly added vertex in $G^*$ is already selected, this study selects one more vertex that already exists in $G$ to ensure that every triangle in $G^*$ has at least two vertices selected. In other words, one has to cover each edge in $G$ and the total cost of the selected vertices does not exceed $N * (|E| + 1)$. The conditional equivalents only selecting $N$ nodes in $G$ to cover every edge in $G$. Consequently, the corresponding VERTEX COVER with $G$ and $N$ cannot be solved, either. □

Apparently, the WEIGHTED 2-VERTEX FEEDBACK SET is NP-hard by reducing any instance of the VERTEX COVER problem to an instance of the WEIGHTED

[3]Without loss of generality, we ignore the graphs which contains cycles of length 1 and 2. In particular, a cycle of length 2 can be treated as a single edge for VERTEX COVER problems.

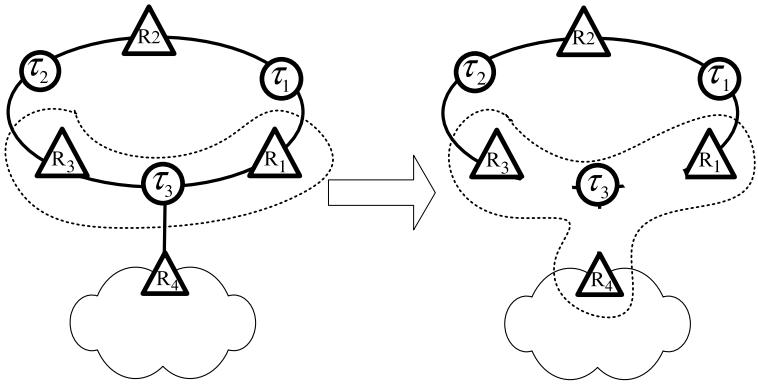



**Fig. 6** The selection for critical resources and the removing of edges in a resource allocation graph

2-VERTEX FEEDBACK SET problem. Furthermore, since for a resource allocation graph only has resources that can be selected as critical resources, it must be a bipartite graph. Based on the NP-hardness of the WEIGHTED 2-VERTEX FEEDBACK SET, this work defines the RESTRICTED WEIGHTED 2-VERTEX FEEDBACK SET FOR BIPARTITE GRAPHS as follows:

**Definition 3** (RESTRICTED WEIGHTED 2-VERTEX FEEDBACK SET FOR BIPARTITE GRAPHS) Given an undirected and bipartite graph $G = (V_1, V_2, E)$, and a positive integer $K$, let $w(v) \in Z^+$ for each $v \in V_1 \cup V_2$. Does a set $V' \subset V_1$ exist, for each cycle in $G$ that contains at least two elements in $V'$, and $\sum_{v \in V'} w(v) \leq K$.

**Theorem 4** *RESTRICTED WEIGHTED 2-VERTEX FEEDBACK SET FOR BIPARTITE GRAPHS is NP-hard.*

*Proof* Consider an instance of the WEIGHTED 2-VERTEX FEEDBACK SET with an undirected graph $G = (V, E)$ with a cost constraint $K$. This work transforms $G$ into an undirected bipartite graph $G^* = (V_1^*, V_2^*, E^*)$ as follows. Notably, $V_1^*$ is an exact copy of $V$, and each vertex in $V_1^*$ weights the same as it does in $V$. For each edge $(v_i, v_j) \in E$ and the corresponding vertices $v_i^*$ and $v_j^*$ in $V_1^*$, a new node $v_{i,j}^*$ of weight $K + 1$ is created in $V_2^*$ for connecting nodes $v_i^*$ and $v_j^*$. In other words, every edge $(v_i, v_j)$ in $E$ is transformed into two adjacent edges $(v_i^*, v_{i,j}^*)$ and $(v_{i,j}^*, v_j^*)$ in $E^*$. Thus, the connectivity of vertices in $V$ is exactly the same as that of vertices in $V_1^*$. Let $K^*$ the cost constraint of the RESTRICTED WEIGHTED 2-VERTEX FEEDBACK SET FOR BIPARTITE GRAPHS be $K$ (which is the cost constraint of the WEIGHTED 2-VERTEX FEEDBACK SET). Because selecting any vertex in $V_2^*$ violates the cost constraint, only vertices in $V_1^*$ can be selected. Thus, an algorithm solves any instance of the RESTRICTED WEIGHTED 2-VERTEX FEEDBACK SET FOR BIPARTITE GRAPHS with graph $G^* = (V_1^*, V_2^*, E^*)$ and cost constraint $K^*$ can solve the corresponding instance of the WEIGHTED 2-VERTEX FEEDBACK SET with graph $G = (V, E)$ and cost constraint $K$. □

The heuristics for selecting the critical resources can be done based on the spanning trees concept: Given a resource allocation graph $G$, the algorithm starts with another graph $G'$ with the same set of vertices for $G$ but without any edges. One edge at a time is moved from $G$ to $G'$ in the same manner as when growing a spanning tree. Once a cycle is detected in $G'$, two arbitrary resources are selected for the cycle as critical resources (Notably, only one cycle may exist because a spanning tree is under construction). This work removes all edges in $G'$ and $G$ and in the path between the two selected resources. Any resource not in the path but connected to some task in the path in $G$ is also selected as a critical resource. Whenever such a critical resource is selected, any edge connecting the newly selected critical resource and any task on the path is removed from $G'$ and $G$ (see example in Fig. 6). The above procedure is repeated until no edges exist in $G$. The time complexity of this heuristics algorithm is $O(|V|^2 * |E| \log |E|)$, where $V$ is the number of vertices and $E$ is the number of edges in the given resource allocation graph.

## 5 Performance evaluation

### 5.1 Experimental setup

This section provides evaluation of the proposed BCCP/ECCP. We are to show that, with proper settings of the ceiling table, tasks can have good response. We shall focus our discussion on ECCP because active resources are considered in experiments. For comparison, all the task sets generated are evaluated with PCP in a restrictive way. With PCP, when a higher-priority task suspends itself to access an active resource, the lower-priority task can not execute over the CPU, and active resources are managed as if they were passive resources.

Synthesized workloads are used in our experiments. The number of tasks per task set is randomly chosen between 5 and 10. Tasks are purely periodic, and task periods are randomly picked form [2000 ms, 10000 ms]. The CPU utilization of each task is between 8% and 10%, and the total CPU utilization of each task set is between 50% and 70%. The first arrivals of tasks are arbitrary. The number of passive resources a task set is between 5 and 10. The collection of resources that a task will request is randomly chosen, but a task with a high CPU utilization will require a larger collection of resources. If a task accesses an active resource, then the duration of the access is between 5% and 10% of the length of its task period. The number of active resources a task set has is between 0 to 9. A task with a long task period will try to lock many active resource. In the experiments, 1000 task sets are generated for each system configuration. Each simulation run of a task set stops by the end of the least common multiple of all its task periods.

The ceiling table is configured with two different patterns, namely PATTERN-I and PATTERN-II. For PATTERN-I, there are three different settings: 1/4, 1/2, and 1/1, each of which means that how many high-priority tasks sacrifice their privileges of resource access by setting all their ceiling-table entries as "*" (or 2). 1/4 stands for that the first one forth high-priority tasks are sacrificed. For PATTERN-II, there are also three different settings: 1/4, 1/2, and 1/1, each of which means how many

resources a task voluntarily gives up its access privileges by setting "*" (or 2) in the corresponding table entry of those resources.

Our performance metrics are *Average Response Ratio* and *Longest Response Ratio*. The Average Response Ratio of a task set and a ceiling table is defined as the ratio of the average task response with BCCP/ECCP to that with PCP. The higher the system utilization is, the smaller Average Response Ratio will be. The Longest Response Ratio of a task set is accordingly defined with respect to the longest task response of the task set. The Longest Response Ratio is to observe whether small degradation of high-priority-task response can be effectively traded for significant improvement of low-priority-task response.

## 5.2 Experimental results

Figures 7(a) and 7(b) show the Average Response Ratio of BCCP/ECCP with PATTERN-I and PATTERN-II, respectively. The X axis is the number of shared active resources, and the Y axis is the Average Response Ratio. The smaller Average Response Ratio is, the higher the system utilization will be. Note that, if there is no active resource, ECCP becomes BCCP. As shown in Fig. 7, when there are nine different active resources, the Average Response Ratio of BCCP/ECCP can reach 0.8, which is 80% of the Average Response Ratio of PCP (which is 1.0).

In Fig. 7(a), we can see that curve "1/1" is always the lowest and curve "1/4" is always the highest (please see the previous section for the definitions of "1/1", "1/2", and "1/4" with respect to PATTERN-I). That is because system utilization is improved if there are many high-priority tasks sacrifice their privileges of resource access. It is also true when the number of active resources is large, and therefore all the curves gradually drop as the total number of active resources increases. In Fig. 7(b), it is shown that, curve "1/1" and curve "1/4" are always the lowest and the highest, respectively. Which means, if access privileges of many resources are voluntarily relinquished, then system utilization can be improved. However, as in Fig. 7(b) the access privilege of a resource is not necessarily relinquished by a high-priority task, performance improvement in Fig. 7(b) is not as good as that in Fig. 7(a).

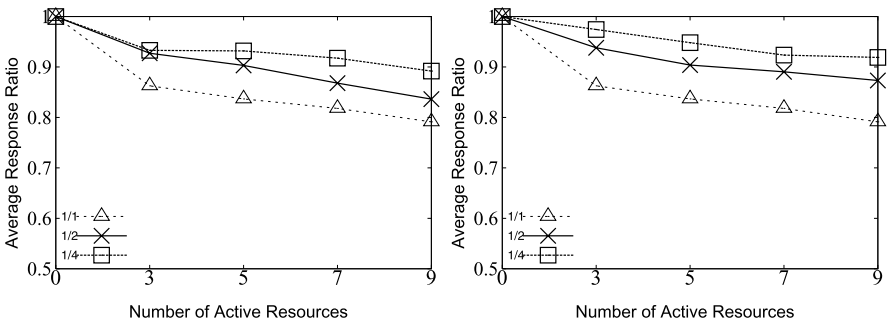



**Fig. 7** Average response ratio of BCCP/ECCP with different patterns for ceiling-table setting

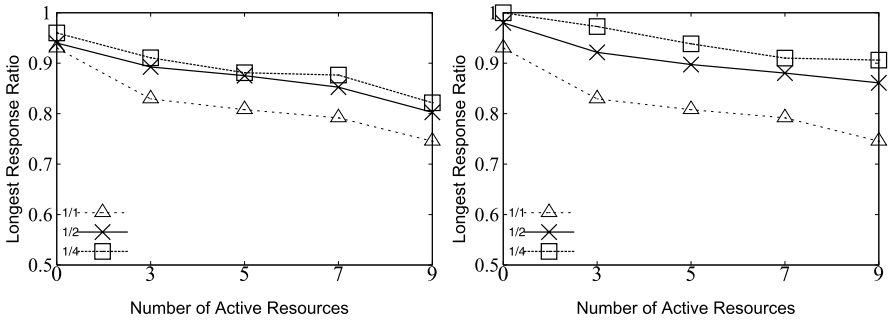



**Fig. 8** Longest response ratio of BCCP/ECCP with different patterns for ceiling-table setting

Figures 8(a) and 8(b) show the Longest Response Ratio of BCCP/ECCP with PATTERN-I and PATTERN-II, respectively. The X axis is the number of shared active resources, and the Y axis is the Longest Response Ratio. The figure shows that, with BCCP/ECCP, the Longest Response Ratio can achieve 0.75, which is better than the achievable Average Response Ratio (i.e., 0.8). That is because, under BCCP/ECCP, resource-access privileges of high priority tasks can be traded for responsiveness of low-priority tasks. We can also noted that, even though there is no active resource (in this case ECCP is BCCP), the Longest Response Ratio of BCCP/ECCP is still smaller than 1.0 (recall that PCP is the baseline).

## 6 Conclusion

This paper is motivated by the needs of resource synchronization protocols to accommodate different priority inversion requirements of different tasks. We consider situations in which tasks might voluntarily give up the CPU and suspend themselves until completing I/O operations. This paper proposes configurable resource synchronization protocols for engineers to adjust the maximum number of priority inversions for each task. A table-based approach is first proposed to adjust the maximum numbers of priority inversions for tasks without suspension. We then extend the approach to considerations of task suspension. The ceilings of resources become configurable to allow lower-priority tasks to grab resources while higher-priority tasks suspend themselves to wait for I/O operations. An algorithm is proposed to assign values to the ceiling table, given the maximum number of priority inversions for each task. System utilization is traded with the maximum numbers of priority inversions for different tasks. A deadlock-prevention method with low run-time overheads is proposed to avoid deadlocks. The minimum cost problem in breaking deadlock cycles is also proven to be NP-hard.

For the future research, we will further exploit resource synchronization issues when the performance of processors and I/O devices might change over time (due to technology advances or the adoption of different hardwares). We must point out that when the performance of processors or I/O devices improves, a schedulable schedule

might become unschedulable (Stankovic et al. 1995). Methodologies might be needed to enforce the schedulability of tasks even if the advance of hardware technology boosts the performance of processors or I/O devices.

## Appendix: Algorithms

### BCCP_MAX_BLOCKING_TIME

(Tasks are sorted by their priorities, let $\tau_0$ be the highest priority task.)
1: $n$: the number of tasks
2: $m$: the number of resources
3: CT′: the revised ceiling table based on CT
4: **BCCP_MAX_BLOCKING_TIME**($\tau_i$)
5: {
6: $U_i$: a two-dimensional array: $U_i[0..(n-i-1)][1..m]$
7: $U_i[0] \longleftarrow \text{CT}'(\tau_i)$ // $U_i[0]$ is set as the resource usage of $\tau_i$
8: for ($x = 1; x \leq (i-1); x++$) {
9: $U_i[x] \longleftarrow$ Inheritance($U_i[x-1]$, CT′($\tau_x$)) // find out all possible inheritance blocking
10: for ($x = i; x \leq n-1; x++$) {
11: $U_i[x] \longleftarrow$ Transitive($U_i[x-1]$, CT′($\tau_{x+1}$)) // find out all possible transitive blocking
12: }
13: $B_{\tau_i}$ = Blocking_time($U_i[x]$) // calculate the maximum total blocking time
14:}

15:**Blocking_time**($U_i[z]$)
16:{
17: $B_{\tau_i}$: the maximum total blocking time imposed on $\tau_i$ (initially zero).
18: MB: the maximum blocking time imposed on $\tau_i$ in PCP.
// it could be derived from the total blocking time caused by direct blocking
// and priority ceiling blocking as same as that in PCP
19: $\text{BR}_p$: the longest duration of all tasks on accessing $R_p$ (constant).
20: for ($p = 1; p \leq m; p++$) { // $p$-th column corresponds to resource $R_p$
21: if $U_i[z][p] = *$
22: $B_{\tau_i} = B_{\tau_i} + BR_p$ // For all entries marked as "1" we only have to find out the maximum blocking time
23: }
24: $B_{\tau_i} = B_{\tau_i} + MB$
25: return $B_{\tau_i}$
26:}

27:**Inheritance**($U_i[y]$, CT′($\tau_a$))
28:{
29: $Q_x$: an one-dimension array of length $m$
30: $Q_x \longleftarrow$ [0...0]
31: for ($q = 1; q \leq m; q++$) {
32: if ( ($U_i[y][q] == 0$) and (CT′($\tau_a$, $R_q$) == $*$) )
33: $Q_x[q]$="*" // in this case, blocking imposed on $\tau_a$ could be a inheritance blocking to the current task.
34: else
35: $Q_x[q] = U_i[y][q]$
36: } // for
37: return $Q_x$
38 }

39:**Transitive**($U_i[y]$, CT′($\tau_j$))
40:{
41: $Q_x$: an one-dimension array of length $m$
42: $Q_x \longleftarrow$ [0...0]
43: intersection: a temporary flag
44: intersection = 0
45: for ($p = 1; p \leq m; p++$)
// check if $\tau_j$ and ($\tau_i ....\tau_{j-1}$) has some shared resource
46: if (($U_i[y][p] \neq 0$) and (CT′($\tau_j$, $R_p$) $\neq 0$))
47: intersection = 1

```
48:   if (intersection == 1) {
49:     for (q = 1; q ≤ m; q++) {
50:       if ( (U_i[y][q] == 0) and (CT'(τ_j, R_q) == *) )
51:           Q_x[q] = "*"     // in this case, blocking imposed on τ_j could be transitively propagated to the current task.
52:       else
53:           Q_x[q] = U_i[y][q]
54:     } // for
55:   }// if
56:   return Q_x
57 }
```

## ECCP_MAX_BLOCKING_TIME

```
(Tasks are sorted by their priorities and let τ_0 be the highest priority task.)
1: n: the number of the tasks
2: m: the number of the passive resources
3: M: the number of the active resources
4: CT': the revised extended ceiling table based on CT
5: ECCP_MAX_BLOCKING_TIME(τ_i)
6: {
7:    U_i: a two-dimensional array: U_i[0..(n − i − 1)][1..m]
8:    U_i[0] ⟵ CT'(τ_i)     // U_i[0] is set as the resource usage of τ_i
9:    for (x = 1; x ≤ (i − 1); x++) {
10:     U_i[x] ⟵ Inheritance(U_i[x − 1], CT'(τ_x))     // find out all possible inheritance blocking
11:   for (x = i; x ≤ n − 1; x++) {
12:     U_i[x] ⟵ Transitive(U_i[x − 1], CT'(τ_{x+1}))     // find out all possible transitive blocking
13:   }
14:   B_{τ_i} = Blocking_time(U_i[x])     // calculate the maximum total blocking time
15:}

16:Blocking_time(U_i[z])
17:{
18:   B_{τ_i}: the maximum blocking time imposed on task τ_i (initially zero).
19:   MB: the maximum blocking time imposed on τ_i in PCP.
      // it could be derived from the total blocking time caused by direct blocking
      // and priority ceiling blocking as same as that in PCP
20:   BR_p :the maximum duration in the locking of passive resource R_p of all tasks(constant)
21:   BR^a_g :the maximum duration of accessing on active resource R^a_g
      by corresponding sub-job among all tasks(constant)
22:   for (p = 1; p ≤ m; p++) {     // p-th column corresponds to resource R_p
23:     if (U_i[z][p] > 1)
24:       B_{τ_i} = B_{τ_i} + (U_i[z][p] − 1)*BR_p
          // each entry with value larger than one could contribute CT'(τ_i, R_p) − 1 priority inversions
25:   }
26:   B_{τ_i} = B_{τ_i} +MB
27:   for (g = 1; g ≤ M; g++)
28:     if (τ_i uses R^a_g)
29:       B_{τ_i} = B_{τ_i} + BR^a_g
30:   return B_{τ_i}     // an active resource contributes one additional priority inversion
31:}

32:Inheritance(U_i[y], CT'(τ_a))
33:{
34:   Q_x: an one-dimension array of length m
35:   Q_x ⟵ [0...0]
36:   for (q = 1; q ≤ m; q++) {
37:     if ( (U_i[y][q] == 0) and (CT'(τ_a, R_q) > 1) )
38:       Q_x[q] = CT'(τ_a, R_q)
          // in this case, blocking imposed on τ_a could be a inheritance blocking to the current task.
39:     else
40:       Q_x[q] = U_i[y][q]
41:   } // for
42:   return Q_x
43 }

44:Transitive(U_i[y], CT'(τ_j))
```

```
45:{
46:   Q_x : an one-dimension array of length m
47:   Q_x ⟵ [0...0]
48:   intersection: a temporary flag
49:   intersection = 0
50:   for (p = 1; p ≤ m; p++)
      // check if τ_j and (τ_i ....τ_{j-1}) has some shared resource
51:     if ((U_i[y][p] ≠ 0) and (CT'(τ_j, R_p) ≠ 0))
52:       intersection = 1
53:   if (intersection == 1) {
54:     for (q = 1; q ≤ m; q++) {
55:       if ( (U_i[y][q] == 0) and (CT'(τ_j, R_q) > 1) )
56:         Q_x[q] = CT'(τ_j, R_q)
            // In this case, blocking imposed on τ_j could be transitively propagated to the current task.
57:       else
58:           Q_x[q] = U_i[y][q]
59:     } // for
60:   } // if
61:   return Q_x
62:}
```

## SETTING_ECCP_CEILING_TABLE

(Tasks are sorted by their priorities and let $\tau_0$ be the highest priority task.)
1: $n$: the number of the tasks
2: $m$: the number of the passive resources
3: $M$: the number of the active resources
4: CT: the ceiling table(Initially all entries are zero)
5: FR[1..$n$][1..$m$]: FR[$i$][$p$] denotes how many times $\tau_i$ requests $R_p$ in each its period
6: RU[1..$n$][1..$m$]: RU[$i$][$p$] denotes whether $\tau_i$ uses $R_p$ or not.
7: RBT[1..$n$]: the remaining duration of tolerable blocking time for each task.
//the initial values in RBT are derived from a schedulability analysis.
8: $\mathrm{BR}_p$: the maximum duration in the locking of passive resource $R_p$ of all tasks
9: $\mathrm{BR}_g^a$: the maximum duration of accessing on active resource $R_g^a$ by the corresponding sub-job among all tasks

```
10:SETTING_ECCP_CEILING_TABLE
11:{
12:   fr_i[1..m], CR_i[1..m];
13:   MB_i : the maximum blocking time imposed on τ_i in PCP.
      // it could be derived from the total blocking time caused by direct blocking
      // and priority ceiling blocking as same as that in PCP
14:   for (i = 1; i ≤ n; i++) {
15:     CT(τ_i) ⟵ RU[i]; CR_i ⟵ FR[i];
16:     fr_i ⟵ Sum_Usage_Frequency(τ_i);
        // for the entries marked as "1" 's and the active resource
17:     for (g = 1; g ≤ M; g++)
18:       if (τ_i uses R_g^a)
19:         RBT[i] = RBT[i] − BR_g^a
20:     RBT[i] = RBT[i] − MB_i;
21:     while ((RBT[i] > 0) and (fr_i != [0...0])) {
22:       R_p = the most frequently used resource in fr_i;
23:       if ((CT(τ_i, R_p) >= 1) and (RBT[i] > BR_p) and (CR_i[p] > 0) and (fr_i[p] > 0)) {
            // τ_i could tolerate one more priority inversion on accessing R_p.
24:           if (!∃ ((CT(τ_x, R_p) == 0) and (RBT[x] <BR_p)|x = 1...i − 1)) {
                // check if all higher priority tasks could tolerate this blocking which is
                transitively propagated to them.
25:               subtract BR_p from RBT[x] where (CT(τ_x, R_p) == 0) for x = 1...i − 1;
26:               CT(τ_i, R_p)++; RBT[i] = RBT[i]- BR_p; fr_i[p]−−; CR_i[p]−−;
27:           } // if (CT(τ_i, R_p) >= 1)
28:           else
29:             fr_i[p] = 0;
30:       }
31:       else {
32:           if ((RBT[i] <BR_p) or (CR_i[p] == 0)) // no more blocking is allowed on accessing R_p
33:             fr_i[p] = 0;
34:       } // else
```

```
35:       }// while
36:    }// for
37:}

38:Sum_Usage_Frequency(τ_i)
39:{ // calculate the number of requests on each resources made by low priority tasks (compared to τ_i)
40:    fr[1..m] ⟵ [0...0];
41:    for (p = 1; p ≤ m; p++)
42:       for (k = i; k ≤ n; k++)
43:          fr[p] = FR[k, p]+ fr[p];
44:    return fr
45:}
```

## References


Baker TP (1990) A stack-based resource allocation policy for real-time process. In: IEEE 11th real-time system symposium, 4–7 Dec 1990

Chen M, Lin KJ (1990) Dynamic priority ceiling: a concurrency control protocol for real-time systems. Real-Time Syst 2(4)

Garey MR, Johnson DS (1979) Computers and intractability

Han C-C, Lin K-J, Hou C-J (1996) Distance-constrained scheduling and its applications to real-time systems. IEEE Trans Comput, July 1996

Hsueh C-W, Lin K-J (1998) On-line schedulers for pinwheel tasks using the time-driven approach. In: 10th euromicro on real-time systems, June 1998, pp 180–187

Kuo T-W, Huang G-H, Ni S-K (1999) A user-level computing power regulator for soft real-time applications on commercial operating systems. J Chin Inst Electr Eng 6(1):13–25

Kuo T-W, Liang M-C, Shu LC (2001) Abort-oriented concurrency control for real-time databases. IEEE Trans Comput (SCI) 50(7)

Liang M-C, Kuo T-W, Shu LC (2003) A quantification of aborting effect for real-time data accesses. IEEE Trans Comput (SCI) 52(5)

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. J Assoc Comput Mach 20(1), 46–61

Rajkumar R, Sha L, Lehoczky JP (1988) Real-time synchronization protocols for multiprocessor. In: IEEE real-time systems symposium

Sha L, Rajkumar R, Lehoczky JP (1990) Priority inheritance protocols: an approach to real-time synchronization. In: IEEE Trans Comput 39(9)

Sides DJ (1995) A dynamically adaptive real-time data acquisition and display system. In: IEEE real-time technology and applications symposium, May 1995

Spuri M, Buttazzo G, Sensini (1995) Scheduling aperiodic tasks in dynamic scheduling environment. In: IEEE real-time systems symposium

Stankovic JA, Spuri M, Di Natale M, Buttazzo GC (1995) Implications of classical scheduling results for real-time systems. IEEE Comput 28(6)

Stoica I, Abdel-Wahab H, Jeffay K, Baruah SK, Gehrke JE, Plaxton CG (1996) A proportional share resource allocation algorithm for real-time, time-shared systems. In: IEEE real-time systems symposium, pp 288–299

Waldspurger CA (1995) Lottery and stride scheduling: flexible proportional-share resource management. PhD thesis, technical report, MIT/LCS/TR-667, Laboratory for CS, MIT, September 1995

Waldspurger CA, Weihl WE (1995) Stride scheduling deterministic proportional share resource management. Technical memorandum MIT/LCS/TM-528. Laboratory for CS, MIT, July 1995

Wang K, Lin T-H (1994) Scheduling adaptive tasks in real-time systems. In: IEEE 15th real-time systems symposium, December 1994

Wu J, Kuo T-W, Hsueh C-w (1999) RCPCP: a ceiling-based protocol for multiple-disk environments. In: 5th international conference on real-time computing systems and applications

**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** is Assistant Professor at Department of Computer Science, National Chiao-Tung University, Taiwan. His research interest is in operating systems, storage systems, and real-time system. He had served on the editorial board of Journal of Signal Processing Systems (Springer, SCI-E), and technical committees of international conferences, including ACM Symposium on Applied Computing (ACM SAC), IEEE Real-Time Systems Symposium (IEEE RTSS), IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (IEEE RTCSA), and IEEE/IFIP International Conference On Embedded and Ubiquitous Computing (EUC).