Hindawi
Scientific Programming
Volume 2017, Article ID 3403685, 20 pages
https://doi.org/10.1155/2017/3403685

Hindawi

Research Article

A Design That Incorporates Adaptive Reservation into Mixed-Criticality Systems

Fei Guan, Long Peng, Luc Perneel, Hasan Fayyad-Kazan, and Martin Timmerman

Electronics and Informatics Department, Vrije Universiteit Brussel (VUB), Pleinlaan 2, 1050 Brussel, Belgium

Correspondence should be addressed to Fei Guan; fei.guan@vub.ac.be

Received 18 September 2016; Revised 22 December 2016; Accepted 4 January 2017; Published 1 February 2017

Academic Editor: Basilio B. Fraguela

Copyright © 2017 Fei Guan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper presents a design and implementation of a Mixed-Criticality System (MCS) extended from μC/OS III. It is based on a MCS model that uses an adaptive reservation mechanism to cope with the uncertainties in task execution times and to increase the resource utilization in MCS. The implementation takes advantage of the tasks' recent execution records to predict their required computational resource in the near future and adjusts their reserved budget according to their criticality levels. The designed system focuses on soft real-time tasks. An overrun tolerance algorithm is used to limit the deadline miss ratios between a rise to the task's actual consumption and the change to the amount of reservation. More than two criticality levels can be handled without introducing obvious additional overhead at each added level. The case study evaluation demonstrates that the reserved resource for each task is always close to its actual consumption; the tasks' deadline misses are bounded by the different requirements specified by the criticality levels; during overload conditions, high-criticality tasks are guaranteed to have sufficient resource reservation. Although there is still room for improvement if it comes to processing overhead, this research brings some inspirations in both modelling and implementation aspects of MCS.

1. Introduction

Embedded systems are in high numbers around us today. They range from smart watches to flight control systems. Among different types of embedded systems, some of them do not permit for malfunctions as they could harm human lives or the environment. Examples of these so called safety-critical systems can be found in medical devices, aviation, and transport industry. In the related industry standards, the safety-critical functions are classified into multiple criticality or safety levels in relation to the hazard they generate by their failure (e.g., DO-178B, IEC 61508). The concept of a Mixed-Criticality System (MCS) is therefore defined and aims at a software platform that allow functions with different criticality levels to coexist on a shared hardware platform [1].

The goal of a MCS is to only provide high levels of reliability to the sole functions requiring it. A fair amount of research has been devoted to this area. One comprehensive overview is addressed in [2]. In general, three fundamental research issues are considered important: the resource isolation, the resource distribution and sharing among the isolated parts, and the balance between isolation and sharing. A wide range of resource kinds are involved in these topics, including power and energy [3–8], communication bus and memory access [9–20], and processor usage [21–42] in both the single- and multiprocessors platforms.

The research in this paper is limited to the temporal isolation and sharing problem on a single processor platform. It has been a major focus in MCS research. The research history in this area can be traced back to Vestal's paper [1]. It was followed by a series of publications that covered both processor resource distribution schemes with fixed priority (FP) [24, 26–29, 31–39] and earliest deadline first (EDF) [21–23, 25, 40–42] based scheduling in MCSs.

Not all of these studies use the same system model but their defined models share some common features. In general, a task is characterized by multiple WCET values with different levels of confidence. In order to avoid severe underutilization of the resource, every task in the system executes with a budget that equals its low-confidence WCET at the beginning. Only when a given task with a high-criticality overruns does the system raise the reservation of high-criticality tasks

to their high-confidence WCETs and triggers operations such as discarding low-criticality tasks, extending their deadlines, lowering their priorities, or shrinking their execution times.

These models have certain limitations. The first issue is related to WCET assessment. For time-randomised hardware, defining different WCET budget with different confidence levels can be done with probabilistic WCET (pWCET) [44, 45]. But for time-deterministic hardware the existing WCET estimation techniques (both static and measurement-based techniques) can not associate the confidence levels with the WCET estimates in an easy way [2]. Although recent studies [46, 47] present that the pWCET can also be used for time-deterministic hardware, the technique is immature and limited to very simple systems [48]. Even for the time-randomised platform which has the best fit for pWCET techniques, there is a lack of trustworthiness of the provided results due to their strong dependence on the assumptions made in the measurement, particular hardware features, or a number of other aspects [48]. Moreover, if the target task's execution time varies a lot from one instance to another, which is highly possible due to the impact of unpredictable environment and high complexity of the system, using offline WCET estimates as a task's bandwidth budget can cause low resource usage [36, 49–51].

The second issue is about a pessimistic assumption that is made in the general MCS models. In [52], Burns classifies the event that a task executes longer than its expected WCET as a fault. As defined in these models, all high-criticality tasks' executions will consume up to high-confidence WCET values after one high-criticality task encounters such a fault. As pointed out by Easwaran and Shin [53], this is highly unlikely to happen in practice if these tasks are independent of each other. Therefore, the paralysation or service degradation of low-criticality tasks based on this assumption is sometimes overdone.

The third issue is that the traditional models do not take into account the fact that at certain criticality level, sometimes even the highest ones, the tasks can tolerate some deadline misses. In a dual-criticality system, for example, a high-criticality task should never miss a deadline according to the classical assumption. However, as stated in [54], the industry standards actually do not have specific definitions for the relationship between deadline misses and critical failures. It conceivable that a small number of deadline misses has little or no effect on safety. If this is indeed the case, then avoiding all the deadline misses is certainly too pessimistic and causes a waste of resource.

In this paper, we design and implement a software framework extended from μC/OS III. It introduces adaptive reservation concept into the area of MCSs. Considering the issues mentioned, the designed system supports a MCS model different from the existing ones. It allows tasks to have deadline misses within certain limits but still based on the conjecture that the strictness of the timing constraints lines up with the criticality levels of the tasks. Instead of reserving resource according to several offline WCET estimates, a prediction mechanism is adopted to calculate the required computation resource for each task individually during runtime. Whether the tasks' requirements will be fulfilled is determined by the admission control according to their criticality levels and the system's available resource. In the overload condition, tasks are sacrificed in a reverse order of criticality levels to release resource and maintain a schedulable system. Until the prediction mechanism reacts to a task's resource consumption change, a slack (reserved but unused processor bandwidth) distribution algorithm is adopted to handle the budget overrun events and lower the risk of deadline miss.

The organization of this paper is as follows. Section 2 provides an overview of related work; Section 3 introduces notions and the MCS model adopted in this paper; Sections 4 and 5 describe all the services provided by the designed system and the basic structure used for task isolation; Sections 6 to 10 explain the implementation details of these services; Section 11 presents the case study evaluation results for this framework; and Section 12 presents the final conclusions and future work.

2. Related Work

In this section we first review the adaptive resource reservation and slack allocation algorithms in traditional real-time systems, their potential in resolving the resource efficiency, and overrun tolerance issues in MCS. Then we show the usage of slack distribution algorithms in the context of MCS. At last, we review one existing solution towards replacing the greedy algorithms that simultaneously increase the budget for all high-criticality tasks.

In traditional real-time systems, there is a vast literature using the online admission control and adaptive reservation to deal with the uncertainty of execution time and to avoid the disadvantage of offline WCET assessment. They compute and adjust the budget of each task according to its actual demand. Typical implementation employs feedback and prediction mechanisms together with online schedulability testing to measure and predict the resource reservations during runtime [55–58]. This kind of technique has been barely considered in the area of MCS. One possible reason would be that there is always delay between the change of demand and the system's reaction towards the demand. During the delay, overrun events can happen which in turn cause deadline violations. Since critical tasks have high standard for deadline satisfaction, additional endeavor needs to be made for introducing this technique into MCS.

For handling the transient overrun faults as well as increasing resource utilization in reservation based scheduling, slack allocation algorithms have been studied [59–66]. These researches present that, by redistributing the reserved but unused resource, most overrun events within each reservation unit can be well handled without violating the timing constraint. In our approach in particular, we adopt the algorithm presented in [66] which claims to have better performance than IRIS (Idle-time Reclaiming Improved Server) [65], BEBS (Best-Effort Bandwidth Server) [60], GRUB (Greedy Reclamation of Unused Bandwidth) [61], RBED [62], CBS (Constant Bandwidth Server) [59], and CASH (CApacity ScHaring) [66] in bounding deadline misses.

To take advantage of both adaptive reservation schemes and slack allocation algorithms, some studies combine these two techniques together, as is the case in our paper. In [67–70], the authors present work that dynamically adapts the reservation parameters on top of CBS [59]. In [71, 72], Palopoli et al. integrate a feedback control mechanism together with ShRUB (Shared Reclamation of Unused Bandwidth), a variant of GRUB [61]. This kind of approach has certain benefit that it can avoid resource overallocation with a budget adapted to the needs and tolerate the budget overrun until the system reacts to the measurements. Unfortunately, since these mentioned works do not take into account the notion of criticality levels, they are not directly applicable for MCS.

Targeting for MCS, slack distribution algorithms have already been adopted in multiple papers [25, 35, 36, 42, 51]. The basic idea of them is to let the low-criticality tasks execute on the slack of the high-criticality tasks. In [35, 36], De Niz et al. propose zero slack rate-monotonic scheduling and later combine it with additional support for Quality-of-Service (QoS) Resource Allocation Model. In [25], Park and Kim introduce CBEDF (Criticality Based EDF) on top of EDF scheduling. In [42], Lipari and Buttazzo reuse the policy of GRUB [61] in MCS. In [51], Groesbrink et al. propose an algorithm for QoS-aware system where slack is adaptively allocated to tasks that can benefit from additional resource.

In order to replace the greedy algorithms that simultaneously increase the budget for all high-criticality tasks, one approach is proposed by Gu et al. [73]. They partition tasks into separate components. Then for tasks inside each component, they define the fault tolerance limit by a number of violations of WCETs. Only when the WCET violations within a component exceed the component's expected limit are the tasks in other components affected. Our work is different from theirs in that it monitors each task individually and does not rely on the correctness of the offline WCET estimation.

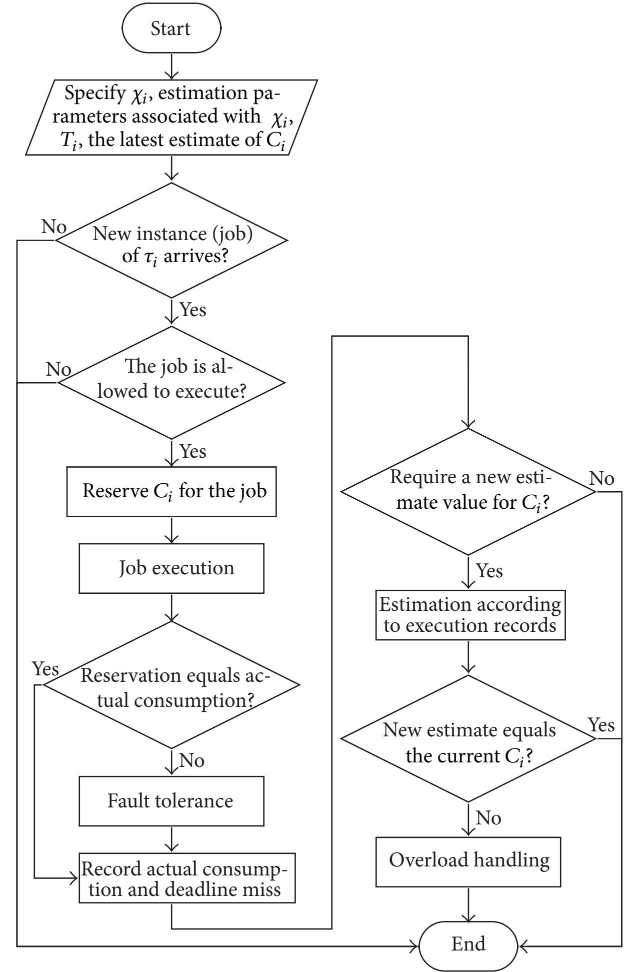


Figure 1: Flow chart for overall adaptive reservation process of τ_i.

3. Model

In a MCS, we define an ordered set of criticality levels: $\chi\{\chi_1, \ldots, \chi_m\}$ ($m \in N^*$, $\chi_i \in N$). A smaller χ_i represents a lower criticality. Only periodic tasks are considered in this paper. Each task τ_i is characterized by the criticality level, a relative deadline D_i, and a period T_i, where $D_i = T_i$.

The system predicts the requirement of a task by a prediction mechanism during runtime and then grants the task a budget C_i for each release period T_i. Therefore, the processor usage for each task is $U_i = C_i/T_i$. The major differences between our model and general MCS models are the following: the reserved resource does not equal the offline WCET estimate and it is modified adaptively and individually for each task during runtime.

A set of tasks is said to be schedulable under EDF scheduling if $CPUUsage = \sum U_i \leq 1$. Otherwise, the tasks will be suspended in a reverse order of criticality levels by admission control until the scheduling is feasible.

The overall adaptive reservation process for a task τ_i is shown in Figure 1. Upon the new job arrival of τ_i, the system first checks if τ_i has been suspended as a consequence of a processor overload condition. If it is not the case, then a budget C_i is reserved for it. During the job execution, if the actual consumption does not equal the reserved budget C_i, the fault tolerance service will be activated to deal with the overrun problem or redistribute the reserved but unused resource. When τ_i finishes the job execution in the current release period, the system determines whether C_i should be replaced by a new estimation to apply from its next release period. The decision is made according to τ_i's execution records, the acceptable deadline miss ratio for a task at criticality χ_i, and the estimation interval defined by the user. This estimation process also takes into account the acceptable deadline miss ratio for a task at criticality χ_i. With the new estimated value of C_i, the system does an overall calculation to predict whether the processor will be overloaded or if enough resource will be released for suspended tasks if any. According to these predictive calculations, the system then suspends or reactivates tasks taking into account their criticality levels. Such operations preferentially grant the required budget to a higher criticality level.

The detailed design and implementation are described in the following sections.

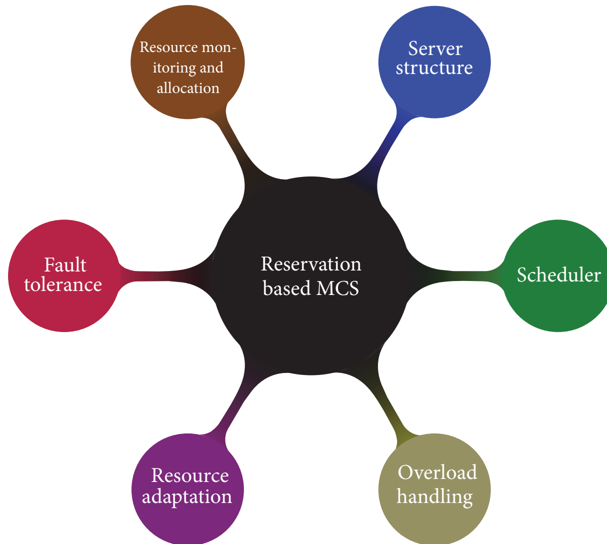


Figure 2: Services.

4. System Architecture

This section briefly describes all the services provided by the described system and their connections with each other.

Required by the adaptive reservation process (Figure 1), the system proposed provides the basic services that support reservation based scheduling and the extended services that are implemented above the basic ones and support adaptive modification to the reserved budget of each task. Figure 2 shows all the services included in the design.

The basic services refer to task isolation, processor allocation and resource monitoring, and general support for a scheduler. The extended services refer to the fault tolerance, resource adaptation, and overload handling.

Task isolation structure is used to ensure that each task only has access to the processor bandwidth assigned to it. This is the central idea of a reservation based system and its purpose is to minimize the interferences among tasks that are sharing the same hardware platform. As a result of task isolation, each task belongs to a reservation unit, which is called server in this design.

The processor allocation and monitoring service is responsible for distributing a specific amount of resources to a certain task (or server) and monitoring the actual consumption of tasks' executions.

The scheduler is what is used in scheduling the servers. Instead of scheduling the tasks directly, a reservation based system has global scheduler that can only see the servers. Besides the algorithm design according to the selected scheduling policy (e.g., EDF, FP), there are some general supports needed for such a scheduler: the definition for server queue data structure, the design of queue management mechanism, the scheduling point selection, and the interaction with the local scheduler of the underlying OS (μC/OS in this case).

The fault tolerance service is designed to tolerate the transient overrun faults before any change to the budgets assignment. It takes into account that servers other than the one being overrun might have spare budget. Or the overloaded server itself will have spare budget in the following execution. This kind of spare resource can be used to resolve the overrun event which avoids a waste of resource on the one hand and prevents performance lost on the other hand. Besides that, this service also serves as the foundation for the resource adaptation service. It loosens the requirement for the budget values reserved for each server, meaning that as long as the difference between the actual execution time and the reserved budget is within a range, it is bearable by the server. As a result, less conservative and more flexible budget estimations are allowed.

The resource adaptation service is used for predicting and adjusting servers' budgets during runtime. The fault tolerance service can handle occasionally overrun events and reduces the wasted resource to a certain extent. This is under the assumption that the reserved budget is not too much lower than what is required in most real executions. If this is not the case, then there would be serious problems. On the one hand, a budget value which is being exceeded frequently will greatly add to the fault handling cost. If these overrun events are happening consecutively, there is even an accumulative difficulty in handling them which will further increase the risk of deadline miss. On the other hand, an overconservative budget value can increase the resource lost, with or without the fault tolerance mechanism. Even worse, overallocation to one or several servers sometimes gives a false impression that the system is overloaded. This is the major drawback when using WCETs as reserved budget values as in traditional MCS implementation. This stated condition motivates us to design this resource adaptation service that dynamically changes the budget value for every server via feedback control and online estimation.

The system overload handling service is designed to be activated when the system cannot provide required budget for all the servers. This service supports the schedulability test and budget reconfiguration with respect to the criticality levels. The objective is to react quickly upon the overload event and redistribute the resource to primarily consider the demands of high-criticality tasks. The schedulability test algorithms associated with this service highly depend on the correction of the budget values provided by the resource adaptation service.

5. Server Structure

Figure 3 shows the basic structure of the proposed system with server support. The servers are scheduled with the global scheduler which refers to "scheduler" in Figure 2. In the following paragraphs, the scheduler for the servers will be denoted as global scheduler and the μC/OS III's scheduler will be denoted as local scheduler.

Two types of servers are implemented in this paper: the periodic server and the idle server. The periodic servers receive processor bandwidth from the resource allocator periodically. The idle server only starts running when no other server requires the processor resource. μC/OS III has an idle task which runs whenever there are no other tasks ready.

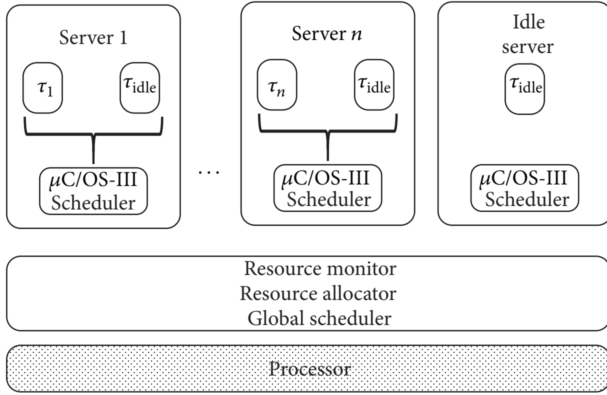


Figure 3: Support for servers.

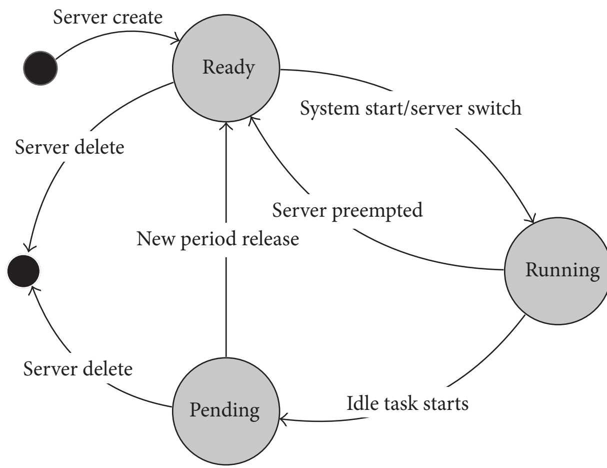


Figure 4: Three basic states of a periodic partition.

In this design, both these two kinds of servers contain idle task, but the idle server contains only the idle task.

Each task τ_i is assigned a periodic server S_i which is identified by a resource interface (T_i, C_i, χ_i). This interface informs the global scheduler that the server houses a task with criticality χ_i and its execution requirement is C_i in every period T_i.

A server has three states as shown in Figure 4. It indicates some basic movements from one state to another. A periodic server is put into the ready state by calling the server-creation function. The global scheduler is responsible for selecting the server to be put into running state. The selected server then will be switched in upon system start or server switch. Any server that finishes the task's execution will be put into pending state. When a server's remaining period reaches zero, it is put back into the ready state. If the global scheduler then detects that the newly released server has become the most important (depends on EDF scheduling policy), it will preempt the currently running server. In the meantime, the currently running server will stay in the ready state and wait until it becomes the most important again. The servers can be deleted by calling the server delete function.

6. Resource Monitoring and Allocator

The resource monitoring of this framework can be divided into three parts. The first is the heart beat resource monitor (HBRM), which executes upon every system clock tick. Its purpose is to keep track of the remaining budget of the active server as well as the release time of each server's new period. When a system clock tick event occurs, it will be triggered and will preempt the currently running task of the active server. A server S_i has a pair of counters (Π_i, Θ_i) indicating the remaining period (relative deadline) and budget. Their initial values equal (T_i, C_i). The time unit for indicating the values is the system clock tick. Every time the HBRM is triggered by a system clock tick event, Π_i is decreased by one. Once Π_i reaches zero, server S_i will be released, while when the HBRM detects that S_i is in the running state, Θ_i will be decreased by one. In the meantime, the queue to which this server belongs will be updated depending on global scheduling policy, fault tolerance policy, and resource adaptation policy. The procedure is shown in Algorithm 1.

Figure 5 shows an example of a timing diagram of a periodic server. The parts with dark background show the work done by HBRM. More detailed description about the changes of Θ_i will be introduced later in the following sections.

The second part is the Server Idle Detection (AID). It is used by us to detect τ_i's completion. From the moment the idle task starts running, the system will know that the current task τ_i has finished its execution. Then τ_i's remaining budget Θ_i (slack), if greater than one, is available for redistribution among other servers. The redistribution policy and its implementation is introduced in Section 8.

The third part is Per-server Budget Usage Recorder (PBUR). In order to get the budget usage in each execution of S_i, there are three possible ways:

(1) Using the remaining budget of S_i. Once the idle task starts running, if the current server is not the idle server, it indicates that τ_i has finished its executing. Then the value of $C_i - \Theta_i$ is the budget actually used by τ_i.

(2) Similar to HBRM. When S_i is the currently running server, $S_i.BudgetTickCounter$ is incremented upon every system clock tick. Once the idle task starts running, $S_i.BudgetTickCounter$ is saved as the budget actually used by τ_i.

(3) Using the hardware timer to mark the time $t_{i,\text{active}}$ that S_i starts running, the time $t^j_{i,\text{preempted}}$ that S_i is preempted for the jth time during the current execution, the time $t^j_{i,\text{resume}}$ that S_i resumes after the jth preemption, and the time $t_{i,\text{complete}}$ that τ_i completes its execution. Assume that in one instance of periodic server S_i it is preempted n times. The actually used budget C'_i equals $(t_{i,\text{complete}} - t_{i,\text{active}}) - \sum_{j=1}^{n}(t^j_{i,\text{preempted}} - t^j_{i,\text{resume}})$.

Our design chooses the third approach. The first has the advantage of limited memory reading times. However, it will be very complex to provide a budget reclamation and

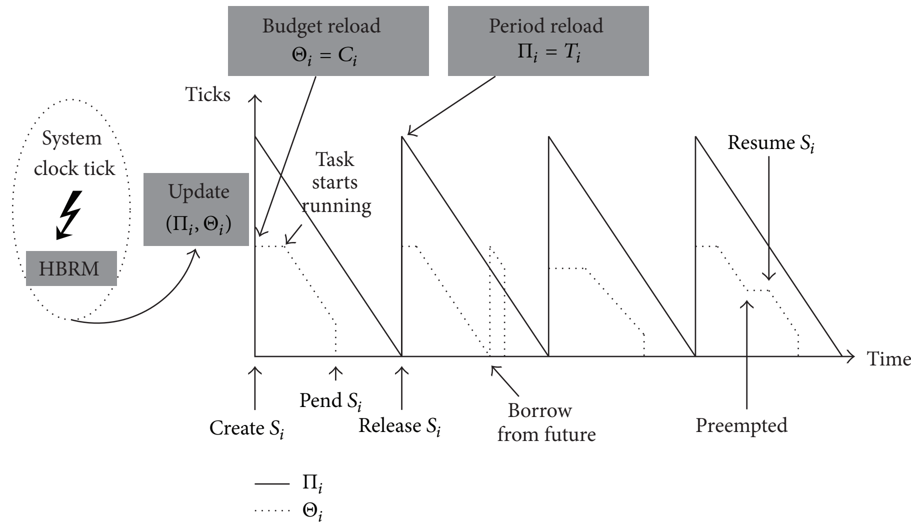


Figure 5: Periodic server with heart beat resource monitor.

```
(1) procedure HBRM
(2)     Update_Remaining_Budget(CurrentRunningServer)
(3)     Update_Remaining_Period(S_1, ..., S_n)
(4)     Update_Queue(ServerPendingQueue)
(5)     Update_Queue(ServerReadyQueue)
(6)     Update_Queue(···)
(7)     ...
(8) end procedure
```

Algorithm 1

borrowing interface for the fault tolerance service, because the remaining budget Θ_i is influenced not only by C_i but also by the reclamation and borrowing slacks. The second is a straightforward approach, but changing the variable's value after every clock tick event causes more overhead than the other two approaches. In the implementation of the third solution, a separate hardware timer is used other than the one for the system clock tick. The resolution (42 ns) is set to be higher than the system clock tick. Thus, it prevents a loss of accuracy during statistical calculation.

7. Scheduler Support

There are four scheduling points for the global scheduler: when a server is created; when a server in the ready state is deleted; after HBRM's execution; and when the idle task starts running and the current server is not the idle server. The pseudocode in Algorithm 2 shows the overall scheduling process. It includes two steps: selection of the next running server and selection of the next running task. The function *Global_Schedule*() selects the running server S_i. The detailed procedure is shown in Algorithm 3. *Local_Schedule*() represents the original scheduler of μC/OS III. It is called every time after *Global_Schedule*() and decides which one among τ_i and τ_{idle} should run.

Algorithm 3 shows the global scheduler's behaviour in EDF scheduling (with no support for fault tolerance service). It always selects the head item in the *ServerReadyQueue* to be the next running server. If the head item in the *ServerReadyQueue* exists and is not the currently running server, a server switch will be implemented. The local scheduler then switches to the corresponding server's *TaskReadyQueue*. *TaskReadyQueue* as defined in μC/OS III consists of a bitmap containing the priority levels that are ready and a table containing pointers to all the tasks ready.

Doubly linked queues (*ServerReadyQueue*, *ServerPendingQueue*, etc.) are employed for global scheduling. Each queue is a data structure of *SERVER_LIST*. It consists of one pointer to the first server in the queue. Correspondingly, a server's basic data structure has three fields as shown in the list below.

```
Server_TCB {
    *PrevPtr;
    *NextPtr;
    *ServerListPtr;
    ...
};
```

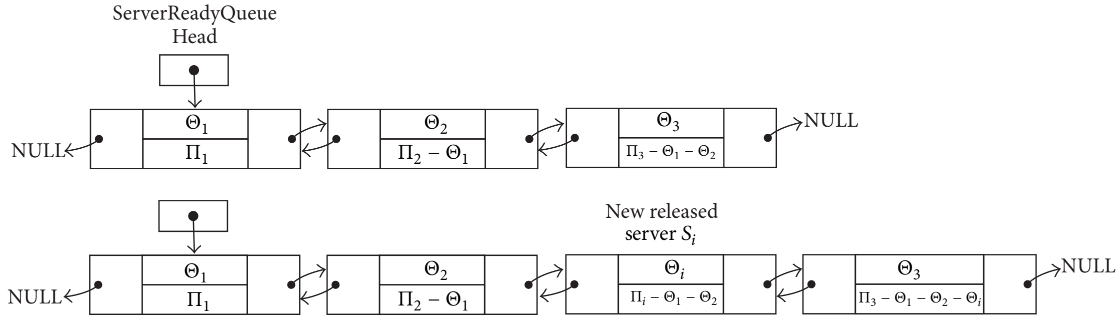


Figure 6: Ready queue insertion example ($\Pi_1 \leq \Pi_2 < \Pi_i \leq \Pi_3$).

```
(1)  procedure SCHEDULE
(2)     Global_Schedule()
(3)     Local_Schedule()
(4)  end procedure
```

Algorithm 2

.ServerListPtr is a pointer to a data entry of the queue to which the server belongs. *.NextPtr* and *.PrevPtr* are used for doubly linked queues. The last server in a queue *.NextPtr* points to *NULL*.

Figure 6 presents an example of the insertion of a newly released server S_i into *ServerReadyQueue*, where EDF is the global scheduling policy. A pair of integers, Π_1 and Θ_1, need to be decremented upon system clock tick interrupt.

8. Fault Tolerance

The fault tolerance service relies on the resource monitor HBRM and AID. The purpose is to handle the situation where the task's actual execution time exceeds the assigned budget of the server to which it belongs. The implementation of this service employs the BACKSLASH presented in [66] whose main principles are summarised as follows:

(1) Allocate slack as early as possible.

(2) Allocate slack to the one with the earliest original deadline.

(3) Allow borrowing against a server's own future resource reservations to complete the current execution. The server's priority after borrowing is changed to the one from which the resources are borrowed.

(4) Retroactively allocate slack to servers that have borrowed from their current budget during their last release period.

To implement the principles, other than *ServerPendingQueue* and *ServerReadyQueue*, two more queues are used: the server borrowing queue in ready state (*ServerReadyBorrowQueue*) and the server borrowing queue in pending state (*ServerPendingBorrowQueue*).

When the HBRM detects that the budget of the running server S_i has been exhausted ($\Theta_i = 0$) while τ_i's execution remains unfinished (as the second execution period of S_i indicated in Figure 5), S_i will be put into the *ServerReadyBorrowQueue*. Meanwhile, S_i borrows from its own future budget with an extended deadline ($\Theta_i = C_i$, $\Pi_i = \Pi_i + T_i$). Now, server S_i simultaneously exists in two queues *ServerReadyQueue* and *ServerReadyBorrowQueue*. *ServerReadyBorrowQueue* is sorted with the servers' real deadlines and *ServerReadyQueue* is sorted with extended deadlines. A different pair of pointers, *.BrwNextPtr* and *.BrwPrevPtr*, are used in *ServerReadyBorrowQueue* as shown in the server's data structure below.

```
Server_TCB {
    ...
    *BrwPrevPtr;
    *BrwNextPtr;
    ...
};
```

When τ_i eventually finishes one execution, S_i is put into the *ServerPendingBorrowQueue* in order of real deadlines. The queue arrangements are done after the AID by the idle task and a schedule point is followed. In a different scenario that S_i is detected idle without borrowing, its remaining budget Θ_i will be distributed to other servers as slack. The slack receiver is selected from *ServerPendingBorrowQueue*, *ServerReadyBorrowQueue*, and *ServerReadyQueue*. The overall process is expressed by Algorithm 4.

The slack donation is processed by Algorithm 5. The head server S_r in *ServerPendingBorrowQueue* is the first one that has its remaining budget increased. Depending on the value of Θ_i, S_r's remaining budget can be increased by up to the processor capacity that it consumed beyond C_r during its last run. At the same time, Θ_i is decreased by the same amount. Then, if Θ_i is still greater than zero, the next available server in *ServerPendingBorrowQueue* will increase its remaining budget. Until *ServerPendingBorrowQueue* is empty, the slack goes to the server in the ready queue that has the shortest original deadline. After the slack donation process, the head server in the *ServerReadyQueue* starts running.

Compared with the example of the plain EDF described in Section 7, the ready queue management method is modified

```
(1)  Function GLOBAL_SCHEDULE()
(2)    If CurrentRunningServer = ServerReadyQueue.HeadServer then
(3)      Return
(4)    else if ServerReadyQueue.HeadServer =
     NULL&&CurrentRunningServer = IdleServer then
(5)      Return
(6)    else if ServerReadyQueue.HeadServer =
     NULL&&CurrentRunningServer ≠ IdleServer then
(7)      CurrentRunningServer = IdleServer
(8)      TaskReadyQueue = IdleServer.TaskReadyQueue
(9)    else
(10)     CurrentRunningServer = ServerReadyQueue.HeadServer
(11)     TaskReadyQueue = CurrentRunningServer.TaskReadyQueue
(12)   end of
(13) end function
```

Algorithm 3

```
(1)  if CurrentRunningServer = IdleServer then
(2)    Return
(3)  else if CurrentRunningServer = ServerReadyBorrowQueue.HeadServer
     then
(4)    ServerPendingBorrowQueueInsert(CurrentRunningServer)
(5)    ServerReadyBorrowQueueRemove(CurrentRunningServer)
(6)    ServerReadyQueueRemove(CurrentRunningServer)
(7)  else if CurrentRunningServer.Θ_i > 0 then
(8)    ServerReadyQueueRemove(CurrentRunningServer)
(9)    Slack_Donation(Θ_i)
(10) end if
```

Algorithm 4

to be used together with this fault tolerance service, because the exact time at which a server in the *ServerReadyQueue* starts to run can no longer be predicted. After the modification, all the servers' remaining periods in the ready queue have to be decreased upon system clock tick. Figure 7 presents an insertion of a newly released server S_i into the *ServerReadyQueue* with the ready queue management method which adapts to this fault tolerance policy.

9. Resource Adaptation

The resource adaptation service is designed to adaptively change the reservation for each server during runtime. It relies on the whole of the support from the resource monitor HBRM, AID, and PBUR. The statistical tool adopted here is Chebyshev's inequality which is providing a simple implementation possibility needed in our limited embedded environment.

Based on Chebyshev's inequality, for a positive number k, a random variable X with the standard deviation σ, the bound follows formula (1). $P(|X - \overline{X}| \geq k\sigma)$ represents the possibility that X's value is not inside the bound. Since the distribution of X has equal lower and upper tails (Figure 8), k can be estimated by (2). The budget's upper bound here is then estimated by (3):

$$P\left(\left|X - \overline{X}\right| \geq k\sigma\right) \leq \frac{1}{k^2}, \quad k \geq 1, \tag{1}$$

$$k = \sqrt{\frac{1}{2 * P\left(X - \overline{X} \geq k\sigma\right)}}, \tag{2}$$

$$X = \overline{X} + \sqrt{\frac{1}{2 * P\left(X - \overline{X} \geq k\sigma\right)}} * \sigma. \tag{3}$$

The statistical calculation resource is the budget actually used by N successive past executions: $\{C'_{ij} \mid \forall j \in m \leq j \leq m + N - 1\}$, where j donates the jth execution. We use (3) to estimate the future budget (i.e., $C_i = X$) and reassign the result to each server as the new reservation budget. The mean and standard deviation are calculated by

$$\overline{X} = \frac{1}{N} \sum_{j=m}^{m+N-1} C'_{ij},$$
$$\sigma = \sqrt{\frac{1}{N-1} * \sum_{j=m}^{m+N-1} \left(C'_{ij} - \overline{X}\right)^2}. \tag{4}$$

(1) **function** SLACK_DONATION(Θ_i)
(2) **while** $\Theta_i > 0$ **do**
(3) $S_r = ServerPendingBorrowQueue.HeadServer$
(4) $S_s = ServerReadyBorrowQueue.HeadServer$
(5) $S_t = ServerReadyQueue.HeadServer$
(6) **if** $S_r \neq NULL$ **then**
(7) **if** $\Theta_i \geq C_r - \Theta_r$ **then**
(8) *Donate Slack* $C_r - \Theta_r$ *To* $S_r(\Theta_r, \Pi_r)$
(9) $\Theta_i = \Theta_i - (C_r - \Theta_r)$
(10) $ServerPendingBorrowQueueRemove(S_r)$
(11) $ServerPendingQueueInsert(S_r)$
(12) **else**
(13) *Donate Slack* Θ_i *To* $S_r(\Theta_r, \Pi_r)$
(14) $\Theta_i = 0$
(15) **end if**
(16) **else if** $S_s \neq NULL$ **then**
(17) **if** $S_s = S_t$ **then**
(18) *Donate Slack* Θ_i *To* $S_s(\Theta_s, \Pi_s)$
(19) $\Theta_i = 0$
(20) **else if** $\Pi_s < P_s$ **then**
(21) *Donate Slack* Θ_i *To* $S_s(\Theta_s, \Pi_s)$
(22) $\Theta_i = 0$
(23) **else if** $\Pi_s - P_s < \Pi_t$ **then**
(24) *Donate Slack* Θ_i *To* $S_s(\Theta_s, \Pi_s)$
(25) $\Theta_i = 0$
(26) **else**
(27) *Donate Slack* Θ_i *To* $S_t(\Theta_t, \Pi_t)$
(28) $\Theta_i = 0$
(29) **end if**
(30) **else if** $S_t \neq NULL$ **then**
(31) *Donate Slack* Θ_i *To* $S_t(\Theta_t, \Pi_t)$
(32) $\Theta_i = 0$
(33) **else**
(34) $\Theta_i = 0$
(35) **end if**
(36) **end while**
(37) $ServerPendingQueueInsert(CurrentRunningServer)$
(38) **end function**

Algorithm 5

The configurable variables from the user side are $P(X - \overline{X} \geq k\sigma)$ and N. A smaller $P(X - \overline{X} \geq k\sigma)$ can guarantee a lower overrun rate as well as a higher preassigned budget. For different χ_i, there is a corresponding P_i. Normally they should meet the condition that when $\chi_i > \chi_j$, then $P_i > P_j$.

With regard to the choice of the value of N, on the one hand, it should be a relatively big number (≥30) to have a better statistical precision. The execution frequency of the statistical recalculation for S_i is $1/(N * T_i)$, which can also be lowered by increasing N. On the other hand, since the point at which τ_i's computation requirement dramatically changes is not predictable, the longer the budget reestimation calculation interval, the higher the chance that a high overrun rate will occur between two reestimation calculations. Examples are shown in Figure 9(a). With low frequency, when a server S_i's budget requirement increases to a higher value only a few release periods after τ_i's $2N$th execution, the earliest time for it to receive a higher budget as a reaction to the increase of actual consumption is almost N release periods after. Therefore, from τ_i's $(2N + 1)$th to $3N$th execution, S_i might suffer from a high overrun rate. With high frequency, the duration of the overrun situation will obviously be shortened.

In order to achieve better statistical precision and react quickly upon changes, a solution with dynamic interval between two budget estimation calculations is proposed. Each time τ_i overruns its assigned budget ($C'_{ij} > C_i$), a counter *OverrunCtr* will increase by one. From the moment the overrun rate is higher than the expected percentage ($P(X - \overline{X} \geq k\sigma)$), it triggers an immediate recalculation for budget prediction. In Figure 9(b), for instance, after the mth execution, the system arranges a recalculation after detecting that $OverrunCtr/(m - 2N) > (1 - P(X - \overline{X} \geq k\sigma))$. If the overrun rate stays below $P(X - \overline{X} \geq k\sigma)$, the budget estimation calculation will happen every N executions, where N is configured by the user.

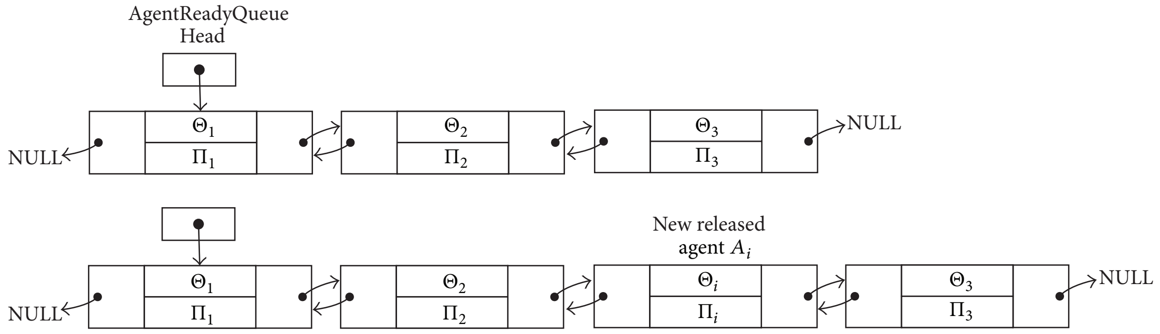


FIGURE 7: Ready queue insertion example ($\Pi_1 \leq \Pi_2 < \Pi_i \leq \Pi_3$).

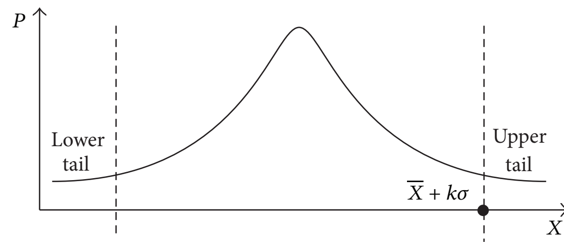
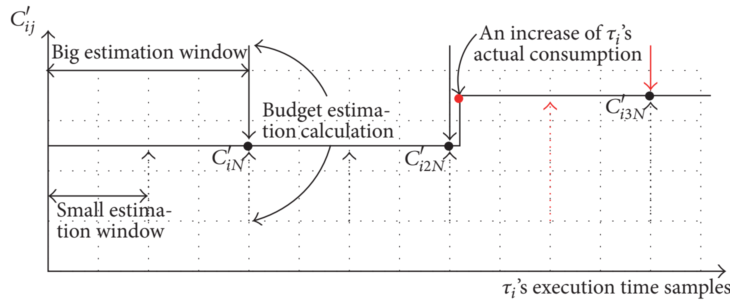
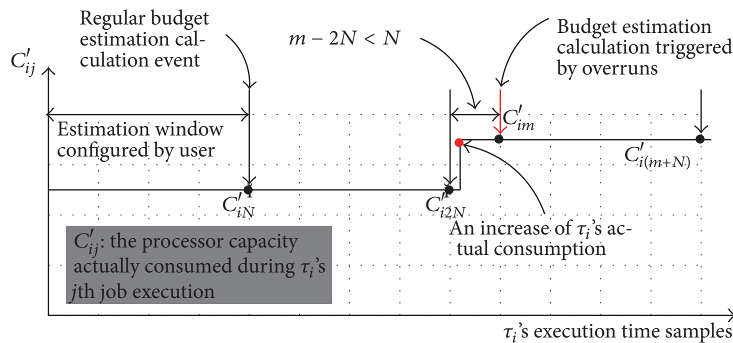


FIGURE 8: Probability distribution, upper bound $\overline{X} + k\sigma$.



(a) Budget reestimation calculation on fixed intervals: The red arrows indicate the earliest time for the system to react to the increase of τ_i's consumption when configuring the system with small (below) and big estimation intervals (above)



(b) Dynamically triggered budget reestimation calculation

FIGURE 9: Budget reestimation calculations, x-axis denotes the number of execution time samples since the first job released by τ_i; y-axis denotes the processor capacity actually consumed during each job execution.

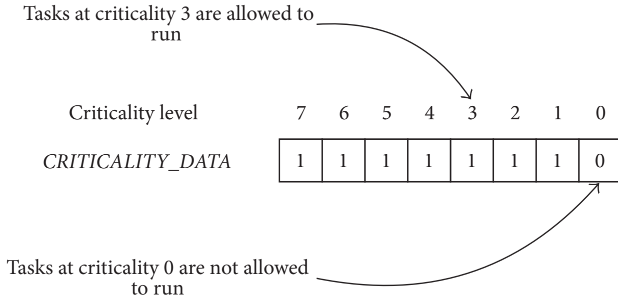


Figure 10: An 8-bit *CRITICALITY_DATA*.

The proposed dynamic interval policy guarantees for each task with criticality level χ_i the rate of overruns is bounded by P_i. Therefore the rate of its deadline misses is also bounded by P_i in a nonoverload system.

The value of *CPUUsage* is modified upon each server budget reassignment event with (5), where ΔC_i is the difference from the previously assigned budget

$$CPUUsage = CPUUsage + \frac{\Delta C_i}{T_i}. \quad (5)$$

10. System Overload Handling

The overload handling service serves the situation where the resource adaptation service cannot assign the servers' required budget due to system overload. It is designed to respect the tasks' criticality levels in a MCS. The idea is to suspend low-criticality level tasks for some time in order to make resources available for a high-criticality level task.

A *CRITICALITY_DATA* entry is used to identify whether servers at each criticality level are allowed (1) or not (0) to run the housed tasks. The number of criticality levels supported depends on the bit-size of *CRITICALITY_DATA* which is limited by the processor data width. For example, on a 32-bit processor, it can support up to 32 criticality levels. If needed, this limitation can be easily overcome by using several such data entries. With an 8-bit data entry the criticalities are numbered from right to left as demonstrated in Figure 10. Since for the time being the existing standards consider no more than five criticality levels, one data entry is enough. When the number of criticality levels is less than the number of bits in *CRITICALITY_DATA*, several upper bits are masked to 1 according to the difference of these two numbers. Besides that, a vector *SERVER_CPUUsage*[] is used to store the amount of processor resources occupied by each criticality level when they are allowed to run.

OverloadHandle() is called after dynamic budget prediction. Algorithm 6 presents the process, where $C_{i(n+1)}$ is the newly calculated budget value, C_{in} is the current budget value for S_i, and $U_{i(n+1)} = (C_{i(n+1)}/T_i)$ $U_{in} = C_{in}/T_i$. Figure 11 is the flow chart for the function *OverloadHandle*().

We use special instructions: Count Leading Zeros (CLZ) and Counting Tailing Zeros (CTZ), which can be found in many processors, to speed up the overload handling process (*OverloadHandle*()).

When a system overload is detected, the first step is using the CTZ instruction to count how many zeros are there in a *CRITICALITY_DATA* entry starting from the right. For example, if *CRITICALITY_DATA* is 8 bit, 0xFE will result in 1 which identifies that the servers at criticality zero already stopped running their tasks. Assume there are l tailing zeros (Figure 12); then the system adds the values from *SERVER_CPUUsage*[l] to a budget sum. If the sum is smaller than the overloaded processor resource, set the corresponding bit to zero in *CRITICALITY_DATA*. Then repeat the steps until the sum value equals or is greater than the overloaded processor usage. The process is shown in Algorithm 7. The corresponding part in Figure 10 is identified as 1.

When a server has a recalculated budget lower than the current one, then this overload handling service first uses CLZ instruction to count how many zeros are there in ~ *CRITICALITY_DATA* from left. Assume there are m leading zeros and *CRITICALITY_DATA* is n bit (Figure 13), then criticality level $n - 1 - m$ is the highest criticality which is being stopped from running its tasks. Then, *SERVER_CPUUsage*[$n-1-m$] will be added to a sum. If the sum is smaller than the spare processor resource, the $n-1-m$ bit is set to be 1. Then repeat the steps until the sum is equal to or greater than the spare processor resource. In another condition that the one requiring resource cannot reclaim enough processor bandwidth, it will suspend itself to avoid affecting servers with higher criticalities. The process is shown in Algorithm 8. The corresponding part is identified as 2 in Figure 10.

This service adds an additional check (*ServerModeCheck*()) upon each server's execution; as soon as it detects the corresponding bit in *CRITICALITY_DATA* is modified, the servers will immediately stop or restart the housed task and change the required budget. Being suspended means the server has zero budget, but the system still switches it from pending state to ready state at the beginning of its periods. When the system tries to start the tasks' execution, *ServerModeCheck*() will detect that the server is in the suspended mode and therefore put it back to the pending state.

11. Performance Evaluation

In this section, the system's performance in supporting MCS is evaluated. The experiments are performed on a single core ARM-based platform, Beaglebone Black. The processor runs at a frequency of 1 GHz and the system's clock tick frequency is set to be 1 kHz (default value of μC/OS III).

Evaluation 1 measures the added overhead of server switch, *Global_Schedule*(), server budget and period update, server queue arrangement, slack detection and donation, and reserved budget calculation and prediction.

The overhead tests are based on the technique described in [43]. A 32-bit DMtimer, which is a timing register of AM3358, is used for measuring the time intervals. The timer is configured to run at 24 MHz; the resolution of the evaluation test results is 0.04 μs. The results shown in this chapter are rounded to the nearest 0.1 μs ($\pm$0.1 μs). This accuracy is about three times faster than the tracing system overhead (see

```
(1) SERVER_CPUUsage[χ_i] ← SERVER_CPUUsage[χ_i] + (U_i(n+1) − U_in)
(2) if C_i(n+1) > C_in ∧ (U_i(n+1) − U_in) ≤ 1 − CPUUsage then
(3)     C_i ← C_i(n+1)
(4)     CPUUsage ← CPUUsage + (U_i(n+1) − U_in)
(5)     Return
(6) else
(7)     OverloadHandle()
(8) end if
```

Algorithm 6

```
(1)  sum ← 0, temp_data ← CRITICALITY_DATA
(2)  while sum < (U_i(n+1) − U_in) − (1 − CPUUsage) do
(3)      l ← CTZ(temp_data)
(4)      if l = χ_i then
(5)          break the loop
(6)      end if
(7)      sum ← sum + SERVER_CPUUsage[l]
(8)      bit at position l in temp_data ← 0
(9)  end while
(10) if l = χ_i then
(11)     C_i ← 0
(12)     bit at position χ_i in CRITICALITY_DATA ← 0
(13)     CPUUsage ← CPUUsage − U_in
(14) else
(15)     C_i ← C_i(n+1)
(16)     CRITICALITY_DATA ← temp_data
(17)     CPUUsage ← CPUUsage − (sum − (U_i(n+1) − U_in))
(18) end if
```

Algorithm 7

Table 1: Tracing overhead test results.

Tracing overhead	Result
Maximum	0.3 μs
Minimum	0.2 μs
Average	0.2 μs

Table 1). The timer available is not synchronized with what we want to measure.

Figure 14 shows the test methodology. For the operation that needs to be tested, upon the start and the end of it, two time stamps are obtained and put to local memory. All the tested operations are related to scheduling and masked from interrupts. So this test will not be delayed by interrupt handlers. There is no task configured with higher priority than the one being tested. So the difference of t_1 and t_2 is the overhead caused solely by the tested operation.

Every test obtains more than 10000 samples making the results sufficient for statistical analysing.

Figure 15 shows the test procedure for tracing overhead. There is no operation between the two time stamp writing actions, so the difference between t_2 and t_1 is the tracing overhead. The aim of this test is to measure precisely the tracing system overhead to enhance the overall accuracy. Tracing system overhead is more hardware-related than software-related. The sources of tracing system overhead are timer register reading and memory writing

$$t_2 - t_1 = OH. \quad (6)$$

The test result (Table 1) shows that there is only very limited jitter in the overhead of the evaluation trace.

The overhead of an operation is obtained by subtracting the minimum tracing overhead (Table 1) from the test result.

In Figure 16, we compare the overheads of server switch when there are different numbers of servers. The results indicate that the value is not related to the number of servers. There is not much difference between the maximum and the average overhead due to the fact that the test set is small enough to fit into the cache. How the cache will affect the performance is beyond the scope of this paper and will not be discussed here.

In Figure 17, *Global_Schedule*()'s overhead in two conditions is shown. The values on the left hand side contain no server switch time. These values are important due to the fact that *Global_Schedule*() is called upon every system clock tick and most of the time it returns with no need for switching between two servers. Here it has a value with complexity $O(1)$. The values on the right hand side show what happens when the server switch time is not subtracted

```
(1)  if ~ CRITICALITY_DATA = 0 then
(2)      return
(3)  end if
(4)  sum ← 0, temp_data ←~ CRITICALITY_DATA
(5)  while temp_data ≠ 0 do
(6)      m ← CLZ(temp_data)
(7)      sum ← sum + SERVER_CPUUsage[n − m − 1]
(8)      if sum > (1 − CPUUsage) + (U_in − U_i(n+1)) then
(9)          break the loop
(10)     end if
(11)     bit at position n − m − 1 in temp_data ← 0
(12) end while
(13) CRITICALITY_DATA ←~ temp_data
```

Algorithm 8

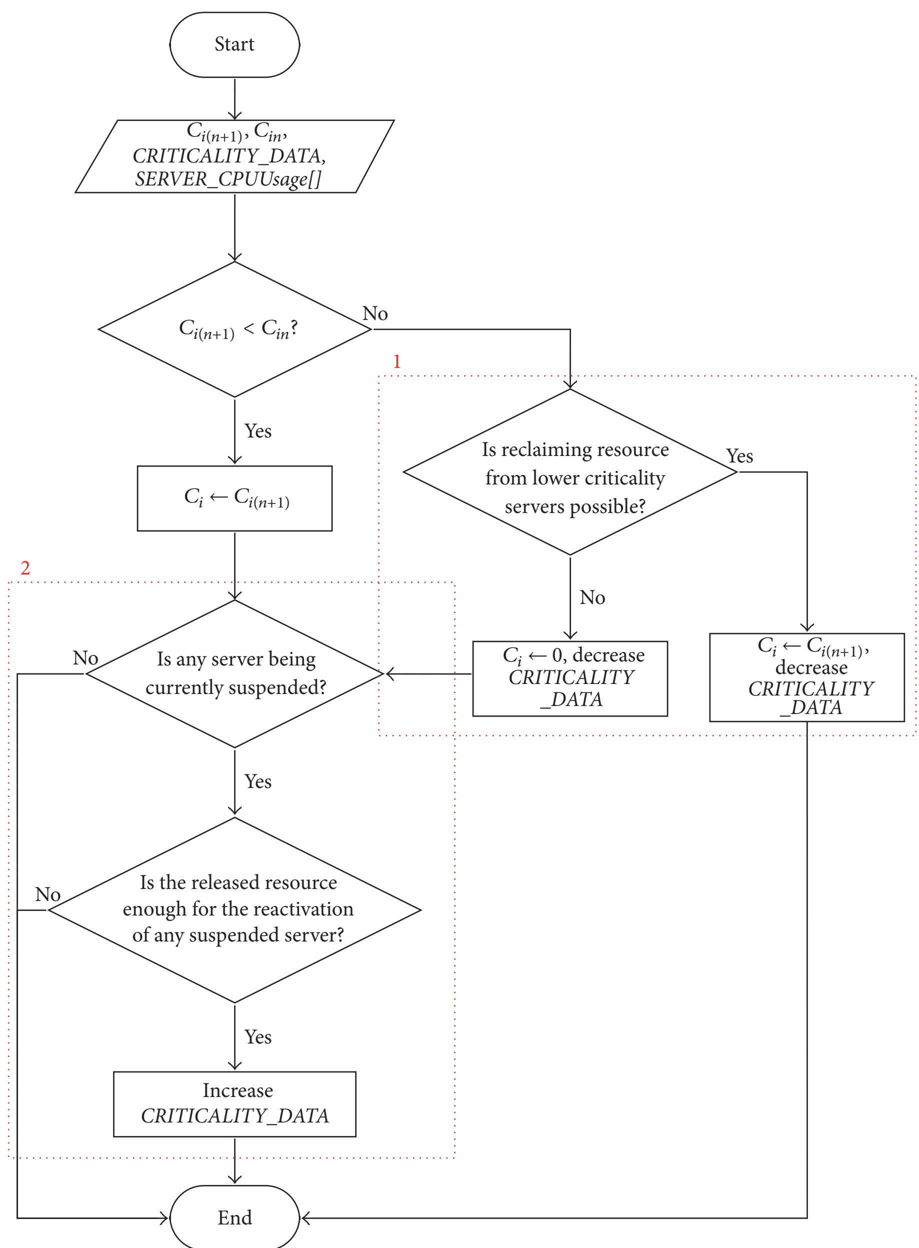


Figure 11: Flow chart for *OverloadHandle*().

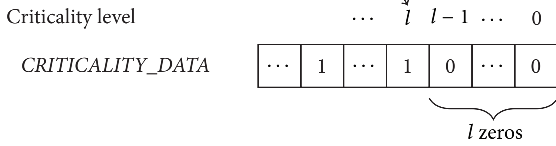


Figure 12: CTZ in use of finding the lowest criticality level of the tasks that are currently allowed to run in the system.

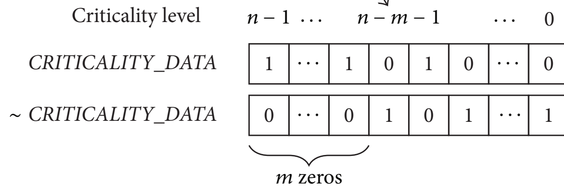


Figure 13: CLZ in use of finding the highest criticality level of the tasks that are currently prevented from execution.

from *Global_Schedule*(). The maximum value occurs when a currently running server is preempted by another newly released server with a shorter deadline. Then the system has to save the preempted server's status before switching between their task-ready lists.

Figure 18 shows the overhead caused by the servers' remaining budgets and periods updating. It is not a constant value due to two reasons: first, the *ServerPendingBorrowQueue* and *ServerReadyBorrowQueue* added by the fault tolerance service introduce uncertainty; second, every member's remaining period in the *ServerReadyQueue* needs to be decreased instead of only the first member's as in the simple EDF example (Figure 6). The complexity of its maximum overhead is then $O(M)$ $(M \leq N)$, where M is the largest possible number of servers that can be in the *ServerReadyQueue* simultaneously and N equals the number of active servers in the system.

Figure 19 denotes the accumulated overhead introduced by *Update_Queue*(*ServerPendingQueue*), *Update_Queue*(*ServerReadyQueue*), *Update_Queue* (*ServerPendingBorrowQueue*), and *Update_Queue* (*ServerReadyBorrowQueue*). Its maximum value is strongly related to how many servers are created in the system. This is as expected. The reason is that the *Update_Queue*() function needs to go through the queues to find the right place to insert the newly released servers or those whose budgets are exhausted. The worst case happens when every server in the queues is checked. Thus the complexity is $O(N)$, where N equals the sum of the number of servers in ready state and in pending state.

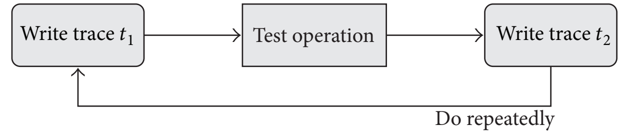


Figure 14: Performance measurement methodology [43].

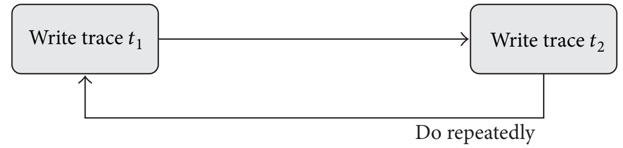


Figure 15: Tracing overhead test procedure [43].

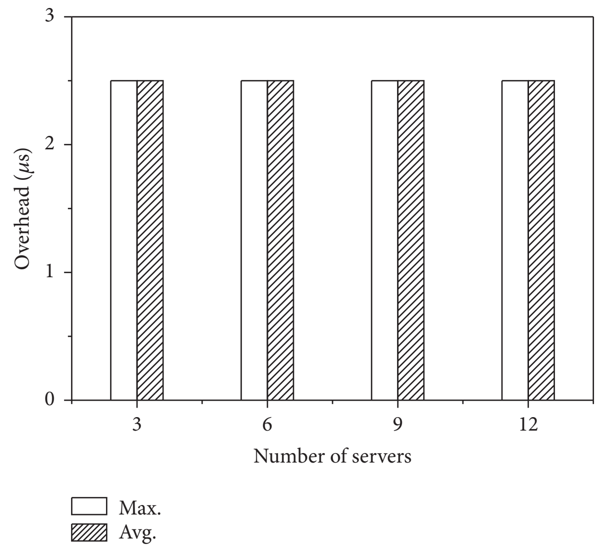


Figure 16: Server switch overhead.

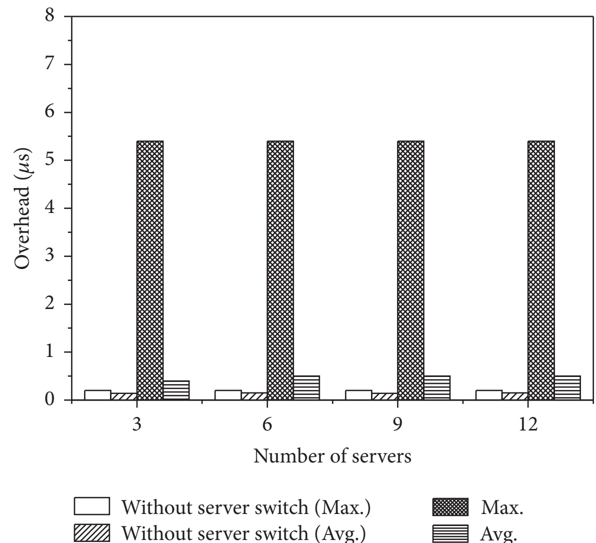


Figure 17: Overhead of *Global_Schedule*().

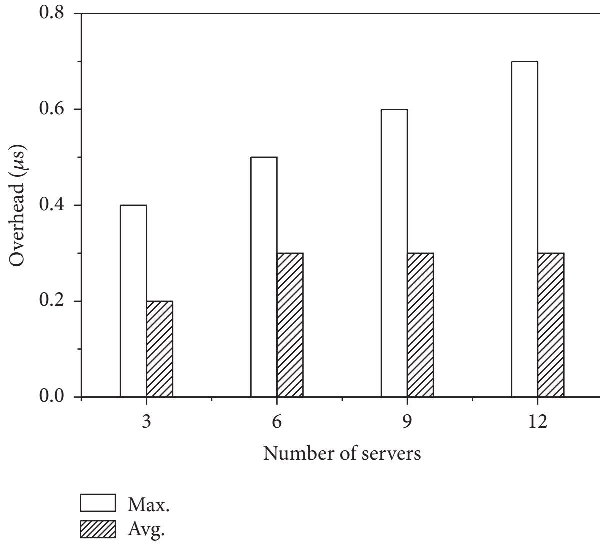


Figure 18: Overhead caused by servers' budgets and periods updating.

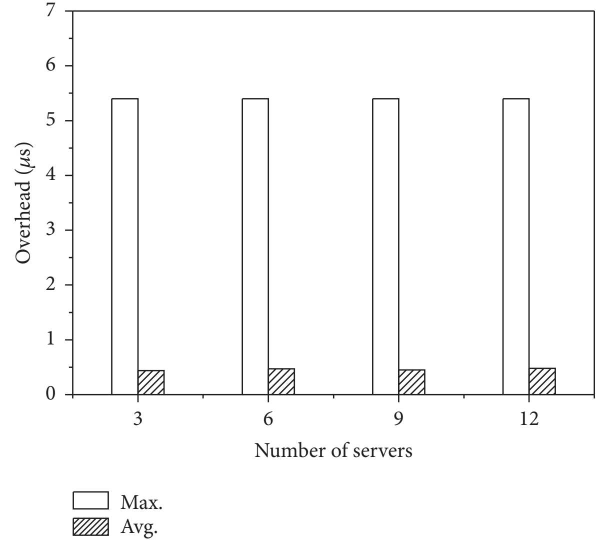


Figure 20: AID + slack donation.

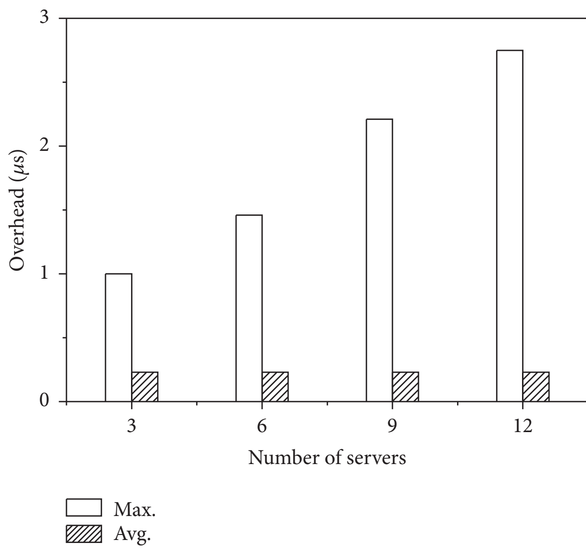


Figure 19: Accumulated overhead caused by *ServerQueueUpdate*().

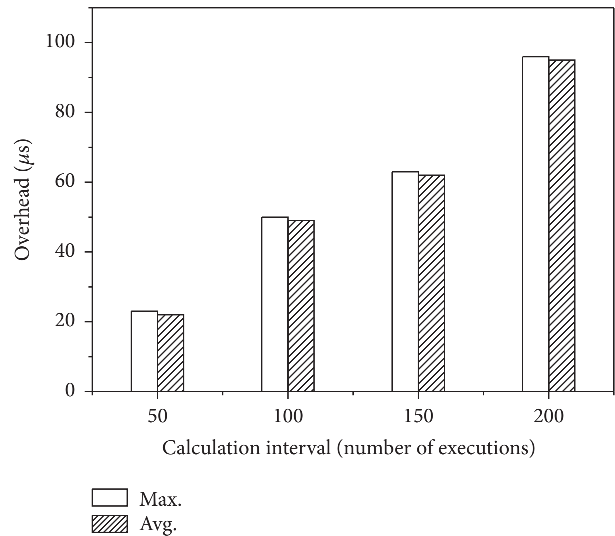


Figure 21: Overhead for reserved budget value estimation.

Figure 20 indicates the overhead introduced by AID and slack donation.

Figure 21 shows the overhead in deciding a server's budget value in relation to the calculation interval (the amount of execution records used in the prediction). The value is highly dependent on the calculation interval. However, when considering processor time (%), which equals (*Absolute Calculation Overhead*/(*ServerPeriod* $*$ *Number of Executions Between each Calculation*)) $*$ 100%, the values for different calculation intervals are almost the same. For instance, for a server that has a period equals 100 ms, the adaptation calculation will cost about 0.005% of the processor time, regardless of how many execution records are taken into account by the calculation. Therefore, if permitted by the system's memory space, adopting a longer regular calculation interval is a good choice in consideration of a better statistical accuracy, and it will not increase the relative overhead.

Evaluation 2 detects, with our platform, whether the fluctuation in a task's computation requirement can be effectively handled and whether the overrun faults can be tolerated in a nonoverload system. For this evaluation, a task is designed with variable consumption in each release period. Its processor bandwidth consumption is generated following a normal distribution with a 10% standard deviation from the mean. By design, the value of the mean changes between the 3000th and 3600th period, the 6000th and 7400th period, and the 12000th and 14400th period. For our system, the calculation parameter $P((X - \overline{X}) \geq k\sigma)$ for this task is set to be 10%. The regular calculation interval in the resource

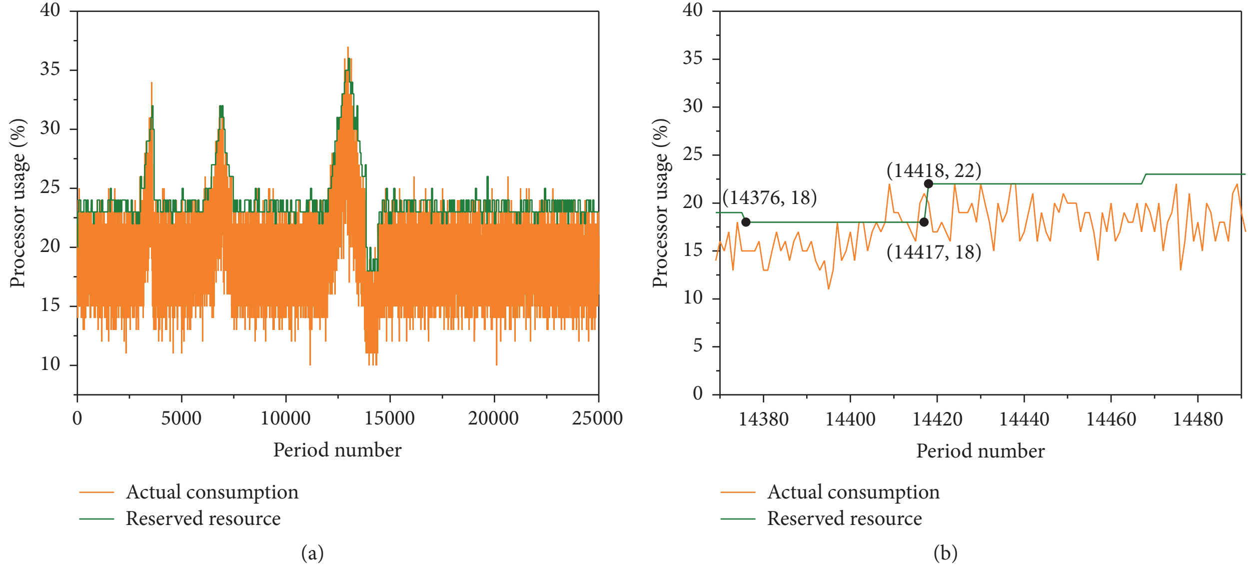


Figure 22: (a) Overall resource adaptation performance. (b) Partial enlarged view of (a): after the execution of the 14417th release period, the resource monitor detects that $OverrunCtr/(m - 2N + 1) = 5/(14417 - 14376 + 1) = 11.9\% > 10\%$ and thus a new calculation event is triggered immediately based on the dynamic calculation interval policy.

Table 2: Overruns and deadline misses.

Period number (∗1000th)	0–3	3–3.6	3.6–6	6–7.4	7.4–12	12–14.4	14.4–25	Total
OR (%)	0.57	2.2	0.50	0.71	0.39	0.88	0.56	0.60
DMR (%)	0	0	0	0	0	0	0	0

adaptation service is set as 50. The system is not overloaded during the whole process.

The overrun ratios (ORs) and deadline missing ratios (DMRs) in Evaluation 2 are shown in Table 2. The results show that when the fluctuation appears (from the 3000th to the 3600th release period, from the 6000th to the 7400th release period, and from the 12000th to the 14400th release period), the OR is higher. However, the overall OR is less than 0.6%. The highest OR is detected in the period from the 3000th to the 3600th execution and is still less than 3%. No DMR event is detected. The bound, which is 10% for both OR and DMR as defined, holds in the tested case.

Figure 22(a) presents the task's actual consumption and the assigned budget in 1–25000th release period. It shows that if a big rise in the consumption lasts for a certain number of periods, a change will be triggered to the assigned budget value. The budget prediction follows the actual change correctly and rapidly. The partial enlarged view in Figure 22(b) shows that when the OR gets too high, a reprediction event will start immediately due to the dynamic calculation interval policy.

Evaluation 3 detects the following: under overload condition, whether the system can guarantee the resource for high-criticality tasks; when there is enough resource, whether the low-criticality tasks can be restarted. The overheads caused by *OverloadHandle*() and *ServerModeCheck*() are also measured. The regular calculation interval in the resource adaptation service is set as 100. There are four periodic tasks running

Table 3: Overview of the tasks being tested in Evaluation 3.

Task	Period (ms)	Criticality	$P((X - \overline{X}) \geq k\sigma)$
τ_1	160	0	10%
τ_2	100	2	2%
τ_3	200	1	6%
τ_4	1000	0	10%

Table 4: ORs DMRs and SRs in Evaluation 3.

Task	τ_1	τ_2	τ_3	τ_4
OR (%)	0.16	0.15	0.68	0.49
DMR (%)	0	0	0	0
SR (%)	63	0	36	63

in this case study. Their criticality levels and corresponding $P((X-\overline{X}) \geq k\sigma)$ are listed in Table 3. τ_2 is predefined to have a highly fluctuated resource consumption which causes system overload at certain time point. The budgets for the suspended tasks are set to be 2 clock ticks instead of zero for a better observation and *SERVER_CPUUsage*[] values are adjusted correspondingly.

In Evaluation 3, the tasks' ORs, DMRs, and the ratios that they are suspended (SRs) due to system overload are shown in Table 4. During the time that the tasks are not suspended, no deadline misses are detected for all the tasks.

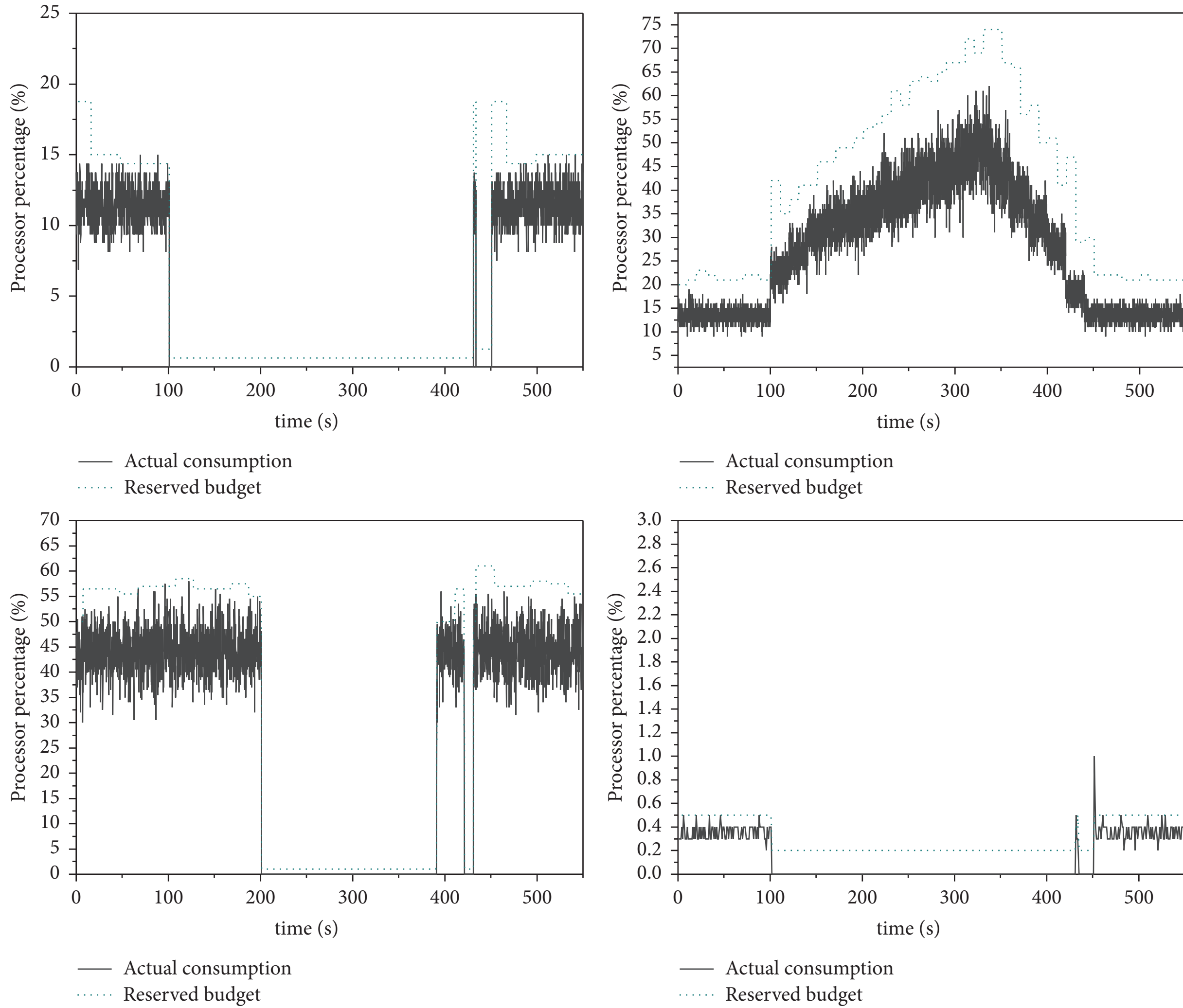


Figure 23: The actual consumption of τ_i and the reserved resource for each task. From top left to bottom right: τ_1, τ_2, τ_3, and τ_4.

Figure 23 indicates the changes of these four tasks' actual consumption and assigned budgets across the time. Different from Figure 22, in Figure 23 the x-axis denotes the time instead of release period numbers. It shows that small overrun faults can be tolerated for tasks with different criticality levels; when an overload condition is detected, the system can adapt quickly to the situation by suspending some servers in the order of low-criticality to high-criticality and ensure the task with the highest criticality, which is τ_2 in this case, has enough resource; as soon as the system recovers enough resource, the stopped servers start running again in the order of high-criticality to low-criticality. From Figure 23 one can also tell that if offline WCET is used for the tasks' budget value, τ_2 will receive processor bandwidth $\geq$47%, while the other tasks with lower criticalities will be sacrificed during the whole process.

Figure 24 indicates the overhead caused by the overload handling service in Evaluation 3. Concluded from Figures 16, 17, 18, 19, 20, 21, and 24, the overall overhead caused by all the services will not be greater than 2% of processor resource in the worst case.

12. Conclusion and Future Work

This paper has presented a design that incorporates adaptive reservation into a MCS. Based on the MCS model proposed, a resource adaptation method is used to adaptively change the reserved budget for each task and a slack distribution policy is employed to handle overrun faults. As shown in the evaluation, the overhead caused by the added layer does not take more than 2% of the processor resource. We believe that these values, although have space to be further lowered, are still acceptable in many practical circumstances.

The case studies show the following: the tasks' overrun ratios and deadline misses are bounded by user's predefined values for each criticality level; the system is aware of the available resource and guarantees lower criticality tasks a good chance to receive their required resource; the resource usage efficiency of the system is high during nonoverload condition; upon system overload, the tasks with higher criticality levels can always receive enough resource.

In future work, we want to improve the system by adding criticality considerations in the fault tolerance service which

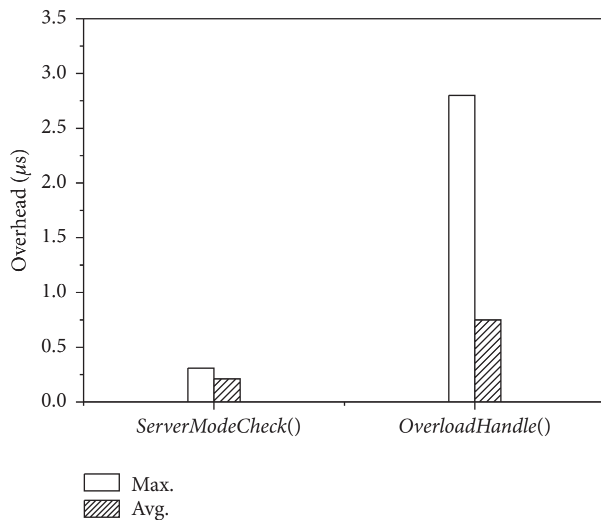


FIGURE 24: Overhead caused by overload handling.

means increasing the priority of higher criticality tasks in receiving slacks. As defined by the adopted slack distribution policy, only dynamic slack has been considered. In future work, we would like to also discover the potential applications of static slack (unreserved resources). We also plan to extend this work to other shared hardware resources, multiprocessor systems, and distributed systems.

Competing Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS '07)*, pp. 239–243, December 2007.

[2] A. Burns and R. Davis, "Mixed criticality systems—a review," Tech. Rep., Department of Computer Science, University of York, 2016, https://www-users.cs.york.ac.uk/~burns/review.pdf.

[3] M. Ali Awan, D. Masson, and E. Tovar, "Energy-aware task allocation onto unrelated heterogeneous multicore platform for mixed criticality systems," in *Proceedings of the 36th IEEE Real-Time Systems Symposium (RTSS '15)*, p. 377, IEEE, San Antonio, Tex, USA, December 2015.

[4] S. Baruah and Z. Guo, "Mixed-criticality scheduling upon varying-speed processors," in *Proceedings of the IEEE 34th Real-Time Systems Symposium (RTSS '13)*, Vancouver, Canada, December 2013.

[5] F. Broaekaert, A. Fristsch, L. San, and S. Tverdyshev, "Towards power-efficient mixed critical systems," in *Proceedings of the 9th Florian Broekaert, Laurent San, Agnes Fritsch (Thales Communications & Security) Sergey Tverdyshev (SYSGO) OSPERT Workshop*, pp. 30–35, July 2013.

[6] P. Haririan and A. Garcia-Ortiz, "A framework for hardware-based dvfs management in multicore mixed-criticality systems," in *Proceedings of the 10th International Symposium on IEEE Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC '15)*, pp. 1–7, Bremen, Germany, 2015.

[7] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele, "Energy efficient DVFS scheduling for mixed-criticality systems," in *Proceedings of the 14th International Conference on Embedded Software (EMSOFT '14)*, New Delhi, India, October 2014.

[8] X. Zhang, J. Zhan, W. Jiang, Y. Ma, and K. Jiang, "Design optimization of security-sensitive mixed-criticality real-time embedded systems," in *Proceedings of the 1st Workshop on Real-Time Mixed Criticality Systems (ReTiMiCS '13)*, 2013.

[9] A. Addisu, L. George, V. Sciandra, and M. Agueh, "Mixed criticality scheduling applied to jpeg2000 video streaming over wireless multimedia sensor networks," in *Proceedings of the 1st Workshop on Mixed Criticality Systems (WMC '13), and IEEE Real-Time Systems Symposium (RTSS '13)*, pp. 55–60, December 2013.

[10] H. Ahmadian and R. Obermaisser, "Time-triggered extension layer for on-chip network interfaces in mixed-criticality systems," in *Proceedings of the 18th Euromicro Conference on Digital System Design (DSD '15)*, pp. 693–699, Madeira, Portugal, August 2015.

[11] N. Audsley, "Memory architectures for NoC-based real-time mixed criticality systems," in *Proceedings of the 1st Workshop on Mixed Criticality Systems*, pp. 37–42, IEEE, Vancouver, Canada, December 2013.

[12] S. Baruah and A. Burns, "Achieving temporal isolation in multiprocessor mixed criticality systems," in *Proceedings of the 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 21–26, 2014.

[13] A. Burns, J. Harbin, and L. S. Indrusiak, "A wormhole NoC protocol for mixed criticality systems," in *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS '14)*, pp. 184–195, IEEE, Rome, Italy, December 2014.

[14] G. Carvajal and S. Fischmeister, "An open platform for mixed-criticality real time ethernet," in *Proceedings of the IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE '13)*, pp. 153–156, 2013.

[15] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *Proceedings of the 13th International Conference on Embedded Software (EMSOFT '13)*, 15, 1 pages, Quebec, Canada, October 2013.

[16] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Systems*, vol. 52, no. 4, pp. 399–449, 2016.

[17] T. Hollstein, S. P. Azad, T. Kogge, and B. Niazmand, "Mixed-criticality NoC partitioning based on the NoCDepend dependability technique," in *Proceedings of the 10th International Symposium on Reconfigurable and Communication-centric Systems-on-Chip (ReCoSoC '15)*, Bremen, Germany, July 2015.

[18] R. Obermaisser, Z. Owda, M. Abuteir, H. Ahmadian, and D. Weber, "End-to-end real-time communication in mixed-criticality systems based on networked multicore chips," in *Proceedings of the 17th Euromicro Conference on Digital System Design (DSD '14)*, pp. 293–302, Verona, Italy, August 2014.

[19] W. Steiner, "Synthesis of static communication schedules for mixed-criticality systems," in *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented*

Real-Time Distributed Computing Workshops (ISORCW '11), pp. 11–18, IEEE, Newport Beach, Calif, USA, March 2011.

[20] Y.-S. Kim and H.-W. Jin, "Towards a practical implementation of criticality mode change in RTOS," in *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '14)*, September 2014.

[21] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized Mixed-criticality sporadic task systems," *Real-Time Systems*, vol. 50, no. 1, pp. 48–86, 2014.

[22] S. Baruah, V. Bonifaci, G. D'Angelo et al., "Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems," *Journal of the ACM*, vol. 62, no. 2, article 14, 2015.

[23] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "Mixed-criticality scheduling of sporadic task systems," in *Algorithms—ESA 2011*, vol. 6942 of *Lecture Notes in Comput. Sci.*, pp. 555–566, Springer, Heidelberg, Germany, 2011.

[24] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS '11)*, pp. 34–43, Vienna, Austria, December 2011.

[25] T. Park and S. Kim, "Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems," in *Proceedings of the 9th ACM International Conference on Embedded Software*, pp. 253–262, ACM, 2011.

[26] S. Baruah and B. Chattopadhyay, "Response-time analysis of mixed criticality systems with pessimistic frequency specification," in *Proceedings of the IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '13)*, pp. 237–246, Taipei, Taiwan, August 2013.

[27] S. Baruah and A. Burns, "Implementing mixed criticality systems in ada," in *Reliable Software Technologies—Ada-Europe 2011*, pp. 174–188, Springer, 2011.

[28] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed criticality systems," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '10)*, pp. 13–22, April 2010.

[29] A. Burns and S. Baruah, *Timing Faults and Mixed Criticality Systems*, Springer, Berlin, Germany, 2011.

[30] A. Burns and R. I. Davis, "Mixed criticality on controller area network," in *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS '13)*, pp. 125–134, IEEE, Paris, France, July 2013.

[31] F. Dorin, P. Richard, M. Richard, and J. Goossens, "Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities," *Real-Time Systems*, vol. 46, no. 3, pp. 305–331, 2010.

[32] T. Fleming, *Extending mixed criticality scheduling [Ph.D. thesis]*, University of York, 2013.

[33] H.-M. Huang, C. Gill, and C. Lu, "Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4, article 126, 2014.

[34] M. Neukirchner, P. Axer, T. Michaels, and R. Ernst, "Monitoring of workload arrival functions for mixed-criticality systems," in *Proceedings of the IEEE 34th Real-Time Systems Symposium (RTSS '13)*, pp. 88–96, Vancouver, Canada, December 2013.

[35] D. De Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *Proceedings of the Real-Time Systems Symposium (RTSS '09)*, pp. 291–300, December 2009.

[36] D. De Niz, L. Wrage, A. Rowe, and R. R. Rajkumar, "Utility-based resource overbooking for cyber-physical systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5, article 162, 2014.

[37] V. Sciandra, P. Courbin, and L. George, "Application of mixed-criticality scheduling model to intelligent transportation systems architectures," *ACM SIGBED Review*, vol. 10, no. 2, p. 22, 2013.

[38] Q. Zhao, Z. Gu, and H. Zeng, "Integration of resource synchronization and preemption-thresholds into EDF-based mixed-criticality scheduling algorithm," in *Proceedings of the IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '13)*, pp. 227–236, Taipei, Taiwan, August 2013.

[39] Q. Zhao, Z. Gu, and H. Zeng, "PT-AMC: integrating preemption thresholds into mixed-criticality scheduling," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 141–146, EDA Consortium, 2013.

[40] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Proceedings of the 16th Design, Automation and Test in Europe Conference and Exhibition (DATE '13)*, March 2013.

[41] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS '08)*, pp. 147–155, IEEE, Prague, Czech Republic, July 2008.

[42] G. Lipari and G. Buttazzo, "Resource reservation for mixed criticality systems," in *Proceedings of the Workshop on Real-Time Systems: The Past, The Present, and the Future*, pp. 60–74, York, UK, March 2013.

[43] F. Guan, L. Peng, L. Perneel, and M. Timmerman, "Open source FreeRTOS as a case study in real-time operating system evolution," *Journal of Systems and Software*, vol. 118, pp. 19–35, 2016.

[44] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *Proceedings of the 23rd Real-Time Systems Symposium (RTSS '02)*, pp. 279–288, Austin, Tex, USA, December 2002.

[45] J. Hansen, S. Hissam, and G. A. Moreno, "Statistical-based WCET estimation and validation," in *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET '09)*, June 2009.

[46] S. Altmeyer and R. I. Davis, "On the correctness, optimality and precision of static probabilistic timing analysis," in *Proceedings of the 17th Design, Automation and Test in Europe (DATE '14)*, IEEE, March 2014.

[47] F. J. Cazorla, E. Quiñones, T. Vardanega et al., "Proartis: probabilistically analyzable real-time systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2, supplement, article 94, 2013.

[48] J. Abella, C. Hernandez, E. Quinones et al., "WCET analysis methods: pitfalls and challenges on their trustworthiness," in *Proceedings of the 10th IEEE International Symposium on Industrial Embedded Systems (SIES '15)*, pp. 39–48, Siegen, Germany, June 2015.

[49] Y. Zhu and F. Mueller, "Feedback EDF scheduling exploiting dynamic voltage scaling," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04)*, pp. 84–93, May 2004.

[50] A. Abbas, E. Grolleau, M. Loudini, and D. Mehdi, "A real-time feedback scheduler for environmental energy harvesting," in

Proceedings of the 3rd International Conference on Systems and Control (ICSC '13), Algiers, Algeria, October 2013.

[51] S. Groesbrink, L. Almeida, M. De Sousa, and S. M. Petters, "Towards certifiable adaptive reservations for hypervisor-based virtualization," in *Proceedings of the 20th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '14)*, pp. 13–24, Berlin, Germany, April 2014.

[52] A. Burns, "System mode changes-general and criticality-based," in *Proceedings of the 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 3–8, 2014.

[53] A. Easwaran and I. Shin, "Compositional mixed-criticality scheduling," in *Processings of the 7th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Rome, Italy, December 2014.

[54] P. Graydon and I. Bate, "Safety assurance driven problem formulation for mixed-criticality scheduling," in *Proceedings of the 1st International Workshop on Mixed Criticality Systems (WMC '13), and IEEE Real-Time Systems Symposium (RTSS '13)*, pp. 19–24, December 2013.

[55] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: framework, modeling, and algorithms," *Real-Time Systems*, vol. 23, no. 1-2, pp. 85–126, 2002.

[56] R. Santos, G. Lipari, E. Bini, and T. Cucinotta, "On-line schedulability tests for adaptive reservations in fixed priority scheduling," *Real-Time Systems*, vol. 48, no. 5, pp. 601–634, 2012.

[57] O. Naseer, A. Shah, and A. A. Khan, "Feedback control scheduling for crane control system," in *Proceedings of the 20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS '13)*, pp. 187–195, IEEE, Scottsdale, Ariz, USA, April 2013.

[58] G. Buttazzo and L. Santinelli, "Adaptive mechanisms for component-based real-time systems," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '15)*, pp. 1–8, Montreal, Canada, June 2015.

[59] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98)*, pp. 4–13, IEEE, 1998.

[60] S. Banachowski, T. Bisson, and S. A. Brandt, "Integrating best-effort scheduling into a real-time system," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, Lisbon, Portugal, December 2004.

[61] G. Lipari and S. Baruah, "Greedy reclamation of unused bandwidth in constant-bandwidth servers," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (RTS '00)*, pp. 193–200, IEEE, Stockholm, Sweden, June 2000.

[62] S. A. Brandt, S. Banachowski, C. Lin, and T. Bissom, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS '03)*, pp. 396–407, December 2003.

[63] G. Bernat, I. Broster, and A. Burns, "Rewriting history to exploit gain time," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, Lisbon, Portugal, December 2004.

[64] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS '00)*, pp. 295–304, August 2002.

[65] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "IRIS: a new reclaiming algorithm for server-based real-time systems," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04)*, pp. 211–218, Toronto, Canada, May 2004.

[66] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, December 2005.

[67] L. Abeni, L. Palopoli, and G. Buttazzo, "On adaptive control techniques in real-time resource allocation," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS '00)*, Sweden, Europe, June 2000.

[68] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS '02)*, pp. 71–80, Austin, Tex, USA, December 2002.

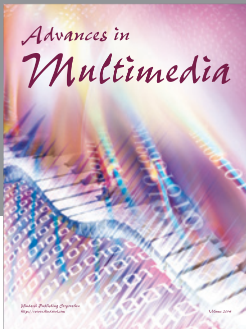
[69] D. Fontanelli, L. Greco, and L. Palopoli, "Soft real-time scheduling for embedded control systems," *Automatica*, vol. 49, no. 8, pp. 2330–2338, 2013.

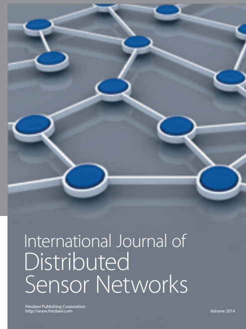
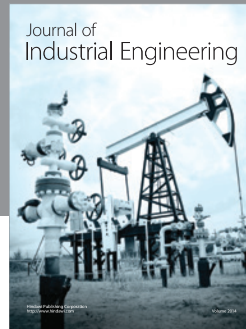
[70] G. Lipari and L. Palopoli, "Real-time scheduling: from hard to soft real-time systems," https://arxiv.org/abs/1512.01978.

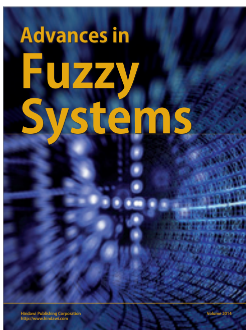
[71] L. Palopoli, L. Abeni, T. Cucinotta, G. Lipari, and S. K. Baruah, "Weighted feedback reclaiming for multimedia applications," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '08)*, pp. 121–126, October 2008.

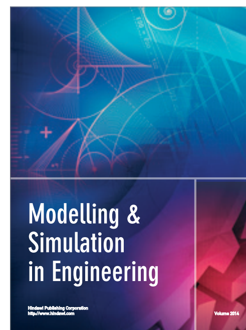
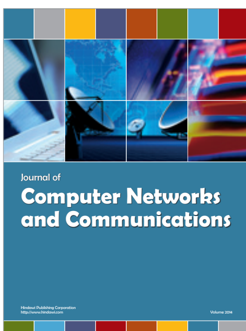
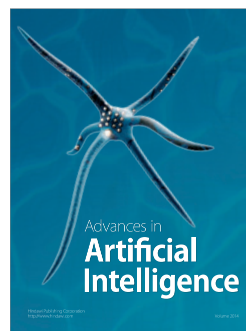
[72] T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari, "A robust mechanism for adaptive scheduling of multimedia applications," *ACM Transactions on Embedded Computing Systems*, vol. 10, no. 4, article 46, 2011.

[73] X. Gu, A. Easwaran, K.-M. Phan, and I. Shin, "Resource efficient isolation mechanisms in mixed-criticality scheduling," in *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS '15)*, July 2015.

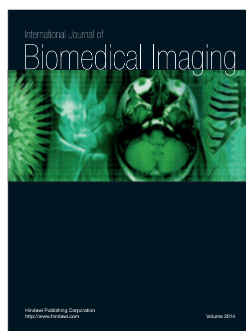
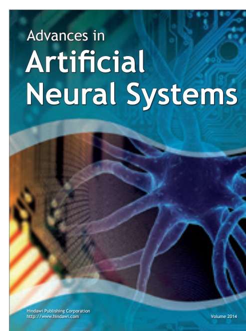
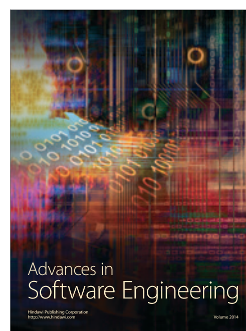

Advances in
Multimedia

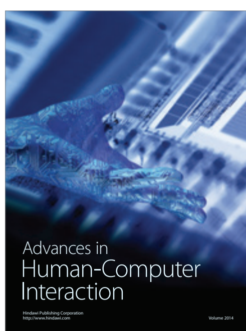
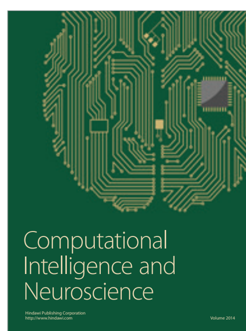
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

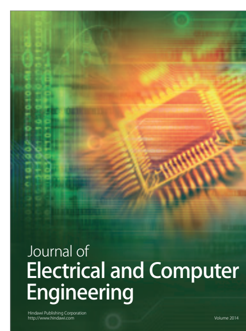
Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
https://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering