PASS: Priority Assignment of Real-Time Tasks with Dynamic Suspending Behavior under Fixed-Priority Scheduling

Wen-Hung Huang and Jian-Jia Chen
Department of Computer Science
TU Dortmund University, Germany
{wen-hung.huang, jia.chen}@tu-dortmund.de

Husheng Zhou and Cong Liu
Department of Computer Science
The University of Texas at Dallas
{husheng.zhou,cong}@utdallas.edu

ABSTRACT

Self-suspension is becoming an increasingly prominent characteristic in real-time systems such as: (i) I/O-intensive systems, where applications interact intensively with I/O devices, (ii) multi-core processors, where tasks running on different cores have to synchronize and communicate with each other, and (iii) computation offloading systems with coprocessors, like Graphics Processing Units (GPUs). In this paper, we show that *rate-monotonic* (RM), *deadline-monotonic* (DM) and *laxity-monotonic* (LM) scheduling will perform rather poor in dynamic self-suspending systems in terms of speed-up factors. On the other hand, the proposed *PASS* approach is guaranteed to find a feasible priority assignment on a speed-2 uniprocessor, if one exists on a unit-speed processor. We evaluate the feasibility of the proposed approach via a case study implementation. Furthermore, the effectiveness of the proposed approach is also shown via extensive simulation results.

Keywords

Self-suspension, schedulability, priority assignment

1. INTRODUCTION

In many real-time and embedded systems, tasks may be suspended by the operating system when accessing external devices such as disks, graphical processing units (GPUs), or synchronizing with other tasks. This behavior is often known as self-suspension. Self-suspensions are even more pervasive in many emerging embedded cyber-physical systems in which the computational components frequently interact with external and physical devices [8, 9]. Typically, the resulting suspension delays range from a few microseconds (e.g., a write operation on a flash drive [8]) to a few hundreds of milliseconds (e.g., offloading computation to GPUs [9, 15]). Such suspension delays negatively impact the timing predictability and cause intractability in hard real-time (HRT) schedulability analysis [18].

The unsolved problem of efficiently supporting self-suspensions in real-time systems has impeded research progress on many related research topics such as predictably supporting I/O-intensive applications and computation offloading. Since the problem of scheduling HRT self-suspending task systems on a uniprocessor is $\mathcal{NP}$-hard in the strong sense [18], it is unlikely that an optimal polynomial-time solution exists. As an appealing choice, approximation algorithms, particularly those based on speed-up factors, have attracted much attention [16]. If a scheduling algorithm $\mathcal{A}$ has a speed-up factor α, then $\mathcal{A}$ guarantees to produce a feasible schedule (i.e., all task deadlines are met) for a given input task set on a processor with speed α, if this task set admits a feasible schedule on a unit-speed processor. Thus, designing scheduling algorithms with provable speed-up factors ensures their qualities for such $\mathcal{NP}$-hard problems.

Classical fixed-priority scheduling algorithms, such as *rate-monotonic* (RM) and *deadline-monotonic* (DM) which are widely used in practice, have been shown to be effective in supporting ordinary non-suspending real-time task systems [12]. Unfortunately, their quality in handling self-suspending tasks is generally unknown. In this work, we show that classical fixed-priority scheduling algorithms become rather pessimistic for supporting self-suspending task systems. As a better alternative, we propose PASS: *a Priority Assignment algorithm for Self-Suspending systems*. We also present the corresponding analysis of PASS and show that PASS yields strong performance in terms of non-trivial speed-up factors.

Overview of related work. An overview of work on scheduling self-suspending task systems can be found in [14]. In [14], a general interference-based analysis framework was developed that can be applied to derive sufficient utilization-based tests for self-suspending task systems on uniprocessors when RM scheduling is applied. [4] shows that the category of fixed-relative-deadline schedulers may yield non-trivial resource augmentation (speed-up factor) performance guarantees. However, the result in [4] can only be applied to a special self-suspending task model, where each task is allowed to suspend for at most once and has a fixed interleaving pattern between computation and suspension phases. Given that the occurrence of most I/O-induced suspensions is unpredictable, the result from [4] can hardly be put into practice. In [11], an integer linear programming (ILP)-based priority assignment algorithm is proposed for scheduling self-suspending task systems with pre-fixed phase interleaving patterns on a uniprocessor under fixed-priority scheduling. However, this approach cannot be applied to the general self-suspending task model, and may incur excessively high runtime complexity due to the ILP-based solution. In summary, most of the existing schemes suffer from two major problems: (*i*) the fully general self-suspending task model (defined in Sec. 2) is not supported, and/or (*ii*) the quality of these tests is unknown.

Contribution. In this paper, we advance the state of the art on supporting HRT self-suspending task systems on a uniprocessor under fixed-priority scheduling. We first prove

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
DAC '15 June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06..$15.00.
http://dx.doi.org/10.1145/2744769.2744891.

that classical fixed-priority scheduling algorithms, RM, DM, and *laxity-monotonic* (LM), have a speed-up factor of ∞, which implies that their corresponding priority assignment schemes are ineffective when self-suspensions are present. To better support self-suspending task systems, we present a new fixed-priority scheduling algorithm PASS for the general self-suspending task model. PASS is proven to yield a speed-up factor of two, and thus guarantees to find a feasible priority assignment on a speed-2 uniprocessor in pseudo-polynomial time, if one exists on a unit-speed processor. The effectiveness of PASS has been validated via both a GPU-offloading case study implemented on top of real hardware and extensive simulation results using traces as well as randomly generated workload parameters.

To the best of our knowledge, this is the first technique that can efficiently analyze the general self-suspending task system with non-trivial speed-up factors.

Organization. The rest of this paper is organized as follows: Section 2 describes the system model. Section 3 presents a motivational example of well-known scheduling and our proposed priority assignment as well as its speed-up factor analysis. Section 4 studies the real-time matrix calculation read-write application, and Section 5 describes our extensive experiments. Finally, Section 6 concludes this paper.

2. SYSTEM MODEL

The general *self-suspending sporadic* (SSS) task model extends the conventional sporadic task model by allowing tasks to suspend themselves. Similar to sporadic tasks, a self-suspending sporadic task releases jobs sporadically, but each job of the task can alternate between computation and suspension phases.

We consider a real-time system to execute a set of n independent, preemptive, self-suspending real-time tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ on a uniprocessor. Each task can release an infinite number of jobs under the given minimum inter-arrival time (temporal) constraint. A self-suspending task τ_i is characterized by a 4-tuple (C_i, S_i, T_i, D_i), T_i denotes the minimum inter-arrival time of τ_i, each of job of τ_i has a relative deadline D_i, C_i denotes the upper bound on total execution time of each job of τ_i, and S_i denotes the upper bound on total suspension time of each job of τ_i. We assume that $C_i + S_i \leq D_i$ for any task $\tau_i \in \tau$.

The utilization of task τ_i is defined as $U_i = C_i/T_i$. We further assume that $\sum_{i=1}^{n} U_i \leq 1$. Task system τ is said to be an *implicit-deadline* system if $D_i = T_i$ holds for each τ_i, and a *constrained-deadline* system if $D_i \leq T_i$ holds for each τ_i; otherwise, an *arbitrary-deadline* system. We restrict our attention here to *constrained-deadline* task systems. A system τ is said to be *feasible* if there exists a scheduling algorithm that can schedule the system without any deadlines being missed.

This model allows tasks to suspend with an unlimited number of suspensions and at arbitrary locations, within each job. (Starting or ending with a suspension phase, of each job, is also permitted.) The number of suspension and their locations can also vary, with respect to job releases. From application perspectives, such a general suspension model is more appropriate due to the unpredictable nature of I/O operations.

In this paper, we focus on fixed-priority scheduling, in which each task is associated with a unique priority. More precisely, all the jobs of a task have the same priority level, and the system always selects the job in the ready queue with the highest-priority level to execute. Clearly, if a job suspends itself, it is no longer in the ready queue. On the other hand, when a job resumes from its self-suspension, it is put into the ready queue again.

3. PRIORITY ASSIGNMENT

In this section we propose our priority assignment approach for self-suspending systems as well as its processor speed-up factor.

3.1 Speed-Up Factor of DM and LM

In the light of simple implementation in kernels, fixed-priority scheduling has been widely adopted in commercial real-time systems. Classical *deadline-monotonic* (DM)/*rate-monotonic* (RM) scheduling algorithm is *optimal* in the sense that if there exists a feasible fixed-priority assignment in a sporadic constrained-deadline/implicit-deadline real-time system without self-suspensions, then it can be scheduled by DM/RM. Nevertheless, in self-suspending systems it has been shown in [4] that no lower bound on processor speed-up factor is guaranteed for RM scheduling.

The alternative, called *laxity-monotonic* (LM) scheduling, assigns the highest priority to the task with the least laxity, that is, $D_i - S_i$. This algorithm can effectively relieve the problem that the long-suspension-length task is prone to miss its deadline under RM scheduling, and seems to provide better performance than RM.

Unfortunately, the following theorem shows that the above algorithms yield rather poor performance in the presence of self-suspensions with respect to the speed-up factors.

THEOREM 1. *The speed-up factor for RM, DM, and LM scheduling is* ∞.

PROOF. Consider the following implicit-deadline task set with one SSS task and one sporadic task:

- $C_1 = 1 - 2\epsilon$, $S_1 = 0$, $T_1 = 1$
- $C_2 = \epsilon$, $S_2 = T - 1 - \epsilon$, $T_2 = T$

where T is any natural number larger than 1 and ϵ can be arbitrary small. It is clear that this task set is schedulable if we assign higher priority to task τ_2 than task τ_1.

Under either RM, DM, and LM scheduling, task τ_1 has higher priority than task τ_2. Thanks to the harmonic system, the schedulability of task τ_2 can be analyzed by examining the demand from task τ_1 together with its upper bound on total execution time and suspension length at its deadline, as the two tasks release simultaneously. Thus, jobs of τ_2 will miss deadlines since $T(1-2\epsilon)+\epsilon+(T-1-\epsilon) > T$.

In order to be schedulable upon this system under RM, DM, or LM on α-speed processor the following must hold[1]:

$$\frac{T(1-2\epsilon)+\epsilon}{\alpha} + (T - 1 - \epsilon) \leq T$$

After reformulation, we have $\alpha \geq \frac{T(1-2\epsilon)+\epsilon}{1+\epsilon}$. Thus, $\alpha \to \infty$ as $T \to \infty$. $\square$

Apart from the poor performance of RM, DM, and LM scheduling, considering all $n!$ possible priority ordering for finding a feasible priority assignment is computationally intractable.

[1]On a speed-α uniprocessor the worst-case execution times C_i become $\frac{C_i}{\alpha}$. However, S_i remains the same.

3.2 Sufficient Condition and Proposed Test

Before proceeding with our approach, we formally define a priority assignment that determines the precedence of tasks under fixed-priority scheduling, as follows:

Definition 1 (priority assignment). *Let π be the priority assignment as a bijective function $\pi : \tau \to \{1, 2, \ldots, |\tau|\}$ to define the priority level of task $\tau_i \in \tau$. Priority levels are numbered from 1 to $|\tau|$ where 1 is the highest and $|\tau|$ the lowest.*

We here provide a sufficient condition that checks whether a priority assignment π in a self-suspending system τ is feasible or not, in the following theorem.

Theorem 2. *A priority assignment π is feasible in a system τ consisting of periodic, independent, preemptable, self-suspending tasks if for all tasks $\tau_i \in \tau$, there exists a time-instant t with $0 < t \leq D_i$ such that*

$$C_i + S_i + \sum_{j:\pi(\tau_j)<\pi(\tau_i)} \widehat{W}_j(t) \leq t \quad (1)$$

where

$$\widehat{W}_j(t) = \left\lceil \frac{t + D_j}{T_j} \right\rceil C_j \quad (2)$$

Proof. Let t_0 denote the release time of task τ_i. In the interval $[t_0, t_0 + D_i]$, the total capacity occupied by task τ_i is at most $C_i + S_i$.

Within the interval $[t_0, t_0 + t]$, the demand from higher-priority task τ_j can be decomposed into two parts: (i) a carry-in job: a job of τ_j having the release time prior to t and the absolute deadline after t, and (ii) body jobs: jobs of τ_j having the release time within $[t_0, t_0 + t]$. The releases are also illustrated in Figure 1.

Due to the fact that each job can at most contribute to its upper bound on total execution time, we can bound the interference from the carry-in job and body jobs by C_j and $\left\lceil \frac{t}{T_j} \right\rceil C_j$, respectively, as the releases of body jobs demand completely all their execution times.

Let t_j^c denote the release time of the carry-in job of task τ_j, and Δ_j denote $t_0 - t_j^c$. Since the carry-in job must execute C_j units prior to $t_j^c + D_j$ in order to meet its deadline, there is no interference from task τ_j within $[t_j^c + D_j, t_j^c + T_j]$.

We then consider the demand from τ_j within the interval with respect to Δ_j: $\forall t > 0$, $\forall 0 < \Delta_j \leq D_i$,

$$\left\lceil \frac{t + D_j}{T_j} \right\rceil C_j \geq \left\lceil \frac{t - (T_j - \Delta_j)}{T_j} \right\rceil C_j + C_j$$

which implies that releasing the carry-in job at time $t_0 - D_j$ results in the maximum demand, for any t. At any time $0 < t \leq D_i$ the demand from each higher-priority task τ_j is bounded above by $\widehat{W}_j(t)$, and the capacity occupied by task τ_i is at most $C_i + S_i$. From Eq. (1) it therefore follows that all tasks meet their deadline, and priority assignment π is feasible in system τ, which concludes the proof. □

It is clear that the sufficient test by Theorem 2 runs in pseudopolynomial time $O(nD_i)$.

Priority assignment. It has been shown in [2] that it suffices to examine a polynomial number of priority orderings in periodic systems for finding a feasible priority ordering. Checking the fixed-priority feasibility of a self-suspending task set can be achieved by using the method, called *Optimal Priority Assignment (OPA) Algorithm* [1, 2, 5]. The

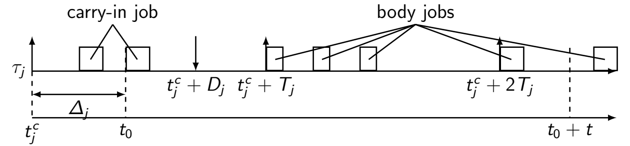

Figure 1: The illustration of releases for the sufficient condition.

OPA algorithm assigns each priority level k to one of unassigned tasks that has no deadline miss along with the other unassigned task assumed to have higher priorities. The iterative priority assignment terminates as soon as either no unassigned task can be assigned at priority level k or all priority levels are assigned. For a schedulability test to be compatible with the OPA algorithm, it must comply with three conditions provided in [5]. For completeness, we state these conditions as follows:

- **Condition 1.** The schedulability of a task τ_i may, according to test S, depend on any independent properties of tasks with priorities higher than k, but not on any properties of those tasks that depend on their relative priority ordering.
- **Condition 2.** The schedulability of a task τ_i may, according to test S, depend on any independent properties of tasks with priorities lower than k, but not on any properties of those tasks that depend on their relative priority ordering.
- **Condition 3.** When the priorities of any two tasks of adjacent priority are swapped, the task being assigned the higher priority cannot become unschedulable according to test S, if it was previously schedulable at the lower priority.

It is not difficult to see that the sufficient schedulability test by Theorem 2 complies with the required conditions for the OPA algorithm. We state this with the following lemma.

Lemma 1. *The sufficient schedulability test by Theorem 2 complies with Condition 1-3.*

Proof. Inspection of Eq. (1) shows that the schedulability of τ_i depends on the set of higher-priority tasks but not on their relative priority ordering, hence Condition 1 holds.

It is obvious that the schedulability of τ_i testing by Eq. (1) has no dependency on the set of tasks with lower priority than k, hence Condition 2 holds.

Consider two tasks τ_a and τ_b initially at priorities k and $k+1$, respectively. If task τ_b is schedulable, it is still schedulable when it is shifted up one priority level to priority k, since the only change of higher-priority task demand is the removal of task τ_a from the set of tasks that have higher priority than task τ_b. Hence, Condition 3 holds. □

Algorithm 1 shows the proposed *PASS* approach adopting the OPA algorithm for the feasible priority assignment in self-suspending systems.

Computational complexity. Checking the feasibility for each priority assignment ("if" statement) is computable in pseudo-polynomial time. Like in [2], at most $n(n+1)/2$ priority orderings need to be checked until either a feasible priority assignment is found or does not exist. Therefore, PASS runs in pseudo-polynomial time.

Lemma 1 suggests the following corollary that is useful for deriving the speed-up factor of the proposed PASS in the remaining section.

Algorithm 1: PASS Approach

input : A set of self-suspending tasks τ
output: Priority assignment π and the feasibility of system τ

```
π ← ∅;
for each priority k from |τ| to 1 do
    for each unassigned task τ_i do
        if task τ_i is schedulable at priority k with all
        unassigned tasks (assume them as higher-priority
        tasks) according to Eq (1) then
            π(τ_i) ← k // assign task τ_i to priority k
            break (continue the outer loop) ;
    return "unscheduable";
return "scheduable";
```

COROLLARY 1. *If there exist feasible priority assignments by testing Theorem 2, the proposed PASS returns one of them.*

PROOF. From Lemma 1 the sufficient test by Theorem 2 is compliant with the above conditions. Following the proof of Theorem 3 in [5], we here conclude this corollary. □

3.3 Speed-up Factor of 2 under Fixed-Priority Scheduling

We now determine the processor speedup factor of the proposed PASS based on checking our sufficient condition. First, we identify the necessary conditions for self-suspending systems in the following theorem.

THEOREM 3. *Suppose a system τ consisting of periodic, independent, preemptable, self-suspending tasks is schedulable on one processor under fixed-priority scheduling with a priority assignment π. For all tasks τ_i, there exists a time-instant t with $0 < t \leq D_i$ such that*

$$C_i + S_i + \sum_{j:\pi(\tau_j)<\pi(\tau_i)} W_j(t) \leq t \tag{3}$$

where

$$W_j(t) = \left\lceil \frac{t+S_j}{T_j} \right\rceil C_j \tag{4}$$

PROOF. We prove this theorem by contrapositive: *if there exists a task τ_i such that for all time-instant t with $0 < t \leq D_i$, Eq. (3) does not hold, then, τ is unschedulabe with priority assignment π.*

This is done by providing a concrete SSS system that is not schedulable by using π. Let t_0 be the time to release the first job of task τ_i. We then release higher-priority tasks τ_j at time $t_0 - S_j$. Let the first jobs of all higher-priority tasks τ_j be suspended for their entire suspension length as they release, i.e., from $t_0 - S_j$ to t_0. Then, the subsequent jobs of these tasks τ_j are released as early as possible, i.e., at $t_0 - S_j + T_j, t_0 - S_j + 2T_j$, etc. In other words, these subsequent jobs suspend themselves only after finishing their executions.

In the above pattern, for any interval in $(t_0, t_0 + D_i]$ in which no higher-priority jobs than task τ_i is executed, we can replace it from the earliest one by a suspension interval, of τ_i, that equals to the length of the interval, until consuming task τ_i's suspension length S_i.

As a result, we can observe that the condition $\forall 0 < t \leq D_i$ with $C_i + S_i + \sum_{j:\pi(\tau_j)<\pi(\tau_i)} W_j(t) > t$ implies that the job of task τ_i released at time t_0 in the above SSS system cannot be finished at any time $t_0 + t$, $\forall 0 < t \leq D_i$, which concludes the proof due to the contrapositive. □

LEMMA 2. *For all tasks τ_i, $\forall t > 0$,*

$$2W_i(t) \geq \widehat{W}_i(t)$$

where $\widehat{W}_i(t)$ and $W_i(t)$ are defined in Eq. (2) and Eq. (4).

PROOF. It is clear that for all time-instant t large than 0, $\left\lceil \frac{t}{T_i} \right\rceil C_i \leq W_i(t)$ and $\widehat{W}_i(t) \leq C_i + \left\lceil \frac{t}{T_i} \right\rceil C_i$. In addition, we know that $\forall t > 0$,

$$2\left\lceil \frac{t}{T_i} \right\rceil C_i \geq C_i + \left\lceil \frac{t}{T_i} \right\rceil C_i.$$

Observing the above inequalities, we can conclude the proof with simple logic. □

Putting all the pieces together, we derive a speed-up factor of 2 for the proposed PASS in the following theorem.

THEOREM 4. *If there exists a feasible priority assignment for a self-suspending system τ on a unit-speed processor, then the proposed approach can find one for system τ on a speed-2 uniprocessor.*

PROOF. Suppose that there exists a feasible priority assignment for a self-suspending system τ on a unit-speed processor.

Let A denote the *set* of priority assignments satisfying the necessary condition in Theorem 3 on a unit-speed processor. We denote B as the *set* of priority assignments compliant to the sufficient condition in Theorem 2 on a speed-2 processor. Since B is the set of priority assignments compliant to the sufficient condition for feasibility, all the priority assignments in B must be feasible.

From Theorem 3 we know that $\forall \pi \in A$, for all tasks τ_i, there exists a time-instant t with $0 < t \leq D_i$ such that

$$C_i + S_i + \sum_{j:\pi(\tau_j)<\pi(\tau_i)} W_i(t) \leq t$$

Besides, we have that for any π, $\forall t > 0$

$$C_i + S_i + \sum_{j:\pi(\tau_j)<\pi(\tau_i)} W_i(t) \geq_1 \frac{C_i}{2} + S_i + \frac{1}{2} \sum_{j:\pi(\tau_j)<\pi(\tau_i)} \widehat{W}_i(t)$$

where the right-handed side, of the above inequality, is the sufficient condition on a speed-2 processor and $\geq_1$ comes from Lemma 2. From this it follows that $A \subseteq B$.

Due to the existence of feasible priority assignments and the above relation on sets, it follows that $A, B \neq \emptyset$. From Corollary 1 we know that if $B \neq \emptyset$, the proposed approach can find a feasible priority assignment, on a speed-2 processor. We here conclude the proof. □

4. CASE STUDY

In this section, we show the effectiveness of RM, LM, and the proposed PASS approach via a case study implementation. We programmed the real-time matrix calculation read-write applications, which read matrices from disk, perform the matrix calculation, and write the result back to disk.

4.1 Implementation

Our case study was conducted with the Linux kernel 3.3.1 on NVIDIA GeForce GTX 480 graphics card and Intel i7 4790K processor. We used Gdev [10] as driver to perform general purpose computing on GPU. In order to propagate

Description	Small			Medium			Large		
Largest matrix	2048 square			2432 square			2560 square		
Total utilization	30.19%			33.98%			38.72%		
Scheduling polices	RM	LM	PASS	RM	LM	PASS	RM	LM	PASS
Feasibility	×	✓	✓	×	✓	✓	×	×	✓

Table 1: The feasibility for different scheduling policies with respect to different problem sizes.

priorities from threads to GPUs, we made some modifications on Gdev. To minimize the undesired influence from initialization, thread monitoring and task dispatching, we specify *CPU affinity* for each thread so that the thread will execute only on the designated CPUs. For example, monitoring and dispatching threads are designated to one dedicated processor, and pure computation threads to another one. We choose the SCHED_FIFO as the default real-time scheduler in operating system. The scheduling on GPUs is similar to priority-based SCHED_FIFO.

Matrix calculation is widely used for high performance computation, such as atom workload on CPUs and GPUs. Due to the high performance of GPUs on float computation and that of CPUs on general purpose operation, we decompose each task into three serialized stages: small integer matrix multiplication on CPUs, float matrix multiplication on GPUs, and integer matrix addition on CPUs. Such combination of matrix calculation is common in real-world image recognition programs [7].

The real-time matrix calculation system consists of 10 periodic, self-suspending tasks. We evaluate three scenarios for their performance with respect to different scheduling policies, as shown in Table 1. We use the analysis in [17] to derive the upper bound on the offloading computation time in multi-tasking environment. Each task is associated with the priority according to its fixed-priority scheduling policy, i.e., RM, LM, and PASS. In the small problem size scenario, execution period of all 10 tasks varies from 5000ms to 9500ms and the total utilization is 30.19%. Each CPU-GPU-CPU task performs calculation on matrix with different sizes, among which the largest matrix size is 2048×2048. With task periods fixed, we only scale up the matrix size to generate the medium and large problem size configurations. In the medium and large configurations, the total utilization is 33.98% and 38.72% respectively. The corresponding largest matrix size is 2432×2432 and 2560×2560.

4.2 Performance Evaluation

As shown in Table 1, the proposed PASS can successfully schedule all tasks for the three problem sizes, whereas the large problem size is unschedulable under LM scheduling, and especially none of scenarios is schedulable under RM scheduling, according to Theorem 2.

We report the maximum response time among jobs during running for 30 minutes. Due to page limitation, we only show the maximum response time of 4 tasks in Figure 2. Other tasks has similar performance. Obviously, the proposed PASS is able to provide a shorter maximum response time than RM and LM for all problem sizes. In summary, we empirically show that PASS is implementable in computation offloading systems with coprocessors and can achieve better performance over RM and LM.

5. EXPERIMENTAL RESULTS

In this section, we conduct extensive experiments using synthesized task sets for evaluating the proposed *PASS*. The effectiveness of the proposed *PASS* was evaluated by comparing to RM and LM scheduling. We also evaluate an upper bound based on the *necessary* condition from Theorem 3,

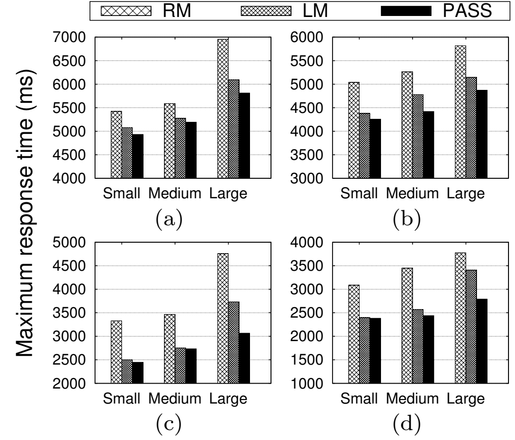

Figure 2: Maximum response time of 4 tasks under different scheduling polices for three problem sizes, from which task periods are (a) 9500ms (b) 8000ms (c) 5500ms (d) 5000ms.

denoted by Necessary Condition (NC).

The metric to compare results is to measure the *acceptance ratio* of the above tests with respect to a given goal of task set utilization. We generate 100 task sets for each utilization level. The acceptance ratio of a level is said to be the number of task sets that are deemed schedulable divided by the number of task sets for this level, i.e., 100.

5.1 Simulation Setup

We first generated a set of sporadic tasks. The cardinality of the task set was 10. The UUniFast method [3] was adopted to generate a set of utilization values with the given goal. We here used the approach suggested by Davis and Burns [6] to generate the task period according to an exponential distribution. The distribution is of two orders of magnitude, i.e., the ratio of the maximum and the minimum possible task periods was 100. The execution time was set accordingly, i.e., $C_i = T_i U_i$. Task relative deadlines were implicit, i.e., $D_i = T_i$. Note that the following results remains similar if task relative deadlines were constrained.

We then converted a proportion p of sporadic tasks to SSS tasks. Suspension lengths of tasks were then generated in a similar manner to the method used in [13]. Suspension lengths of tasks were assigned according to a uniform random distribution, in one of three ranges depending on the suspension type (sstype): $[0.01(T_i - C_i), 0.1(T_i - C_i)]$ (short suspensions, sstype=S in Figure 3), $[0.1(T_i - C_i), 0.6(T_i - C_i)]$ (moderate suspensions, sstype=M in Figure 3), and $[0.6(T_i - C_i), T_i - C_i]$ (long suspensions, sstype=L in Figure 3).

5.2 Results

Figure 3 presents the result for the performance in terms of the acceptance ratio. It is clear that our proposed *PASS* is superior to *RM* and *LM*. For all the tests, the feasibility is inversely proportional to the proportion p, of self-suspension tasks. With short suspension length (Figure 3a, 3b, and 3c), all tests can sustain the feasibility with a utilization of up to 90%. Due to the little impact of short suspension length, the improvement of the proposed *PASS* over *RM* and *LM* scheduling is not significant but still visible. With moderate suspension length (Figure 3d, 3e, and 3f), the proposed *PASS* can achieve significantly improvement over *RM* and *LM* scheduling. When the suspension length is long (Figure 3g, 3h, and 3i), we can observe that the acceptance ratio drops significantly. This is due to the fact

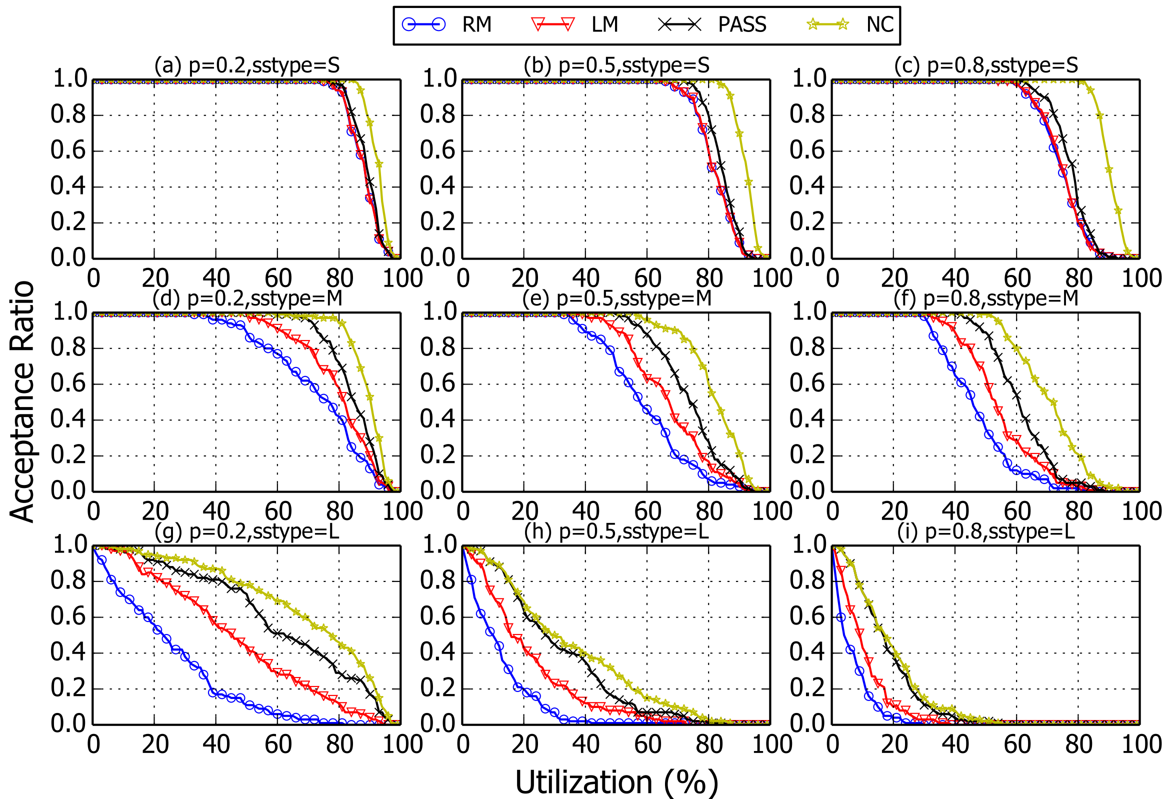

Figure 3: Comparison with different types of the suspension length and proportions of SSS tasks.

that the long-suspension task is by nature prone to miss its deadline. As a result, the space between the optimal scheduling and the proposed *PASS* for the improvement is limited. This can be observed from the task set with high proportion of long-length self-suspending tasks, as seen in Figure 3h and 3i, in which PASS is rather close to the upper bound, NC. On the other hand, a significant improvement can be seen in the task set with low proportion of long self-suspending length, as shown in Figure 3g. One can imagine that few worst-execution times from such tasks can interference with the other tasks. Consequently, a feasible assignment is still achievable once these tasks are associated with higher-priority.

6. CONCLUSION

In this work we propose PASS for priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. We show that there is no lower bound on processor speed-up factor for the classical RM, DM, and LM scheduling in self-suspending systems. The proposed approach is guaranteed to find a feasible priority assignment on a speed-2 uniprocessor, if one exists on a unit-speed processor. Also, the results via the GPU offloading implementation and extensive simulations suggest that the proposed approach can be effectively put into practice.

7. REFERENCES

[1] N. C. Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times.* Citeseer, 1991.

[2] N. C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[3] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[4] J.-J. Chen and C. Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Real-Time Systems Symposium (RTSS)*, 2014.

[5] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[6] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *Computers, IEEE Transactions on*, 57(9):1261–1276, 2008.

[7] J. Inc. Deepbeliefsdk: The sdk for jetpac ios, android, linux, and os x deep belief image recognition framework. https://github.com/jetpacapp/DeepBeliefSDK.

[8] W. Kang, S. H. Son, J. A. Stankovic, and M. Amirijoo. I/o-aware deadline miss ratio management in real-time embedded databases. In *Real-Time Systems Symposium*, pages 277–287, 2007.

[9] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *Real-Time Systems Symposium (RTSS)*, pages 57–66, 2011.

[10] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. Gdev: First-class gpu resource management in the operating system. In *USENIX Annual Technical Conference*, 2012.

[11] J. Kim, B. Andersson, D. d. Niz, and R. R. Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *Real-Time Systems Symposium (RTSS)*, pages 246–257, 2013.

[12] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[13] C. Liu and J. H. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Real-Time Systems Symposium (RTSS)*, pages 425–436, 2009.

[14] C. Liu and J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *Real-Time Systems Symposium (RTSS)*, 2014.

[15] W. Liu, J. Chen, A. Toma, T. Kuo, and Q. Deng. Computation offloading by using timing unreliable components in real-time systems. In *Design Automation Conference (DAC)*, 2014.

[16] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *Proc. of the 29th ACM Symposium on Theory of Computing*, pages 140–149, 1997.

[17] G. Raravi and B. Andersson. Calculating an upper bound on the finishing time of a group of threads executing on a gpu: A preliminary case study. Technical report, IPP Hurray Research Group, 2010.

[18] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Real-Time Systems Symposium (RTSS)*, pages 47–56, 2004.