Online Mode Switch Algorithms for Maintaining Data Freshness in Dynamic Cyber-Physical Systems

Song Han, Kam-Yiu Lam, Deji Chen, *Member, IEEE*, Ming Xiong, Jiantao Wang, *Member, IEEE*, Krithi Ramamritham, *Fellow, IEEE*, and Aloysius K. Mok

Abstract—Maintaining the freshness of real-time data is one of the crucial design issues in cyber-physical systems (CPS). Past studies have focused on designing update algorithms to minimize the workload imposed by a fixed set of update tasks while ensuring the temporal validity of data. In this paper, we revisit this problem in *dynamic cyber-physical systems (DCPS)* which may exhibit multi-modal behavior. Any solution to this problem must recognize that: (1) different update algorithms may be needed in different modes according to the workload in each mode, and (2) temporal validity of data must be maintained not only in each mode but also during the mode switch. To strike a balance between data freshness and system schedulability, we propose a *utilization-based scheduling selection (UBSS)* strategy. We first introduce two synchronous mode switch algorithms, named *search-based switch (SBS)* and *adjustment-based switch (ABS)* to search for the proper switch point online and execute all update tasks in the new mode synchronously. *SBS* checks for temporal validity at the beginning time slot of each idle period in the schedule, while *ABS* relaxes this restriction through schedule adjustment. To support immediate mode switch, we propose an asynchronous switch algorithm named *instant switch (IS)* to reduce the switch delay. *IS* schedules outstanding jobs from the old mode together with the jobs in the new mode using the least-available-laxity-first scheduling policy. Our experimental results demonstrate the effectiveness of these three algorithms. They also show that UBSS strategy can significantly outperform a single fixed update algorithm in terms of maintaining better data freshness while incurring only limited online switch overhead.

Index Terms—Dynamic cyber-physical systems, real-time database, mode switch, temporal validity

1 Introduction

Maintaining data freshness is a key issue in many *Cyber-Physical Systems (CPS)* that depend on the timely processing of massive amount of real-time data [2], [3]. Real-time data track the current status of entities in the system and are typically sampled and stored in a real-time database system (RTDBS) [4]. A sampled data value is valid only for a certain time interval, called temporal validity interval [5], and its freshness degrades with time. To maintain its freshness, a new data value needs to be installed into the database by a corresponding update task before the validity of its old value expires. Otherwise, the system may not be able to detect and respond to environmental changes in a timely fashion.

- *S. Han is with the Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269. E-mail: song@engr.uconn.edu.*
- *K.-Y. Lam is with the Department of Computer Science, City University of Hong Kong. E-mail: cskylam@cityu.edu.hk.*
- *J. Wang is with the Noah's Ark Lab, Huawei Tech. E-mail: roy.wangjiantao@huawei.com.*
- *D. Chen is with the Tongji University. E-mail: dejichen@tongji.edu.cn.*
- *M. Xiong is with Google Inc. E-mail: mxiong@google.com.*
- *K. Ramamritham is with the Department of Computer Science and Engineering, IIT Bombay. E-mail: krithi@iitb.ac.in.*
- *A.K. Mok is with the Department of Computer Science, the University of Texas at Austin, Austin, TX 78712. E-mail: mok@cs.utexas.edu.*

Manuscript received 22 Feb. 2014; revised 9 Oct. 2015; accepted 16 Oct. 2015. Date of publication 29 Oct. 2015; date of current version 2 Feb. 2016. Recommended for acceptance by L. Chen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2015.2496199

In recent years, many efforts have been devoted to designing and analysing update algorithms to maintain the *temporal validity* of real-time data [5], [6], [7], [8], [9], [10]. In general, the simpler algorithms maintain better data freshness and incur lower online scheduling overhead but result in heavier update workloads [5], [6], [7]; sophisticated ones improve the schedulability by reducing the update workload but may sacrifice data freshness [8], [9], [10]. Most of the prior work assume that the update task set in the system is fixed. However, many complex CPSs in the real world, like hybrid and switched systems [11], [12], exhibit multi-modal behavior and each mode is characterized by a set of functionalities that are carried out by different task sets. A change in mode often leads to the modification of task parameters (e.g., task period and execution time) as well as involves the addition of some new tasks and deletion of some existing tasks. We call this type of CPS *dynamic cyber-physical systems (DCPS)*.

A typical example of DCPS is an aircraft control system, involving landing, takeoff and normal cruise modes, where each mode consists of different task sets. To handle the temporal overload when switching among these modes, various mode switch protocols[1] have been proposed to guarantee that no deadlines will be missed during the switch. Most of these works, however, stick to the same scheduling algorithm during the entire execution of the system. They do not consider data freshness and hence do not explicitly

1. Mode switch is also called mode change in the literature.

TABLE 1
Symbols and Definitions

Symbol	Definition
X_i	Real-time data object i $(i = 1, \ldots, m)$
τ_i	Update task updating X_i
$J_{i,j}$	The jth job of τ_i $(j = 0, 1, 2, \ldots)$
$R_{i,j}$	Response time of $J_{i,j}$
C_i	Worst-case execution time of task τ_i
V_i	Validity (interval) length of X_i
$f_{i,j}$	Finishing time of $J_{i,j}$
$r_{i,j}$	Release (Sampling) time of $J_{i,j}$
$d_{i,j}$	Absolute deadline (or deadline) of $J_{i,j}$
P_i	Period of task τ_i in *ML*
D_i	Relative deadline of task τ_i in *ML*
M_i	The ith mode in the system $(i = 0, 1, 2, \ldots)$
$\mathcal{T}_i$	The fixed update task set in mode M_i
Ψ_i	The update algorithm applied on $\mathcal{T}_i$
$\mathcal{T}_k^c$	Changed tasks in switch $M_k \rightarrow M_{k+1}$
$\mathcal{T}_k^u$	Unchanged tasks in switch $M_k \rightarrow M_{k+1}$
$\mathcal{T}_k^-$	Completed tasks in switch $M_k \rightarrow M_{k+1}$
$\mathcal{T}_k^+$	New tasks in switch $M_{k-1} \rightarrow M_k$
t_{MSR}	The issue time of the mode switch request
t_L	The latency requirement of the MSR
t_w	The switch time in the mode switch
$r_{i,j}^k$	Release time of $J_{i,j}$ in mode M_k
$d_{i,j}^k$	Absolute deadline of $J_{i,j}$ in mode M_k
$\Theta_i(a, b)$	Total cumulative CPU demands from higher priority tasks received by τ_i in interval $[a, b)$

address the problem of how to maintain the temporal validity constraints of the real-time data during the mode switch.

In contrast we examine the suitability of using different update algorithms in different modes based on the system workload. What distinguishes our work from previous works is that our goal is to meet the temporal validity constraints not only before and after the mode switch, but also during the mode switch. We aim at achieving a good balance between data freshness and system schedulability. We address two important questions: 1) which update algorithm should we apply in a mode, and 2) when should we perform the switch such that temporal validity can be maintained and data freshness can be maximized during the switch.

To address the first problem, we introduce the *utilization-based scheduling selection (UBSS)* strategy. UBSS selects the update algorithm under which the update task set is schedulable. We prefer a simple one when the system workload is low to provide better data freshness and reduce the online scheduling overhead. When the system load is heavy, we will apply a more sophisticated update algorithm to accommodate more update tasks and meet their deadlines.

We study the second problem in two mode switch scenarios: *clean switch* and *non-clean switch*. In a clean (non-clean) switch, there are no (may be) outstanding update jobs from the old mode at the switch point. An outstanding update job is defined as an update job which has not finished its execution before the switch point. For the clean switch scenario, we introduce two switch algorithms, *search-based switch (SBS)* and *adjustment-based switch (ABS)* to identify proper switch points so that the temporal validity constraints can be satisfied during the mode switch. *SBS* checks the beginning time slot of each idle period (i.e., a set of consecutive idle time slots in the schedule) following the mode switch request (MSR) and determines whether it is a switch point. *ABS* relaxes this restriction on the switch point candidates and adjusts the schedule between the MSR and the current time slot to make it a switch point.

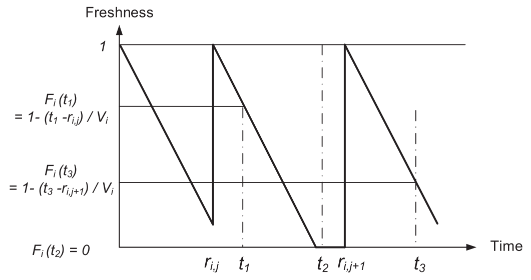


Fig. 1. Freshness of real-time data object X_i.

In the case that a mode switch must be effected immediately after an MSR is received, or *ABS*/*SBS* cannot find a proper switch point, we may have to perform the mode switch in non-clean switch scenarios. To accommodate this need, we propose another switch algorithm named *instant switching (IS)* to schedule both the outstanding jobs from the old mode and update jobs in the new mode, by applying the least-available-laxity-first scheduling policy. Our experimental results demonstrate that *IS* algorithm can offer better flexibility in scheduling, and improve the switch latency significantly.

2 BACKGROUND

In this section, we briefly review the concepts of *temporal validity* and *freshness* of real-time data. For the readers' convenience, we summarize the set of frequently used symbols and their definitions in Table 1.

2.1 Temporal Validity and Data Freshness

Definition 2.1. *A real-time data object (X_i) is temporally valid at time t if, for its latest (say, the* jth*) update job finished before t, the sampling time ($r_{i,j}$) plus the* validity interval *(V_i) of the data object is not less than t,* i.e., $r_{i,j} + V_i \geq t$ *[5].*

Following Definition 2.1, a data value for real-time data object X_i sampled at time t will be valid up to $(t + V_i)$. The freshness of X_i is defined as follows.

Definition 2.2. *The* freshness *of a real-time data object (X_i) at time t, $F_i(t)$, is no smaller than 0 and is 1 minus the ratio of the difference between t and the sampling (or release) time*[2] *of the latest (say, the* jth*) update job of τ_i finished before t to the* validity interval *(V_i) of the data object, i.e., $F_i(t) = max$ $\{1 - \frac{t-r_{i,j}}{V_i}, 0\}$.*

Fig. 1 shows an example of the freshness values of two versions of real-time data object X_i sampled at $r_{i,j}$ and $r_{i,j+1}$. We highlight the freshness values $F_i(t_1)$ and $F_i(t_2)$ for the version sampled at $r_{i,j}$, and $F_i(t_3)$ for the version sampled at $r_{i,j+1}$, respectively. At time $r_{i,j}$, the freshness of the data

2. In this paper, the sampling time of an update job is when it is generated at the data source and the release time is when it is ready for installation in RTDB. To simplify the discussion, we assume that the delay between the sampling time and release time is negligible.

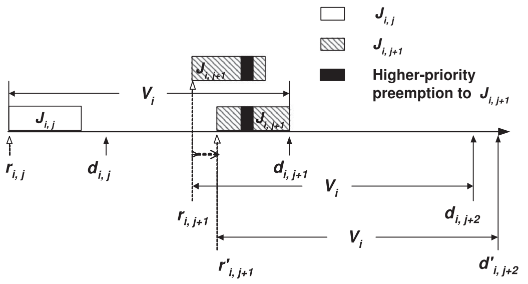


Fig. 2. Illustration of *DS-FP* scheduling: $r_{i,j+1}$ can be shifted to $r^{'}_{i,j+1}$ without violating the validity constraint.

object is 1 and decreases linearly to 0 until time t_2. From t_2, its freshness keeps on as 0 until the next sampling time $r_{i,j+1}$.

2.2 Algorithms for Maintaining Temporal Validity

In this section, we briefly review four efficient update algorithms for maintaining temporal validity of real-time data: *More-Less* (*ML*) [6], [13], *Half-Half* (*HH*) [5], [14], *DS-FP* [15] and *DS-LALF* [16]. The first three algorithms are synchronous fixed-priority scheduling where the task set is governed by fix-priority scheduling algorithms and the first jobs of all tasks are released at the same time. By contrast, *DS-LALF* is asynchronous dynamic scheduling where the task set is executed under a dynamic scheduling algorithm and the release times of the tasks' first jobs could vary.

More-Less (ML). In *ML*, a synchronous and periodic task model is assumed and the first job's response time is the worst-case response time. The scheduling of the jobs has to satisfy the following three constraints for tasks τ_i ($\forall i,\ 1 \leq i \leq m$) [6]:

- *Validity constraint*: the sum of period P_i and relative deadline D_i of τ_i is no larger than V_i, i.e.,

$$P_i + D_i \leq V_i. \tag{1}$$

  The relative deadline of task τ_i is the time period between its release time and absolute deadline.
- *Deadline constraint*: the period of an update task is assigned to be no smaller than half of the validity length of its updated object, while the relative deadline must be greater than or equal to C_i, the worst-case execution time of τ_i, i.e., $C_i \leq D_i \leq P_i$.
- *Schedulability constraint*: for a given set of update tasks, the *Deadline Monotonic* scheduling algorithm is used to schedule the tasks. Consequently, $\sum_{j=1}^{i}(\lceil \frac{D_i}{P_j} \rceil \cdot C_j) \leq D_i \ (1 \leq i \leq m)$.

ML assigns priorities to tasks based on *Shortest Validity First* (SVF), and it assigns deadlines and periods to τ_i as follows:

$$D_i = f_{i,0} - r_{i,0}, \tag{2}$$

$$P_i = V_i - D_i, \tag{3}$$

where $f_{i,0}$ and $r_{i,0}$ are finishing and release times of the first job of τ_i, respectively.

Half-Half (HH). In *HH*, period P_i and relative deadline D_i of task τ_i are each set to be half of the validity interval of the corresponding data object [5], [14].

DS-FP. *ML* is pessimistic on the deadline and period assignments as the relative deadline of an update task is set to be the worst-case response time of the task. To increase the separation of two consecutive jobs from the same task (and thus reduce the update workload), *DS-FP* adaptively derives the relative deadline and release time of a job using the total preemption from higher-priority jobs [15]. Given release time $r_{i,j}$ of job $J_{i,j}$ and deadline $d_{i,j+1}$ of job $J_{i,j+1}$,

$$d_{i,j+1} = r_{i,j} + V_i \tag{4}$$

guarantees that the validity constraint can be satisfied, as depicted in Fig. 2. Correspondingly, Eq. (5) follows directly from Eq. (4):

$$(r_{i,j+1} - r_{i,j}) + (d_{i,j+1} - r_{i,j+1}) = V_i. \tag{5}$$

If $r_{i,j+1}$ can be shifted onward to $r^{'}_{i,j+1}$ along the time line in Fig. 2, it does not violate Eq. (5). After the shift, temporal validity can still be guaranteed as long as $J_{i,j+1}$ is completed by its deadline $d_{i,j+1}$. The idea of *DS-FP* is to defer the release time, $r_{i,j+1}$, of $J_{i,j}$'s next job as late as possible while still guaranteeing the validity constraint.

Definition 2.3. *Let $\Theta_i(a,b)$ denote the total* cumulative processor demands *made by all jobs of higher-priority task τ_j for $\forall j\ (1 \leq j \leq i-1)$ during the time interval $[a,b)$ from a schedule $\mathcal{S}$ produced by a fixed-priority scheduling algorithm. Then,*

$$\Theta_i(a,b) = \sum_{j=1}^{i-1} \theta_j(a,b),$$

where $\theta_j(a,b)$ is the total processor demands made by all jobs of single task τ_j during $[a,b)$.

According to the fixed-priority scheduling theory [17], $r_{i,j+1}$ in *DS-FP* can be derived backwards from its deadline $d_{i,j+1}$ as follows:

$$r_{i,j+1} = d_{i,j+1} - R_{i,j+1}; \tag{6}$$

$$R_{i,j+1} = \Theta_i(r_{i,j+1}, d_{i,j+1}) + C_i, \tag{7}$$

where $R_{i,j+1}$ denotes the response time of $J_{i,j+1}$ deriving backwards from its deadline $d_{i,j+1}$.

Similar to *ML*, *DS-FP* assigns priorities to tasks according to *SVF*. Readers are referred to [8] for details of *DS-FP* algorithm and its schedulability analysis.

DS-LALF. *DS-LALF* [16] is an extension of *DS-FP*. Therefore, the separation between two consecutive jobs from the same update task is also determined based on the total preemption from higher priority jobs at run-time. However, in contrast to *DS-FP* which is a fixed-priority scheduling for synchronous update tasks, *DS-LALF* is a dynamic scheduling for asynchronous update tasks. *DS-LALF* decides the priority of an update job using its actual laxity, i.e., the number of available idle time slots in the time interval between the deadlines of the two consecutive jobs from the same update task, i.e., $[d_{i,k-1}, d_{i,k}]$, after its previous update job $J_{i,k-1}$ has been completed. The details of *DS-LALF* will be discussed in Section 5.

Performance summary. Although *DS-FP* and *DS-LALF* have been shown to give a better performance in reducing the total update workload in maintaining the temporal validity of real-time data [16], [18], their online scheduling overhead is usually higher compared with *ML* and *HH*. This could be a performance problem for mode switch in *DCPS* as the new schedule normally needs to be computed promptly and the switch has to be completed within a short period of time. Otherwise, the overall system performance may be affected and the data validity may become more difficult to be guaranteed. Although the schedulability of *DS-FP* and *DS-LALF* may be better than that of *ML* and *HH*, the data freshness obtained from them may be *lower* as the update periods are in general longer.

2.3 Mode Switch Protocols

In the literature, many protocols have been developed to cope with temporal overload during the mode switch when the set of tasks to be executed may include both old- and new-mode tasks. These mode switch protocols in general can be classified into synchronous and asynchronous protocols with regard to the way old- and new-mode tasks are combined during the mode switch [19]. Real [19] proposes two synchronous protocols, one with periodicity and the other without. They assume that the old-mode tasks may be completed even if they have outstanding executions when an MSR is issued.

Sha et al. [20] introduces an analysis approach for mode switches on single processor system under the rate-monotonic scheduling. The protocol is based on dynamic processor utilization and the rules of the priority ceiling protocol. Tindell et al. [21] extend [20] to the deadline-monotonic scheduling with the periodicity maintained. Pedro and Burns [22] further generalize [21] by introducing possible offsets for all the new-mode tasks, and thus relaxes the periodicity requirement.

All the analysis methods in [20], [21], [22], [23] are limited to strictly periodic task activation. Henia and Ernst [24] eliminate this restriction by allowing complex task activation patterns including periodic with jitter, periodic with burst and sporadic event models. Stoimenov et al. [25] further improve [24] by supporting any event stream model to handle both the earliest deadline first and fixed-priority scheduling of tasks.

Different from prior works, our work focuses on maintaining temporal validity of update tasks during mode switch by applying different update algorithms in different modes based on the system workload.

3 Util-Based Scheduling Selection

In this section, we introduce the *utilization-based scheduling selection* strategy. In the following discussion, we concentrate on the scheduling of update tasks. If there are non-update tasks, e.g., generated from applications, they will be assigned to lower priorities for scheduling. This is based on the assumption that if sensor values have not been updated, there is not as much incentive to re-compute the application tasks. The co-scheduling of the update and application tasks under different task models needs an integrated solution to satisfy the timeliness requirements. It is out of the scope of this paper and will be considered as an important future work.

3.1 Models and Definitions

We model the operational dynamics in a *DCPS* as a series of modes, $M_0, M_1, M_2, \ldots,$ and each mode M_k contains a fixed update task set $\mathcal{T}_k = \{\tau_i\}_{i=1}^{m}$ with known worst-case execution time C_i and validity interval V_i for each update task τ_i $(1 \leq i \leq m)$. Note that to simplify the discussion, we associate the validity interval of a data object to the update task which is responsible for maintaining its validity. Since the jobs to be performed in a *DCPS* are usually pre-defined and thus their worst-case execution times can be estimated with high accuracy. In the model, an update algorithm Ψ_k is applied to update task set $\mathcal{T}_k$ in mode M_k. Following the assumptions in the prior works, we assume that MSR is a sporadic event and it cannot occur during mode transitions. There are in total four types of tasks in our model:

Completed tasks are tasks that are active in the old-mode but do not appear in the new mode. They are allowed to run to completion under the old update algorithm after the MSR with no new job to be released. However, they *cannot* be aborted for the purpose of maintaining the temporal validity of the data.

New tasks are tasks that only appear in the new mode. They are released synchronously at the selected switch point.

Unchanged tasks are tasks that are persistent through the mode switch. They are executed and released after the MSR under the old update algorithm until the new algorithm takes the control.

Changed tasks are tasks that appear in both modes but with modified parameters like C_i and V_i in the new mode. The temporal validity for these tasks during the switch must also be maintained.

We assume that $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$ are the two task sets before and after the mode switch from M_k to M_{k+1}, and they are schedulable under update algorithms Ψ_k and Ψ_{k+1} respectively. Let $\mathcal{T}_k^c$ and $\mathcal{T}_k^u$ denote the changed and unchanged task sets in mode M_k during the switch respectively. According to the definitions in Section 3.1, $\mathcal{T}_k^c$ and $\mathcal{T}_k^u$ are the only tasks that appear in both modes and we have $\mathcal{T}_k^c \bigcup \mathcal{T}_k^u = \mathcal{T}_k \bigcap \mathcal{T}_{k+1}$. The completed task set $\mathcal{T}_k^-$ and the new task set $\mathcal{T}_{k+1}^+$ contain the tasks that only appear in mode M_k and mode M_{k+1}, respectively. We thus have $\mathcal{T}_k^- = \mathcal{T}_k - \mathcal{T}_k^c - \mathcal{T}_k^u$ and $\mathcal{T}_{k+1}^+ = \mathcal{T}_{k+1} - \mathcal{T}_k^c - \mathcal{T}_k^u$.

We denote by t_{MSR} the time when the MSR is issued and t_L the MSR latency requirement. That is, the mode switch must be conducted within the time period $[t_{MSR}, t_{MSR} + t_L]$. In this paper, we focus on the following scenario: A *DCPS* in mode M_k is initially scheduled under Ψ_k. At time t_{MSR}, the system is notified by a change of the task set and is requested to finish a switch before time $t_{MSR} + t_L$. After t_{MSR}, the tasks in $\mathcal{T}_k^-$ are run to completion and the tasks in $\mathcal{T}_k^c \bigcup \mathcal{T}_k^u$ are executed and released as normal under Ψ_k until a certain time point t_w $(t_{MSR} \leq t_w \leq t_{MSR} + t_L)$ when the task set to be executed is changed to $\mathcal{T}_{k+1}$ and a new update algorithm Ψ_{k+1} is in control. The problem is how to

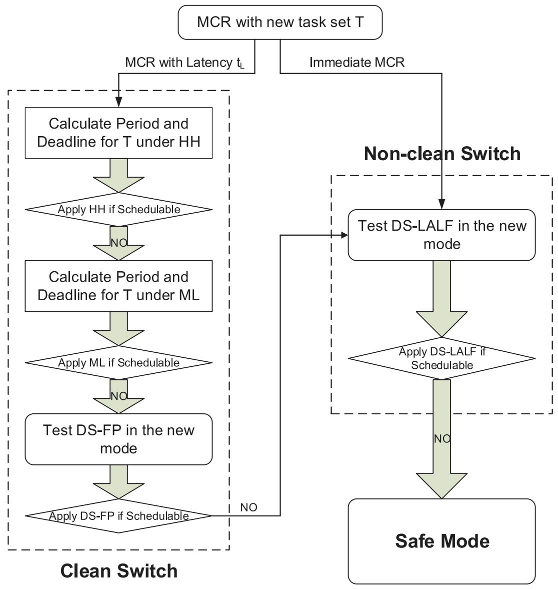

MCR with new task set T
MCR with Latency t_L
Immediate MCR
Calculate Period and Deadline for T under HH
Apply HH if Schedulable
NO
Calculate Period and Deadline for T under ML
Apply ML if Schedulable
NO
Test DS-FP in the new mode
Apply DS-FP if Schedulable
NO
Clean Switch
Non-clean Switch
Test DS-LALF in the new mode
Apply DS-LALF if Schedulable
NO
Safe Mode

Fig. 3. Utilization-based scheduling selection.

choose Ψ_k and Ψ_{k+1} to achieve the tradeoff between data freshness and schedulability, and how to find the proper switch point that can preserve the temporal validity of the tasks in $\mathcal{T}_k^c \bigcup \mathcal{T}_k^u$ during the transition.

3.2 Selection of Update Algorithms

To maintain the temporal validity of data in *DCPS*, the first problem to address is how to select the proper update algorithm for each mode so that the corresponding task set is schedulable. As mentioned in Section 1, in the clean switch scenario, our strategy is to apply the periodic update algorithm when the update workload is low as long as the task set is schedulable. This will provide higher data freshness with lower scheduling overhead. We only have to switch to a more sophisticated algorithm when the schedulability bounds for periodic update algorithms are exceeded. In this way, we maintain the temporal validity by degrading the data freshness.

In this paper, we consider three candidate update algorithms: *HH*, *ML* and *DS-FP* for the synchronous scheduling switch. *DS-FP* has the best schedulability among all the three candidates but the worst data freshness. We choose the update algorithm based on the imposed workload from the update tasks and the selection process is summarized on the left side of Fig. 3. If the system is notified with an MSR, it will calculate the period and deadline for each update task in the new task set under *HH* and *ML*, respectively [6], [14]. Theorem 5 in [17] is used to check the total utilization of the periodic tasks against the schedulability bound. If the task set is not schedulable under *HH*, we will try *ML* instead. We will only choose *DS-FP* when both *HH* and *ML* do not work, i.e., they cannot schedule the set of update tasks. [18] gives the schedulability test for *DS-FP*. Note that any well-defined *DCPS* should have a finite number of modes in which the associated task set is fixed. Thus the update algorithm selection process can be performed offline, and the overhead of the corresponding schedulability tests on the update algorithms will not cause deadline miss in the runtime.

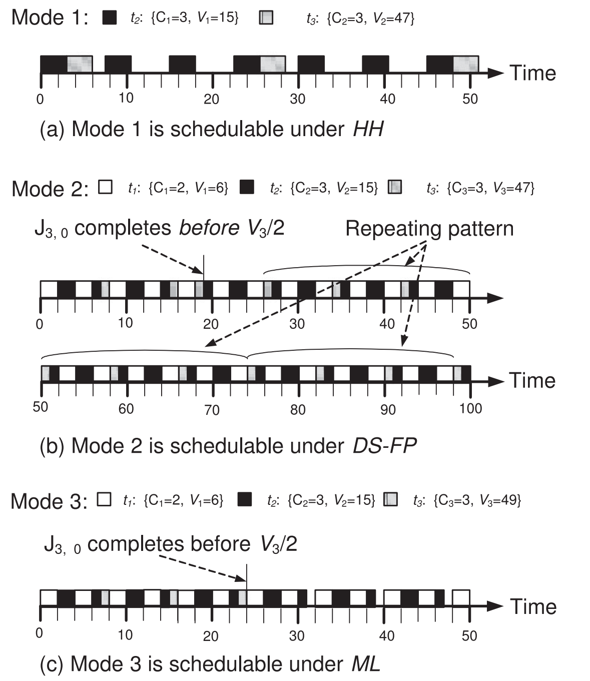


Fig. 4. An example of update algorithm selection.

Example 3.1. Consider three consecutive modes, M_1, M_2 and M_3 as shown in Fig. 4. In M_1, there are two tasks $\tau_2 = (3, 15)$ and $\tau_3 = (3, 47)$. (i.e., $\tau_i = (x, y)$ with $x =$ the worst-case execution time and $y =$ the validity interval.) In M_2, a new task $\tau_1 = (2, 6)$ is added into the system. In M_3, task τ_3's validity interval is incremented by 2 to 49 with other parameters in the task set unchanged. In M_1, the processor utilization for the task set under *HH* is 0.528. This is below 0.828 which is the bound evaluated by Theorem 5 in [17]. According to our strategy, *HH* will be selected for scheduling in M_1. However, in M_2, under *ML*, the first job of τ_3, $J_{3,0}$, completes at time 24, which is greater than $\frac{V_3}{2}$(that is 23.5). Thus, this new task set is not schedulable under either *ML* or *HH*. On the other hand, the same task set is schedulable under *DS-FP*, because the schedule pattern between times 26 and 50 can be found by *DS-FP* and it repeats forever. In M_3, with τ_3's validity interval increased to 49, the new task set is not schedulable under *HH*, but schedulable under *ML* because the deadline constraint is satisfied at the critical instance, i.e., time 0. The critical instance is the time instant that leads to the worst-case response time for all the jobs.

If immediate MSR is requested, or *DS-FP* cannot schedule the set of update tasks as well, as shown on the right side of Fig. 3, we apply *DS-LALF* (to be elaborated in Section 5) to schedule both outstanding jobs in the old mode and new jobs released in the new mode for non-clean switch. If the task set is not schedulable under *DS-LALF*, the system is considered overloaded and will enter the safe mode in which the system runs the most basic functions. How to define the safe mode and its associated update task set is up to the system designer. In common practice, the safe mode can be consisted of a set of hard real-time periodic tasks that must be finished without deadline miss, and has reduced functions compared with other normal modes.

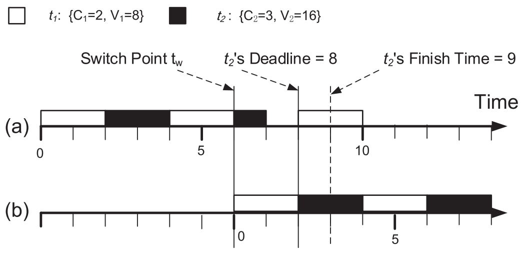


Fig. 5. Non-clean switch from *HH* to *HH* (Failed).

Note that although we currently only consider four candidate update algorithms, UBSS can be easily extended to support other update algorithms, as long as their exact schedulability test can be developed. UBSS will sort these algorithms in the increasing order of their utilization bounds, and choose the simplest algorithm which can schedule the given update task set.

4 Synchronous Switch

Even both the old and new task sets are schedulable under the selected update algorithms, it is not guaranteed that the temporal validity of the real-time data will be maintained. This is because the temporal validity of the tasks persistent through the switch could be violated during the switch. To satisfy all these temporal validity constraints, the switch point should be chosen carefully. In this section we first introduce two different switch scenarios, the clean switch and non-clean switch. For the clean switch scenario, in this section, we propose two synchronous switch algorithms for searching the proper switch points. They are the *search-based switch* and *adjustment-based switch*.

4.1 Clean Switch versus Non-Clean Switch

During the mode switch from M_k to M_{k+1}, since all the tasks are schedulable by their respective update algorithms, all the tasks in $\mathcal{T}_k^- \bigcup \mathcal{T}_{k+1}^+$ are schedulable and thus their temporal validities are maintained. However, if a task in $\mathcal{T}_k^-$ has an outstanding job at the switch point t_w, the scheduling of the job will be undetermined. If we simply drop it, its temporal validity might be violated; If we let it run to finish, then how it should be scheduled after t_w is unspecified.

For tasks in $\mathcal{T}_k^c \bigcup \mathcal{T}_k^u$, although they are schedulable until t_w, Ψ_{k+1} may be affected by their last jobs before t_w and some update algorithms are preconditioned upon the fixed starting times of the first jobs, e.g., all the tasks start at time 0. Similar problem arises if a task in $\mathcal{T}_k^c \bigcup \mathcal{T}_k^u$ has an outstanding job at t_w. The question is whether to consider the outstanding job as the first job in Ψ_{k+1}. If so then it no longer has full execution requirement. If not then it may not meet its deadline defined in Ψ_k depending on how Ψ_{k+1} schedules it. Furthermore, it also interferes Ψ_{k+1} just as those tasks in $\mathcal{T}_k^-$ could do. In either case Ψ_{k+1} do not have a clean start.

Example 4.1. Fig. 5 depicts a non-clean switch scenario where a same set of periodic tasks are scheduled in both modes M_k and M_{k+1} under *HH* and the MCR is at time 6, i.e., $\mathcal{T}_k = \mathcal{T}_{k+1} = \{\tau_1 = (2, 8), \tau_2 = (3, 16)\}$, and $t_w = 6$. In the figure, (a) is the schedule of $\mathcal{T}_k$ under *HH* in the time period [0, 10] if no MCR is requested; (b) is the schedule of $\mathcal{T}_{k+1}$ under *HH* which treats $t_w = 6$ as its time 0, i.e., all the tasks in $\mathcal{T}_{k+1}$ are synchronously scheduled onwards. In this example, τ_2's outstanding job in mode M_k runs to finish but preempted by τ_1's first job in mode M_{k+1} after $t_w = 6$. So it misses its deadline, time point 8, defined in *HH*. Interestingly all jobs after $t_w = 6$ meet deadlines in mode M_{k+1} in spite of this extra execution.

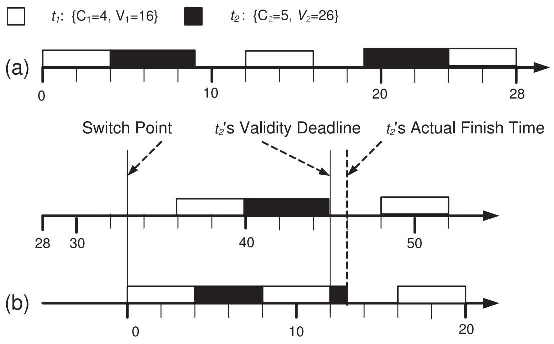


Fig. 6. Clean switch from *DS-FP* to *HH*.

4.2 Search-Based Switch (SBS)

This section studies the clean switch scenarios where:

- The last jobs of all tasks in $\mathcal{T}_k$ are completed before t_w, i.e, no outstanding execution at t_w.
- Ψ_{k+1} schedules $\mathcal{T}_{k+1}$ independent of the prior schedule as if t_w is its time 0.

Consider a mode switch from M_k to M_{k+1}. For a task in $\mathcal{T}_k^c \bigcup \mathcal{T}_k^u$ such that its last job before t_w and the first job after t_w meet the deadlines in their respective Ψ, we cannot say that its real-time requirement is met because the temporal validity has not been taken into account during the switch.

Example 4.2. Fig. 6 depicts a clean switch scenario where a same set of periodic tasks are scheduled in both modes M_k and M_{k+1}, i.e., $\mathcal{T}_k = \mathcal{T}_{k+1} = \{\tau_1 = (4, 16), \tau_2 = (5, 26)\}$. We switch this task set from $\Psi_k = DS - FP$ to $\Psi_{k+1} = HH$ at $t_w = 33$. Modeled in *HH*, in $\mathcal{T}_{k+1}$, $P_1 = D_1 = 8$ and $P_2 = D_2 = 13$. In the figure, (a) is the schedule of T_k under Ψ_k in the time period [0, 52] if no MCR is requested at time 33; (b) is the schedule of T_{k+1} under Ψ_{k+1} which treats $t_w = 33$ as its time 0. For τ_1, its last job before $t_w = 33$ starts at time 24 and its first job after $t_w = 33$ finishes at time 37. The distance is 13, which is less than $V_1 = 16$. For τ_2, however, its last job before t_w starts at time 19 and its first job after t_w finishes at time 46. The distance is 27, which is larger than $V_2 = 26$. So τ_2's validity constraint is violated during the switch.

Example 4.2 exposes hidden real-time requirements that could be missed. To address this problem, we define a successful switch to be the one that for all tasks $\tau_i \in \mathcal{T}_k^c \bigcup \mathcal{T}_k^u$, its first job after t_w finishes within the validity interval from the start time of its last job before t_w. Our problem now becomes how to find a time point at which a successful switch is possible.

An idle period (t_1, t_2) of a schedule begins when the last outstanding execution of any task is finished right before t_1 and there is no new job request until t_2. The process does not execute any job within (t_1, t_2). This definition includes trivial periods in which $t_1 = t_2$. In the clean switch scenario, obviously t_w falls within some idle period and we have the following lemma.

Lemma 4.1. *Any successful switch point falls in an idle period and any time point from the beginning of the idle period to this switch point is also a successful switch point.*

Proof. Suppose $t_w \in (t_1, t_2)$ is a successful switch point. If we switch from any time point in (t_1, t_w), the release times of the last scheduled jobs in Ψ_k will not be changed. The new switch will shift Ψ_{k+1}'s schedule closer to time t_1 and any first job in $\mathcal{T}_k^c \bigcup \mathcal{T}_k^u$ will have an earlier finish time, and hence the temporal validity constraint will be satisfied. □

Note that in Fig. 6b, time 33 is not a successful switch point. However, times from 28 to 32 are all successful switch points. This gives us the *search-based switch* algorithm (Algorithm 1).

Algorithm 1. Search-based Switch (SBS) Algorithm

```
Input: $\mathcal{T}_k, \mathcal{T}_{k+1}, \Psi_k, \Psi_{k+1}$, the search start time $t_0$ and $t_L$.
Output: $t_w$.
1:  for $t = t_0$ to $t_0 + t_L$ do
2:    if $t == t_1$ then
3:      //$t_1$ is the begin point of an idle period
4:      $f = true$;
5:      //Whether $t$ is a possible candidate for $t_w$
6:      for each $\tau_i \in \mathcal{T}_k \bigcap \mathcal{T}_{k+1}$ do
7:        $t_s$ = release time of $\tau_i$ 's last job in $\Psi_k$;
8:        $l$ = time to finish $\tau_i$'s first job in $\Psi_{k+1}$;
9:        if $t - t_s + l > V_i$ then
10:         $f = false$;
11:       end if
12:     end for
13:     if $f == true$ then
14:       return $t$;
15:     end if
16:   end if
17: end for
18: return no $t_w$ exists;
```

As all schedulable schedules have a repeating pattern in discrete time system [18], if t_L, the latency requirement of the MSR is not explicitly given, Algorithm 1 runs at most the length of the shortest pattern of the schedule, $\mathcal{P}_l$, and it will eventually terminate. It will report failure if there is no successful switch point. Otherwise, the algorithm will always return it.

Note that Algorithm 1 does not require that the parameters (e.g., the validity interval) of a task are the same before and after the switch. It means that the algorithm can apply to the tasks both in $\mathcal{T}_k^c$ and $\mathcal{T}_k^u$.

4.3 Adjustment-Based Switch (ABS)

Although *SBS* is straight forward and easy to be implemented, it restricts the switch point candidates only at the beginning of the idle periods in $[t_{MSR}, t_{MSR} + t_L]$. This constraint limits the number of possible candidates and may reduce the promptness of the mode switch.

We want to remove this restriction and take every time point in $[t_{MSR}, t_{MSR} + t_L]$ as the candidates for switch to improve the promptness of the mode switch. However, this extension will put us in the non-clean switch scenario as there could be outstanding jobs at that time point. In this section, we propose the *adjustment-based switch* to address this problem. *ABS* converts the non-clean switch scenario to clean switch scenario through schedule adjustment. By schedule adjustment, we mean changing the release times and deadlines of the jobs. The basic idea of *ABS* is that, at a certain time point t ($t_{MSR} \le t \le t_{MSR} + t_L$), we push all the outstanding jobs back to t. That is, all the outstanding jobs in $\mathcal{T}_k$ must be finished by t in the adjusted schedule. Then the schedule in $[t_{MSR}, t]$ will be adjusted backwards from time t so that the adjusted schedule can guarantee the validity constraints of all the update tasks. Note that if task τ_h is the highest priority task in $\mathcal{T}_k$ whose schedule needs to be adjusted, the schedule of all the lower-priority tasks $\tau_i (h < i \le m)$ in $\mathcal{T}_k$ also needs to be adjusted. After the schedule adjustment, *ABS* releases all the tasks in $\mathcal{T}_{k+1}$ at time t and checks whether for each task $\tau_i \in \mathcal{T}_k^c \bigcup \mathcal{T}_k^u$, it can maintain the temporal validity during the switch. If not, *ABS* will examine the next switch candidate.

Algorithm 2. Adjustment-based Switch (ABS) Algorithm

```
Input: $\mathcal{T}_k, \mathcal{T}_{k+1}, \Psi_k, \Psi_{k+1}, t_0$ and $t_L$.
Output: $t_w$.
1:  for $t = t_0$ to $t_0 + t_L$ do
2:    //$\sum_i \Gamma_i(t)$ is accumulated outstanding execution at $t$
3:    if $I(t_0, t) < \sum_i \Gamma_i(t)$ then
4:      continue;
5:    else
6:      //Adjust the schedule of $\mathcal{T}_k$ in $[t_0, t]$
7:      $f$ = ScheduleAdjustment $(\mathcal{T}_k, t_0, t)$;
8:      if $f$ = fail then
9:        continue;
10:     else
11:     for each $\tau_i \in \mathcal{T}_k \bigcap \mathcal{T}_{k+1}$ do
12:       $t_s$ = adjusted request time of $\tau_i$ 's last job in $\Psi_k$;
13:       $l$ = time to finish $\tau_i$s first job in $\Psi_{k+1}$;
14:       if $t - t_s + l > V_i$ then
15:         //The temporal validity is violated.
16:         $f$ = fail;
17:       end if
18:     end for
19:     if $f$ = success then
20:       return $t$;
21:       end if
22:     end if
23:   end if
24: end for
25: return no $t_w$ exists;
```

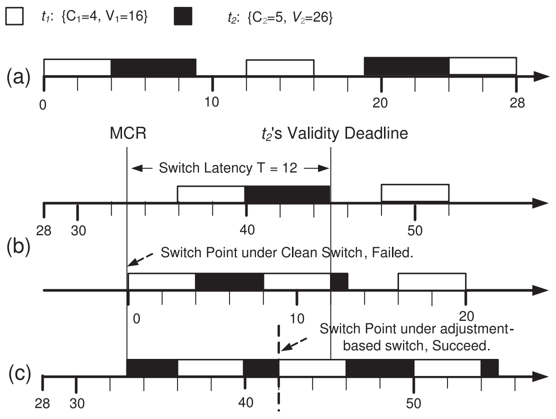
The details of *ABS* is presented in Algorithm 2. We denote by $I(a, b)$ the total number of idle slots between time interval $[a, b]$, and $\Gamma_i(t)$ the outstanding job of τ_i at time t. In Algorithm 2, line 3 checks that at each candidate time t, whether there are enough idle slots in $[t_0, t]$ to accommodate all the outstanding jobs. Algorithm 2 sequentially checks each time slot in $[t_0, t_0 + t_L]$ and returns the earliest proper switch point or reports failure when time $t_0 + t_L$ is reached. The function *ScheduleAdjustment* (T, t_0, t) in line 7 tries to push back all the outstanding jobs in task set T at time t and adjust the schedule in $[t_0, t]$ to satisfy the temporal validity constraints. If the adjustment is successful, lines 13-18 further verify whether the temporal validity of each adjusted job is also maintained during the switch. The algorithm returns success if time t is a valid switch point. Otherwise, each remaining switch candidate will be examined until the algorithm returns failure in line 25.

Algorithm 3. ScheduleAdjustment (T, t_0, t)

Input: Task set T and adjustment period $[t_0, t]$.
Output: Adjusted schedule S in $[t_0, t]$ and $\forall \tau_i$, the adjusted release time of its last job before t, r'_{i,k_i}.

```
1:  $h = \min_i\{i|\tau_i \in T$ and $\tau_i$ has outstanding execution at $t.\}$
2:  $\forall i < h, k_i = k_i - 1$; //No adjustment for $i < h$
3:
4:  //$J_{i,k_i}$ is the last job of $\tau_i$ in $[t_0, t]$
5:  for $i = h$ to $m$ do
6:    $d'_{i,k_i} = t$; //$d'_{i,k_i}$ is adjusted from $d_{i,k_i}$.
7:    $j = k_i$;
8:    $t_s = t$; //Schedule in $[t_s, t]$ will be adjusted.
9:    while $(j > 0)$ do
10:     if $(d'_{i,j} - r_{i,j} < \Theta_i(r_{i,j}, d'_{i,j}) + C_i)$ then
11:       //$J_{i,j}$'s response time $> d'_{i,j} - r_{i,j}$
12:       $r'_{i,j} = d'_{i,j} - \Theta_i(r'_{i,j}, d'_{i,j}) - C_i$;
13:       //cannot adjust the schedule before $t_0$
14:       if $(((j < k_i) \wedge (d'_{i,j+1} - r'_{i,j} > V_i)) \vee (r'_{i,j} < t_0))$
          then
15:         return fail;
16:       end if
17:       if $((r'_{i,j} < d_{i,j-1}))$ then
18:         $d'_{i,j-1} = r'_{i,j}$;
19:       else
20:         $d'_{i,j-1} = d_{i,j-1}$;
21:       end if
22:       $j = j - 1$;
23:     else
24:       //No adjustment for this job
25:       if $(d'_{i,j} - \Theta(t_0, d'_{i,j}) - C_i < t_0)$ then
26:         return fail; //cannot adjust the schedule before $t_0$
27:       else
28:     if $(t_s \geq d'_{i,j})$ then
29:       $t_s = d'_{i,j}$;
30:       break;
31:     else
32:       $d'_{i,j-1} = d_{i,j-1}$;
33:       $j = j - 1$;
34:     end if
35:     if $((j = 0) \wedge (d'_{i,j} \neq d_{i,j}))$ then
36:       return fail;
37:       end if
38:     end if
39:   end if
40:   end while
41: end for
42: return adjusted $S$ in $[t_0, t]$ and $\forall i, r'_{i,k_i}$;
```

Algorithm 3 summarizes the details of the schedule adjustment. Line 1 identifies τ_h, the task with the highest priority in $\mathcal{T}_k$ which has an outstanding job at time t. For each task τ_i $(h \leq i \leq m)$, line 6 assigns t as the deadline of its last job in $[t_0, t]$ if it has outstanding execution. Algorithm 3 adjusts the release time and deadline for each job of these tasks backward sequentially until the condition in line 26 is satisfied where no further adjustment is needed. With the adjusted deadline, line 10 checks whether the corresponding job can be scheduled without exceeding the original release time. Line 12 adjusts the job's release time if necessary. Line 14 verifies two important conditions: 1) whether the job's release time is pushed back before t_0 and, 2) whether the validity constraints are still maintained after the adjustment. Failure will be reported if either of the conditions is not met. Similar checking is also performed in line 23 even when no adjustment is conducted. Lines 16-20 adjust the deadline of the previous job for further processing. If a successful adjustment cannot be achieved after all jobs are tested, line 34 will report failure. Otherwise, the successfully adjusted schedule will be returned.



t_1: {C_1=4, V_1=16}
t_2: {C_2=5, V_2=26}
(a)
0
10
20
28
MCR
t_2's Validity Deadline
Switch Latency T = 12
28
30
40
50
(b)
Switch Point under Clean Switch, Failed.
0
10
20
Switch Point under adjustment-based switch, Succeed.
(c)
28
30
40
50

Fig. 7. A successful adjustment-based switch.

Example 4.3. Following the same task set as in Example 4.2, Fig. 7 depicts a scenario where *SBS* does not work while *ABS* is successful. The MSR is issued at time 33 with a latency requirement of 12. The only switch candidate under *SBS* is time 33 but it does not satisfy the validity constraint for τ_2 during the switch. However, if *ABS* is applied at time 42, the outstanding execution of τ_2 is 3 and it can be adjusted to $[33, 36]$ for execution. At time 42 in the adjusted schedule, τ_2's last job before time 42 is 33 and its first job after time 42 is 55. Since the distance (22) is smaller than V_2, the validity constraint during the switch is satisfied.

4.4 Properties of Switch between *ML* and *DS-FP*

In this section, we look at the switch specifically between *ML* and *DS-FP*. *HH* can be taken as a special case of *ML* with $P_i = D_i = \frac{V_i}{2}$. Theorems 4.1 and 4.2 present two interesting properties of the switch between them.

Theorem 4.1. *For $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$, if $\Psi_k = \Psi_{k+1} = ML$, then any idle point is a successful switch point.*

Proof. We prove that the validity constraint of any task τ in $\mathcal{T}_k^c \bigcup \mathcal{T}_k^u$ is met during the switch. Let $\tau = (C, P)$ and t_w be any idle time considered for switch. Let t be the release time of τ's last job before t_w. Since t_w is an idle time, we have $t_w - t \leq P$. From t_w, the first job of τ will be finished within D. So the distance from the release time of the last job under Ψ_k to the finish time of the first job under Ψ_{k+1} is no more than $(t_w - t) + D \leq P + D = V$. □

Note that Theorem 4.1 holds even if $\mathcal{T}_k \neq \mathcal{T}_{k+1}$. It also holds if the execution times of the tasks in $\mathcal{T}^c_k \bigcup \mathcal{T}^u_k$ change during the switch as long as *ML* schedules both $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$. If the validity intervals of the tasks also change, we have to go back to Algorithm 1 for searching a successful switch point to maintain the strict temporal validity.

Theorem 4.2. *For $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$, if $\mathcal{T}_{k+1} \subseteq \mathcal{T}_k$, $\Psi_k = ML$ and $\Psi_{k+1} = DS\text{-}FP$, any idle point is a successful switch point.*

Proof. We prove that the validity constraint of any task τ in $\mathcal{T}_{k+1}$ is met during the switch. Let t_w be any idle time considered for the switch. Let t be the release time of τ's last job before t_w. Since t_w is an idle time, we have $t_w - t \leq P$. Since $\mathcal{T}_k$ is schedulable under *ML*, its subset $\mathcal{T}_{k+1}$ is also schedulable. Therefore, the worst-case execution time of τ under *ML* is not larger than the relative deadline D. Furthermore, according to Theorem 3.1 in [8], the worst-case execution time of τ under *DS-FP* is not larger than D. So the first job of τ after t_w finishes within D and the distance from the release time of the last job under Ψ_k to the finish time of the first job under Ψ_{k+1} is no more than $(t_w - t) + D \leq P + D \leq V$. □

Note that Theorem 4.2 may not hold if $\mathcal{T}_{k+1} \not\subseteq \mathcal{T}_k$. Even if $\mathcal{T}_{k+1} \subseteq \mathcal{T}_k$, we cannot extend the result to cases where the execution times or validity intervals change. For $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$, if $\Psi_k = DS - FP$ and $\Psi_{k+1} = ML$, an idle time point may or may not be a clean switch point. This is true even if $\mathcal{T}_k = \mathcal{T}_{k+1}$, which is already demonstrated in Example 4.2.

Remark 1. *ABS* and *SBS* are two switch algorithms that we developed for performing efficient online switch. *SBS* fits better in the applications that the update workload is light, so that many idle periods are available in the schedule to find the proper switch point. If the update workload in the system is heavy, it is better to apply *ABS* to create more switch point candidates, so that we can find the proper switch point more promptly.

5 Asynchronous Switch

Section 4 presents two synchronous switch algorithms, *SBS* and *ABS*, for finding proper switch points to either find or create clean switch scenarios, respectively. However, in many practical *DCPS*, instead of allowing a long transition period to find the switch points and create clean switch scenarios, the mode switch may be required to be executed immediately or with a stringent MSR latency requirement, and schedule adjustment may not be allowed because many jobs may have already been finished or partially executed. In these cases, mode switch has to be performed in non-clean switch scenarios. Synchronous switch algorithms, like *SBS* and *ABS*, do not fit well for non-clean switch scenarios for the following two reasons:

- It is difficult to decide how to schedule the outstanding jobs from the completed task set under fixed-priority scheduling approaches.
- For those tasks which are persistent during the switch, it is difficult to assign priorities to their outstanding jobs at the switch point.

To address these problems, in this section, we propose the *instant switch* algorithm to support immediate mode switch. Instead of using fixed-priority-based deferrable scheduling approach like *DS-FP* when the system workload is heavy, *IS* adopts a dynamic deferrable scheduling algorithm called *DS-LALF* [16]. By scheduling outstanding jobs from the old mode together with the jobs in the new mode using the least available laxity first scheduling policy, *IS* can natively support asynchronous switch and offer more efficient switch point searching.

5.1 *DS-LALF* Principles

DS-FP is a fixed-priority update algorithm and the priorities of the update tasks are decided by the Shortest-Validity-First algorithm. In the non-clean switch scenarios, however, it is difficult to decide the priorities for the update tasks which have outstanding jobs at the MSR. Under the SVF algorithm, outstanding jobs with approaching deadlines but relatively larger validity intervals will be preempted by new update jobs in the new task set with relatively shorter validity intervals. This will make the outstanding jobs miss their deadlines and the temporal validity constraints of the corresponding real-time data will be violated during the mode switch. For this reason, in the non-clean switch scenarios, the deadlines or the actual laxities of the jobs (to be defined later) are better indicators of their urgencies and they are consistent with the new update tasks in the new mode.

To overcome this shortcoming of *DS-FP*, a dynamic enhancement of *DS-FP* called *Deferrable Scheduling with Least Actual Laxity First* (*DS-LALF*) [16], is proposed for assigning priorities to update jobs. *DS-LALF* decides their priorities using their actual laxities, i.e., the number of available idle time slots in the time interval $[d_{i,k-1}, d_{i,k}]$ after its previous update job $J_{i,k-1}$ has been completed. *DS-LALF* enqueues update jobs in ascending order of their actual laxities and always schedules the one with the least actual laxity first. As soon as a job $J_{i,j}$ is completed, the deadline $d_{i,j+1}$ of its next job $J_{i,j+1}$ is set to be $r_{i,j+1} + V_i$, and $J_{i,j+1}$ is enqueued. The separation between two consecutive jobs from the same update task is determined based on the total preemption from higher priority jobs at run-time. The details of *DS-LALF* and its schedulability test can be found in [16].

Example 5.1. Fig. 8 depicts a scenario where synchronous switch from *ML* to *DS-FP* does not work while asynchronous switch from *ML* to *DS-LALF* is successful. The MSR is issued at time 34 and no switch latency is allowed which means the switch must be executed immediately. In mode 1, the task set consists of $\tau_1 = (1, 8), \tau_2 = (3, 11)$ and $\tau_3 = (2, 12)$. In mode 2, a new task $\tau_4 = (1, 12)$ is added into the system. Under synchronous switch from *ML* to *DS-FP*, the switch fails because the task set in mode 2 is not schedulable under *DS-FP*. If the asynchronous switch from *ML* to *DS-LALF* is applied at time 34, the validity constraints of all the tasks during the switch from *ML* to *DS-LALF* are satisfied and a repeating pattern is found between times 55 and 85. Thus the asynchronous switch at time 34 is successful.

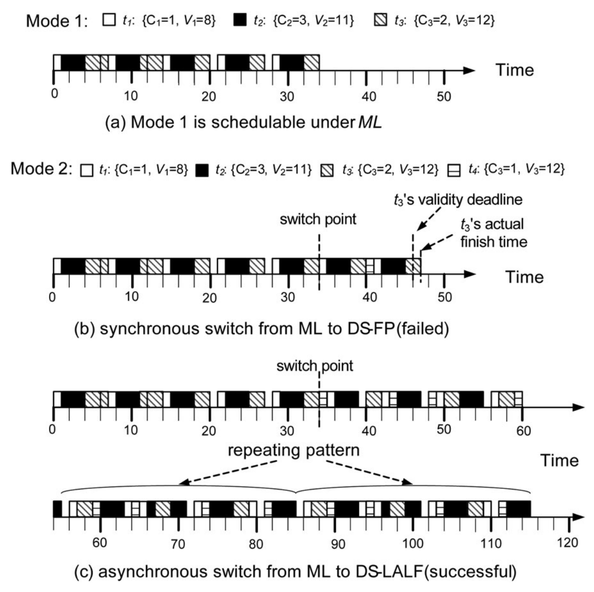

Mode 1: t1: {C1=1, V1=8} t2: {C2=3, V2=11} t3: {C3=2, V3=12}
Time
0 10 20 30 40 50
(a) Mode 1 is schedulable under ML
Mode 2: t1: {C1=1, V1=8} t2: {C2=3, V2=11} t3: {C3=2, V3=12} t4: {C3=1, V3=12}
t3's validity deadline
switch point
t3's actual finish time
Time
0 10 20 30 40 50
(b) synchronous switch from ML to DS-FP(failed)
switch point
0 10 20 30 40 50 60
repeating pattern
Time
60 70 80 90 100 110 120
(c) asynchronous switch from ML to DS-LALF(successful)

Fig. 8. An example of successful asynchronous switch.

5.2 Instant Switch (IS)

Algorithm 4 presents the framework for the *instant switch* algorithm. Similar to the *DS-LALF* algorithm, we maintain a queue, Q_{AS} and initialize it to contain both the outstanding jobs in $\mathcal{T}_k$ and the first jobs of the tasks in $\mathcal{T}_{k+1}^{+}$ to be scheduled in the system (Line 2). We define d_{max} as the largest absolute deadline of the jobs in Q_{AS} in the initial state (Line 3). Note that there is always one job per update task available for scheduling in Q_{AS}. We keep track of D_{max}, the latest absolute deadline among all the jobs that have already been scheduled, and partition all the jobs in Q_{AS} into two sets S_1 and S_2. The jobs with an absolute deadline no larger than D_{max} are put in S_1 (Line 14) while S_2 contains all the remaining jobs (Line 21). *IS* first schedules the job in S_1 with the least actual laxity (Line 15-18). If S_1 is empty, S_2 will be scheduled according to the same principle (Line 22-25). The actual laxity of an update job $J_{a,b}$ is calculated as **CalcIdleSlots**$(d_{a,b-1}, d_{a,b}) - C_{a,b}$ (Line 15 & 22), where the **CalcIdleSlots**$(d_{a,b-1}, d_{a,b})$ function calculates the number of idle slots in the time interval $[d_{a,b-1}, d_{a,b})$ and $C_{a,b}$ is the remaining execution time of $J_{a,b}$.

The **CalcReleaseTime(**$i, k, d_{i,k-1}, d_{i,k}$**)** function calculates the release time $r_{i,k}$ for job $J_{i,k}$ backwards from its deadline $d_{i,k}$, as long as $r_{i,k}$ is not smaller than the deadline $d_{i,k-1}$ of its previous job $J_{i,k-1}$ (Line 33-38). If job $J_{i,k}$ is not an outstanding job in $\mathcal{T}_k$, the deadline of its next job $J_{i,k+1}$ is computed as $d_{i,k+1} = r_{i,k} + V_i$, and $J_{i,k+1}$ is enqueued into Q_{AS} (Line 39-43). By time d_{max}, all the outstanding jobs of $\mathcal{T}_k$ and the first jobs from $\mathcal{T}_{k+1}$ have been scheduled. If the scheduling is successful, after time d_{max}, all the outstanding jobs in Q_{AS} are from $\mathcal{T}_{k+1}$ and the schedulability test algorithm [16] for *DS-LALF* which is based on pattern analysis will be performed (Line 46-51). Note that in Algorithm 4, we search in time window $[t_0, t_L)$ for a proper switch point when the asynchronous switch is successful. If the immediate mode switch is required, t_L will be reduced to 0 and the *IS* algorithm will only be applied at time t_0.

Algorithm 4. Instant Switch (IS) Algorithm

Input: $\mathcal{T}_k, \mathcal{T}_{k+1}, \Psi_k$, the search start time t_0 and t_L.
Output: A successful switch point and a schedule $\mathcal{S}$ if the validity constraint is satisfied during the switch.

```
1:  for $t = t_0$ to $t_0 + t_L$ do
2:      Enqueue all outstanding jobs from $\mathcal{T}_k$ and all the first jobs from $\mathcal{T}_{k+1}^{+}$ to $Q_{AS}$ in the ascending order of deadlines;
3:      $d_{max}$ = the largest deadline of the initial jobs in $Q_{AS}$;
4:      $r_{min}$ = the min release time of jobs from $\mathcal{T}_{k+1}$ in $Q_{AS}$;
5:      $D_{max} = 0$;
6:      result = TRUE;
7:
8:      while $r_{min} < d_{max}$ do
9:        $S_1 = S_2 = \varnothing$;
10:       $J_{i,k}$ = NULL; //The job with least actual laxity
11:       $N_{idle} = \infty$; //$J_{i,k}$'s actual laxity
12:       for each $J_{a,b}$ in $Q_{AS}$ do
13:         if $d_{a,b} \le D_{max}$ then
14:           $S_1 = S_1 \bigcup \{J_{a,b}\}$
15:           if CalcIdleSlots$(d_{a,b-1}, d_{a,b}) - C_{a,b} < N_{idle}$ then
16:             $N_{idle}$ = CalcIdleSlots$(d_{a,b-1}, d_{a,b}) - C_{a,b}$;
17:             $J_{i,k} = J_{a,b}$;
18:           end if
19:         else
20:           if $S_1 == \varnothing$ then
21:             $S_2 = S_2 \bigcup \{J_{a,b}\}$
22:               if CalcIdleSlots$(d_{a,b-1}, d_{a,b}) - C_{a,b} < N_{idle}$ then
23:               $N_{idle}$ = CalcIdleSlots$(d_{a,b-1}, d_{a,b}) - C_{a,b}$;
24:               $J_{i,k} = J_{a,b}$;
25:                 end if
26:               else
27:                 break;
28:               end if
29:             end if
30:           end for
31:           //Schedule the job according to DS-LALF
32:           Dequeue $J_{i,k}$ from $Q_{AS}$;
33:           $r_{i,k}$ = CalcReleaseTime$(i, k, d_{i,k-1}, d_{i,k})$;
34:           if $r_{i,k} < d_{i,k-1}$ then
35:             result = FALSE;
36:           else
37:             Update $r_{min}$;
38:           end if
39:           if $J_{i,k}$ is not an outstanding job from $\mathcal{T}_k^{-}$ then
40:             $d_{i,k+1} = r_{i,k} + V_i$;
41:             Enqueue $J_{i,k+1}$ to $Q_{AS}$;
42:             $D_{max} = \max\{D_{max}, d_{i,k}\}$;
43:           end if
44:         end while
45:
46:         if result == TRUE then
47:           Perform DS-LALF schedulability test on $Q_{AS}$;
48:           if the task set is schedulable then
49:             return $t$;
50:           end if
51:         end if
52:         Schedule the highest priority outstanding job with $\Psi_k$.
53: end for
54: return FAILURE; //the new task set is not schedulable
```

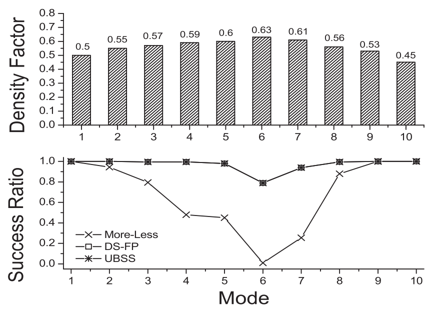


Fig. 9. Success ratio versus CPU utilization.

The worst case time complexity of **CalcIdleSlots** and **CalcReleaseTime** are both $O(V_{max})$ where $V_{max} = \max_{i=1}^{m}(V_i)$. Also, the worst-case time complexity of enqueue and dequeue can be $O(\ln m)$ if a priority queue is used. Thus, the time complexity of the while loop in the instant switch algorithm is $O((m+1) \cdot V_{max} + 2 \cdot \ln m)$.

6 Performance Evaluation

In this section, we present two sets of experiments for the performance evaluation on the proposed algorithms. The first set focuses on the performance comparison between single update algorithm and the online utilization-based scheduling switch (*UBSS*) algorithm. The second set compares the efficiency of the three switch algorithms, *search-based switch*, *adjustment-based switch* and *instant switch*, for searching proper switch points and performing online scheduling switch.

6.1 Simulation Model and Parameters

We define two categories of parameters in our experiments: system and update task parameters. Following the assumptions in [8], [15], we consider a simple configuration with a single CPU and a main memory based RTDBS. There are up to 10 modes in the system. The number of real-time data objects in the system varies from 1 to 25 and their validity intervals are uniformly distributed from 50 to 150 time units. In the experiments, we vary the number of update tasks to generate different update workloads to test their performance. To simplify the model parameters, it is assumed that each task updates one real-time data object, and its CPU time is uniformly distributed from 1 to 5 time units. Note that in the experiments, we do not aim at studying the performance for a particular *DCPS*. Instead, we would like to illustrate the performance of our proposed methods in achieving the goal of performing correct and efficient scheduling switch during mode switches. Thus, a wide range of workloads is used to test their performance by changing the number of update tasks and other important parameters such as the worst-case execution time and validity interval of an update task.

Following the definition of [26], we define the *density factor* of a set of tasks $\mathcal{T}$, denoted by γ, as $\sum_{i=1}^{m} \frac{C_i}{V_i}$. We define the *scheduling success ratio* of a given set of task sets, $S = \{\mathcal{T}_i\}_{i=1}^{m}$ under the update algorithm Ψ, as $\frac{n}{m} \times 100\%$ where n is the number of schedulable task sets in S under Ψ.

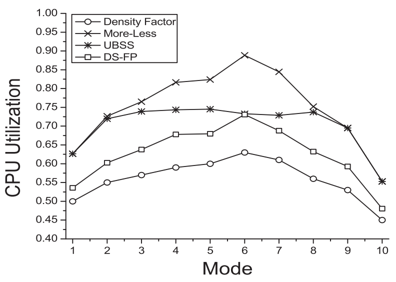


Fig. 10. Comparison of CPU utilization.

6.2 Performance Improvement with *UBSS*

In this set of experiments, we simulate the system fluctuation as a sequence of 10 consecutive modes and each mode has a fixed duration of 20,000 time units. The density factor for each mode is shown at the top of Fig. 9. 200 task sets are randomly generated in each mode and the scheduling success ratios for *ML*, *DS-FP* and *UBSS* are evaluated respectively. As shown in Fig. 9, *DS-FP* and *UBSS* always have the same scheduling success ratio in different modes with varying density factors. This is because at the beginning of each mode, *UBSS* conducts a schedulability test and select the most proper update algorithm for scheduling the tasks. If a task set is only schedulable under *DS-FP*, *UBSS* will choose *DS-FP* to maximize the schedulability. In Fig. 9, we also observe that the scheduling success ratios of all the three algorithms decrease with an increase in the density factor while *DS-FP* and *UBSS* outperform *ML* persistently.

Fig. 10 compares the CPU utilization among the three algorithms. As mentioned in Section 2, *DS-FP* can greatly reduce the CPU utilization compared with *ML* while still maintaining the temporal validity. This is verified in Fig. 10 where the CPU utilization of *DS-FP* is consistently lower than that of *ML* and the difference reaches 15.7 percent when the density factor is 0.63 in mode 6. As observed in Fig. 10, the CPU utilization of *UBSS* is between the *ML* and *DS-FP*. When the density factor is low, the CPU utilization of *UBSS* is close to *ML* because most of the task sets are schedulable under *ML* at that time and *UBSS* prefers choosing periodic update algorithm for maintaining higher data freshness and lower online scheduling overhead. On the other hand, when the system workload is high and most task sets are only schedulable under *DS-FP*, *UBSS* will let *DS-FP* take the control to maximize schedulability. Therefore, its CPU utilization is close to that of *DS-FP* at that time.

Figs. 11 and 12 demonstrate *UBSS*'s improvement over *DS-FP* in maintaining higher data freshness under different system workloads. Suppose mode M_k contains the task set

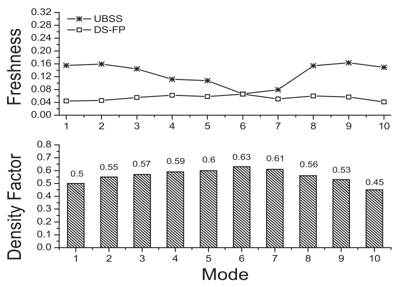


Fig. 11. Freshness w/ high system load.

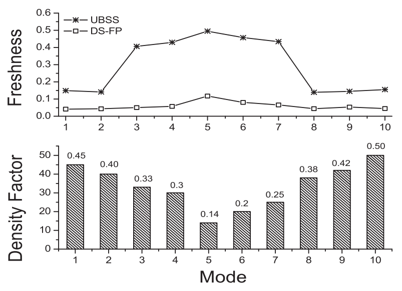


Fig. 12. Freshness w/ low system load.

$\mathcal{T}_k = \{\tau_i\}_{i=1}^m$ and τ_i $(1 \le i \le m)$ has released N_i update jobs by the end of M_k. We measure $\overline{S}_k$, the average data freshness of $\mathcal{T}_k$ by summing up each update job $(J_{i,j})$'s freshness at their finishing time $(f_{i,j})$ and divided by the total number of update jobs, i.e., $\overline{S}_k = \frac{\sum_{i=1}^m \sum_{j=0}^{N_i} S_t(f_{i,j})}{\sum_i^m N_i}$. Both Figs. 11 and 12 show that the average data freshness under *UBSS* is always higher than that of *DS-FP*. Fig. 12 further shows that the improvement increases when the system workload decreases. The main reason for this improvement is when the system workload is low, the task set has a high possibility to be schedulable under *ML* or even *HH*. This assignment greatly improve the data freshness. On the other hand, *DS-FP* always defers the release time of update jobs as late as possible provided that the temporal validity is still maintained. Both Figs. 11 and 12 show that the data freshness remains around 5 percent and is quite stable in the presence of system fluctuation.

Fig. 13 compares the online schedule computation time between *DS-FP* and *UBSS* in each mode. We observe that with the same scheduling success ratio, *UBSS* can greatly reduce the online scheduling overhead especially when the density factor is low. This is because in those scenarios, *ML* is in control most of the time and its online scheduling overhead is much lower than that of *DS-FP*.

6.3 Synchronous versus Asynchronous Switch

In this section, we first present a quantitative comparison of the scheduling success ratio among *ML*, *DS-FP* and *DS-LALF*. The update workload in the experiments is varied by increasing the number of update tasks in the system from 1 to 25.

As shown in Fig. 14, *DS-FP* consistently outperforms *ML* and *DS-LALF* in terms of scheduling success ratio. The scheduling success ratios of *ML* and *DS-LALF* drop below 0.65 when the number of update tasks increases to 18. This happens to *DS-FP* only when there are 20 update tasks in the system. Also, when the number of update tasks increases to 23, almost all task sets cannot be scheduled by *ML* and *DS-LALF* while the scheduling success ratios of *DS-FP* are still above 0.2. The results show that *DS-FP* is a better choice when the system update workload is heavy as it can accommodate more update tasks.

Fig. 15 compares the CPU utilization among *ML*, *DS-FP* and *DS-LALF* using the update task sets that are schedulable under all the three algorithms. The results show that the CPU utilization of *DS-LALF* is consistently lower than that of *DS-FP*, which is in turn consistently lower than that of *ML*. This observation suggests that if the update task set is schedulable under all the algorithms, *DS-LALF* is a better choice, because more available CPU resources can be allocated to other tasks (e.g., application tasks) for improved performance, including scheduling success ratio.

In the second set of experiments, we compare the efficiency of the three proposed switch algorithms when switch from *DS-FP* to *ML* or *DS-LALF*. The task sets before and after the switch, $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$, are simulated as follows. With fixed density factor $(\gamma = 0.6)$ and the number of tasks $(m = 20)$, we randomly generate $\mathcal{T}_k = \{\tau_i\}_{i=1}^m$ and make sure that $\mathcal{T}_k$ is only schedulable under *DS-FP*. $\mathcal{T}_{k+1}$ is defined as a subset of $\mathcal{T}_k$ and is specified by a given percentage p. $\mathcal{T}_{k+1}$ contains the first $\lceil m \times p\% \rceil$ tasks with higher priorities in $\mathcal{T}_k$, i.e., $\mathcal{T}_{k+1} = \{\tau_i\}_{i=1}^{\lceil m\times p\% \rceil}$. We set the switch latency as 2,000 time units and compare the scheduling success ratio and switch latency among the three algorithms. We conduct 200 experiments for each point to get the average value.

In Fig. 16, we observe that *IS* has the better performance than *SBS* and *ABS* consistently. Since it is executed immediately when the MSR is issued, its latency is always 0. We also observe that with a fixed density factor, unless the number of update tasks is very small (< 6), there is no failed scheduling switch from *DS-FP* to *DS-LALF* observed

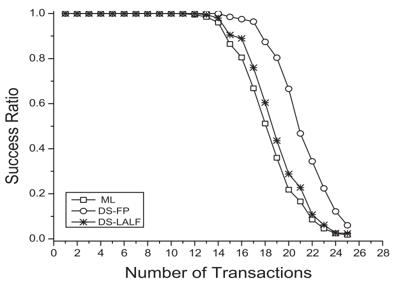


Fig. 14. Comparison of schedulability.

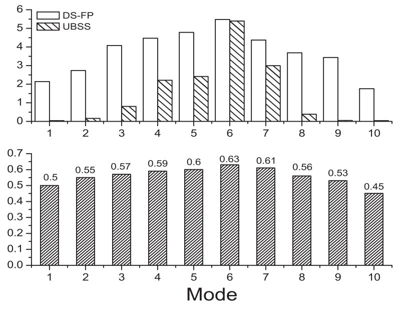


Fig. 13. Overhead versus CPU utilization.

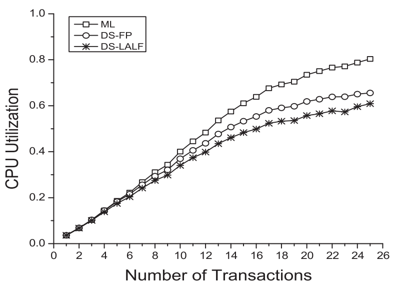


Fig. 15. Comparison of CPU utilization.

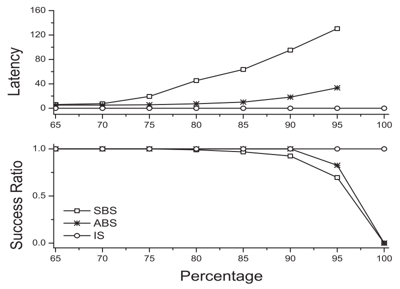


Fig. 16. Switch success ratio and latency.

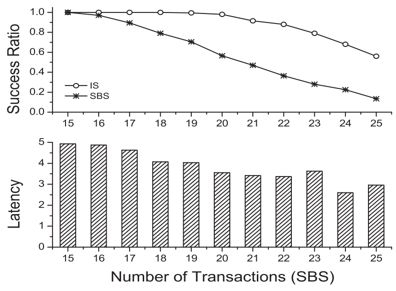


Fig. 17. Switch success ratio and latency.

under *IS*. This is mainly because most of the switch points are not the critical instances of the update tasks under *DS-LALF*. By contrast, the switch success ratio under *SBS* and *ABS*, are both decreasing when $\mathcal{T}_{k+1}$ increases its size. This is because the more tasks $\mathcal{T}_{k+1}$ has, the more temporal validity constraints to be applied on the switch point candidates. On the other hand, by proactively *creating* the switch point through schedule adjustment, *ABS* always outperforms *SBS* in terms of switch success ratio. As $\mathcal{T}_k$ is not schedulable under *ML*, when $p = 100\%$, $\mathcal{T}_{k+1}$ equals to $\mathcal{T}_k$ and the switch success ratios for both drop to 0. Fig. 16 also shows that *ABS* always has lower switch latency and the improvement keeps increasing when $\mathcal{T}_{k+1}$ increases.

In the third set of experiments, we further compare their performances when switching from *ML* to *DS-FP* or *DS-LALF*. Since *ABS* is mainly designed for *DS-FP*, here we focus on comparing *SBS* with *IS* for the switch latency and switch success ratio. In the experiments, the task set before the switch, $\mathcal{T}_k$, is randomly generated. It consists of 15 update tasks and is schedulable under *ML*. The task set after the switch, $\mathcal{T}_{k+1}$, is a superset of $\mathcal{T}_{k+1}$, and contains 0 to 10 additional new tasks with lower priorities. Other parameter settings are the same as the last experiments.

As shown in Fig. 17, with an increased workload in the system, the switch success ratios of *IS* and *SBS* both drop. However, the performance of *IS* is consistently better than that of *SBS*, and the improvement increases with the number of tasks. For the comparison of switch latency, since *IS* is conducted immediately after the MSR, its latency is always 0. The switch latency of *SBS* is also relatively small and decreases gradually when the number of update tasks in the system increases. This is mainly because when the system workload is heavy, most task sets are not schedulable. If the task set is schedulable, the switch point can be found easily.

In summary, our experimental results demonstrate that if dynamic priority scheduling is allowed in a multi-modal system, asynchronous scheduling switch approach should be applied during the mode switch for better switch success ratio and lower switch latency. For the synchronous scheduling switch approaches, *ABS* is more efficient than *SBS* in terms of switch points searching. It also shows that *UBSS* can maintain higher data freshness and significantly reduce the online scheduling overhead while still satisfying the temporal validity constraints of the tasks.

7 Conclusions and Future Work

In this paper we studied the problem of maintaining the temporal validity of real-time data in the presence of mode switches in *DCPS*. Our approach accommodates the use of different update algorithms in different modes to achieve a good balance between higher data freshness and better system schedulability. We introduced two synchronous switch algorithms for the clean switch scenarios and an asynchronous switch algorithm for the non-clean switch scenarios. Extensive experiments are conducted to illustrate the efficiency of the proposed switch algorithms.

We plan to further investigate the properties of the scheduling switch for a wider class of update algorithms, and implement the proposed algorithms in real systems. Another important future work is the co-scheduling of update tasks and application tasks with the purpose to guarantee the deadlines of all the update tasks while minimizing the number of deadline misses in application tasks.

Acknowledgments

An earlier version of this paper appeared in [1].

References

[1] S. Han, D. Chen, M. Xiong, and A. K. Mok, "Online scheduling switch for maintaining data freshness in flexible Real-time systems," in *Proc. 30th IEEE Real-Time Syst. Symp.*, 2009, pp. 115–124.

[2] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proc. 47th Design Autom. Conf.*, 2010, pp. 731–736.

[3] E. Lee and S. Seshia. (2014). Introduction to embedded Systems-A Cyber-physical systems approach, ed. 1.5 [Online]. Available: http://LeeSeshia.org, ISBN 978-0-557-70857-4

[4] G. Ozsoyoglu and R. Snodgrass, "Temporal and real-time databases: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 7, no. 4, pp. 513–532, 1995.

[5] K. Ramamritham, "Real-time databases," *Distrib. Parallel Databases*, vol. 1, no. 2, pp. 199–226, 1993.

[6] M. Xiong and K. Ramamritham, "Deriving deadlines and periods for real-time update transactions," *IEEE Trans. Comput.*, vol. 53, no. 5, pp. 567–583, May 2004.

[7] T. Gustafsson and J. Hansson, "Dynamic on-demand updating of data in real-time database systems," in *Proc. ACM Symp. Appl. Comput.*, 2004, pp. 846–853.

[8] M. Xiong, S. Han, K.-Y. Lam, and D. Chen, "Deferrable scheduling for maintaining real-time data freshness: Algorithms, analysis, and results, *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 952–964, Jul. 2008.

[9] J. Li, J.-J. Chen, M. Xiong, and G. Li, "Workload-aware partitioning for maintaining temporal consistency upon multiprocessor platforms," in *Proc. IEEE 32nd Real-Time Syst. Symp.*, 2011, pp. 126–135.

[10] J. Li, M. Xiong, V. Lee, L. Shu, and G. Li, "Workload-efficient deadline and period assignment for maintaining temporal consistency under edf," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1255–1268, Jun. 2013.

[11] D. Liberzon and A. Morse, "Basic problems in stability and design of switched systems," *IEEE Control Syst.*, vol. 19, no. 5, pp. 59–70, Oct. 1999.

[12] A. J. van der Schaft and H. Schumacher, *An Introduction to Hybrid Dynamical Systems*. London, U.K.: Springer, 1999.

[13] A. Burns and R. Davis, “Choosing task periods to minimise system utilisation in time triggered systems,” in *Proc. 18th IEEE Inf. Process. Lett.*, vol. 58, no. 5, pp. 223–229, 1996.

[14] S.-J. Ho, T.-W. Kuo, and A. K. Mok, “Similarity-based load adjustment for real-time data-intensive applications,” in *Proc. 18th IEEE Real-Time Syst. Symp.*, 1997, pp. 144–153.

[15] M. Xiong, S. Han, and K.-Y. Lam, “A deferrable scheduling algorithm for Real-time transactions maintaining data freshness,” in *Proc. 26th IEEE Real-Time Syst. Symp.*, 2005, pp. 27–37.

[16] S. Han, K.-Y. Lam, J. Wang, K. Ramamritham, and A. K. Mok, “On co-scheduling of update and control transactions in Real-time sensing and control systems: Algorithms, analysis and performance,” *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 10, pp. 2325–2342, Oct. 2013.

[17] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[18] S. Han, D. Chen, M. Xiong, K.-Y. Lam, A. Mok, and K. Ramamritham, “Schedulability analysis of deferrable scheduling algorithms for Maintainingreal-time data freshness,” *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 979–994, Apr. 2014.

[19] J. Real, “Mode change protocols for Real-time systems,” PhD thesis, Universidad Politécnica de Valencia, 2000.

[20] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham, “Mode change protocols for Priority-driven preemptive scheduling,” *Real-Time Syst.*, vol. 1, no. 3, pp. 243–264, 1988.

[21] K. W. Tindell, A. Burns, and A. J. Wellings, “Mode changes in priority pre-emptively scheduled systems,” in *Proc. Real-Time Syst. Symp.*, 1992, pp. 100–109.

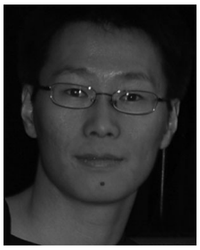
[22] P. Pedro and A. Burns, “Schedulability analysis for mode changes in flexible Real-time systems,” in *Proc. 10th Euromicro Workshop Real-Time Syst.*, 1998, pp. 172–179.

[23] J. Real and A. Crespo, “Mode change protocols for real-time systems: A survey and a new proposal,” *Real-Time Syst.*, vol. 26, no. 2, pp. 161–197, 2004.

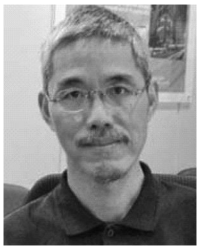
[24] R. Henia and R. Ernst, “Scenario aware analysis for complex event models and distributed systems,” in *Proc. 28th IEEE Int. Real-Time Syst. Symp.*, 2007, pp. 171–180.

[25] N. Stoimenov, S. Perathoner, and L. Thiele, “Reliable mode changes in real-time systems with fixed priority or edf scheduling,” in *Proc. Conf. Des., Autom. Test Eur.*, 2009, pp. 99–104.

[26] M. Xiong, Q. Wang, and K. Ramamritham, “On earliest deadline first scheduling for temporal consistency maintenance,” *Real-Time Syst.*, vol. 40, no. 2, pp. 208–237, 2008.



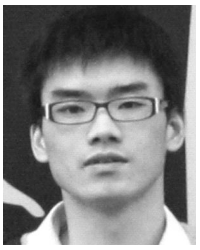
Song Han received the BS degree from Nanjing University in 2003, the MPhil degree from the CityU of Hong Kong in 2006, and the PhD degree from the University of Texas at Austin in 2012, all in computer science. He is currently an assistant professor in the Computer Science & Engineering Department at the University of Connecticut. His research interests include cyber-physical systems and networked embedded systems.



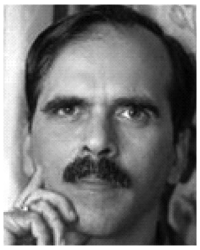
Kam-Yiu Lam received the BSc (Hons; with distinction) degree in computer studies and the PhD degree from the City University of Hong Kong in 1990 and 1994, respectively. He is currently an associate professor in the Department of Computer Science at the City University of Hong Kong. His research interests include real-time database systems, real-time active database systems, mobile computing, and distributed multimedia systems.

Deji Chen received the PhD degree in computer science from the University of Texas at Austin in 1999. He is currently a professor at Tongji University, People’s Republic of China. His research interests include real-time systems and wireless process control. He co-authored the book *WirelessHART - Real-time Mesh Network for Industrial Automation*. He is a member of the IEEE and the IEEE Computer Society.

Ming Xiong received the BS degree from Xian Jiaotong University, the MS degree from Sichuan University, and the PhD degree from the University of Massachusetts, Amherst, all in computer science. He is currently a technical staff at Google. From 2000 to 2009, he was a member of technical staff at the Bell Laboratories Research, Lucent Technologies. His research interests include real-time systems, database systems, and mobile computing.



Jiantao Wang received the BS degree in computer science from the University of Science and Technology of China (USTC) and the PhD degree in computer science from the City University of Hong Kong. He is currently a researcher at Huawei Technologies. His research interests include real-time systems, real-time data management, and flash-based database systems. He is a member of the IEEE.



Krithi Ramamritham received the PhD degree in computer science from the University of Utah. He is currently at IIT Bombay as a professor in the Department of Computer Science. He has served on numerous program committees of conferences and workshops. His editorial board contributions include *IEEE Transactions*, the *Real Time Systems Journal*, and the *VLDB Journal*. He is a fellow of the IEEE and ACM.

Aloysius K. Mok received the BS degree in electrical engineering, the MS degree in electrical engineering and computer science, and the PhD degree in computer science, all from the Massachusetts Institute of Technology. He is the Quincy Lee Centennial professor in computer science at the University of Texas at Austin. His current interests include real-time and embedded systems, robust and secure network centric computing, and real-time knowledge-based systems.