

Online Scheduling Switch for Maintaining Data Freshness in Flexible Real-Time Systems

Song Han*, Deji Chen†, Ming Xiong‡, Aloysius K. Mok*

* Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA

{shan, mok}@cs.utexas.edu

† Emerson Process Management, 12301 Research Blvd., Bldg. III, Austin, TX 78759, USA

Deji.Chen@Emerson.com

‡ Bell Labs, Alcatel-Lucent, Murray Hill, NJ 07974, USA

xiong@research.bell-labs.com

Abstract—Maintaining the temporal validity of real-time data is one of the crucial issues in a real-time database system. Past studies focus on designing algorithms to minimize imposed workload by a fixed set of update transactions while maintaining data freshness within validity intervals. In this paper we revisit this problem by investigating the cost of data freshness maintenance and online scheduling overhead in the presence of mode changes in real-time systems. We propose to apply periodic scheduling policies when the imposed update workload is low to maintain high data freshness. When the update workload becomes high, we propose to switch to more sophisticated algorithms to improve schedulability. In the latter case, not only each scheduling policy must be able to schedule the task set in the corresponding mode, temporal validity must also be maintained during the mode changes. To address this problem, two algorithms, named search-based switch (SBS) and adjustment-based switch (ABS) are proposed to search for the proper switch point online. SBS checks the temporal validity at the beginning time slot of each idle period while ABS further relaxes this restriction through schedule adjustment. Our experimental results demonstrate the correctness and efficiency of these two algorithms. Our results also show that scheduling switch according to the runtime processor workload can significantly outperform a single fixed scheduling policy in terms of data freshness while incurring only limited online switch overhead.

I. INTRODUCTION

Data quality is a serious issue in many application domains that require timely processing of massive amount of real-time data. The real-time data is used to capture the current status of entities in the system and is typically sampled and stored in a real-time database system (RTDBS). Different from traditional data stored in databases, real-time data have time semantics. A sampled value is valid only for a certain time interval [1]–[3] and its quality degrades with time until refreshed. The concept of *temporal validity* is first introduced in [1] to define the correctness of real-time data. Each real-time data object is associated with a *validity interval* that is the lifespan of the current data value. A new data value needs to be installed into the database by a corresponding update transaction before the validity interval of its old value expires. Otherwise, the RTDBS cannot detect and respond to environmental changes in a timely way. The concept of *staleness* [4] is used to measure the degradation in freshness of the current data value and it increases linearly within the validity interval until the data is refreshed.

In recent years, many efforts have been devoted to design and analyze algorithms to maintain the temporal validity of real-time data [1], [5]–[13]. In general, the simpler algorithms maintain higher data freshness and incur lower online scheduling overhead but impose much heavier update workload, while sophisticated

ones improve the schedulability by sacrificing data freshness. Most of the prior work assumes that the real-time system under study never changes and the update transaction set is fixed and persistent. However, many real-time systems in the real world exhibit multi-modal behavior and each mode is characterized by a set of functionalities that are carried out by different task sets¹. A typical example is an aircraft control system given in [14]. In the system, we can distinguish landing, takeoff and normal cruise modes and each mode consists of different task sets. To handle the switch among these modes, various mode change protocols have been proposed. Most of these works, however, stick to the same scheduling policy during the entire execution of the system. They do not consider the data freshness and do not explicitly address the problem of how to maintain the temporal validity constraints during the mode change.

In this paper we take a different approach from the past studies. We apply different scheduling policies in different modes based on the run-time processor workload. We use the more neutral term, *scheduling switch*, to emphasize that a change in resource allocation policy may be in response to concerns other than the domain-specific semantics of operation modes. What distinguishes our work from others is that our goal is to meet the temporal validity constraints not only before and after the mode change, but *during* the mode changes. We aim at achieving the tradeoff between higher data freshness and better schedulability. There are two important issues to be addressed in our problem: 1) which scheduling policy should be applied to a mode, and 2) when to conduct the switch thus the temporal validity can be maintained during the transition.

To address the first problem, our strategy is to select the policies under which the task sets are schedulable and we prefer the simple one when the system load is low to maintain higher data freshness and reduce the online scheduling overhead; On the other hand, when the system load increases, we would have to switch to more sophisticated policies to be able to accommodate more update transactions and meet their real-time requirements. To maintain the temporal validity during the mode change, we propose two algorithms, named search-based switch (SBS) and adjustment-based switch (ABS) to identify proper switch points when the temporal validity constraints can be satisfied. SBS checks the beginning time slot of each idle period after the release of the mode change request (MCR) and verifies whether it is a proper switch point; ABS further relaxes the restriction on the switch point candidates and *adjusts* the schedule between

¹In the following of the paper, we use "transaction" and "task" interchangeably.

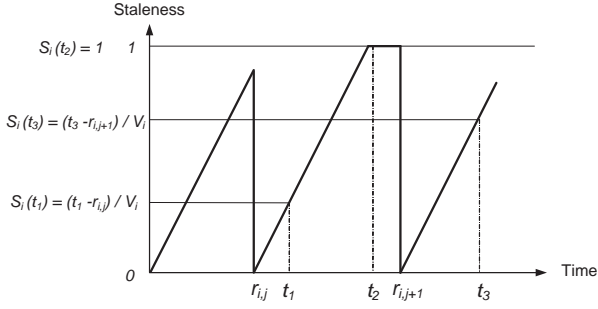


Fig. 1. Staleness of real-time data object X_i at time t_1, t_2 and t_3

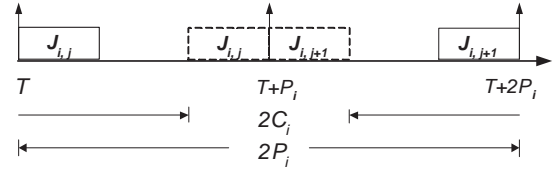


Fig. 2. Extreme execution cases of $J_{i,j}$ and $J_{i,j+1}$

the MCR and the current time slot. Compared with SBS, ABS greatly increases the number of switch point candidates and further improves the *promptness* [14] of the MCR.

The remainder of the paper is organized as follows. Section II summarizes existing mode change protocols and reviews prior work in maintaining temporal validity in RTDBS. Section III describes our task and mode change models. Section IV addresses the problem of how to decide the scheduling policies based on runtime processor workload. Section V studies the online scheduling switch problem and gets into more details with the switch between *ML* and *DS-FP*. Section VI presents our performance studies and we conclude the paper in Section VII and discuss the future works.

II. BACKGROUND

In this section, we briefly review the concept of *temporal validity* and *staleness* of real-time data. We also summarize three existing algorithms for maintaining temporal validity and review several well-known mode change protocols in the bibliography. For the reader's convenience, we summarize the frequently used symbols and their definitions in Table I.

A. Temporal Validity and Data Staleness

Definition II.1: A real-time data object (X_i) is temporally valid at time t if, for its latest (say, the j^{th}) update finished before t , the sampling time ($r_{i,j}$) plus the *validity interval* (\mathcal{V}_i) of the data object is not less than t , i.e., $r_{i,j} + \mathcal{V}_i \geq t$ [1]. \square

A data value for real-time data object X_i sampled at any time t will be valid up to $(t + \mathcal{V}_i)$ and the freshness of the data is measured by *staleness*, which is defined as follows.

Definition II.2: The *staleness* of a real-time data object (X_i) at time t , $S_i(t)$, is no larger than 1 and is the ratio of the difference between t and the sampling (or release) time of the latest (say, the j^{th}) update of τ_i finished before t to the *validity interval* (\mathcal{V}_i) of the data object, i.e., $S_i(t) = \min \left\{ \frac{t - r_{i,j}}{\mathcal{V}_i}, 1 \right\}$. The *freshness* of the real-time data object (X_i) at time t , $F_i(t)$, is $1 - S_i(t)$.

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

B. Algorithms for Maintaining Temporal Validity

In this paper, we take three algorithms as the candidates for our online scheduling switch. They are *Half-Half* (*HH*) [1],

[15], *More-Less* (*ML*) [10], [16], and deferrable scheduling (*DS-FP*) [12]. These algorithms impose diverse update workloads and have varying performances in maintaining data freshness. One of the contributions in our paper is to judiciously choose the algorithm among these candidates in each mode based on runtime workload and achieve the tradeoff between better schedulability and higher data freshness.

Half-Half: In *HH*, the period and relative deadline of an update transaction are each typically set to be one-half of the data validity length [1], [15]. In Figure 2, the farthest distance of two consecutive jobs of τ_i (based on the sampling time $r_{i,j}$ of job $J_{i,j}$ and the deadline $d_{i,j+1}$ of its next job) is $2P_i$. If $2P_i \leq \mathcal{V}_i$, then the validity of real-time object X_i is guaranteed as long as jobs of τ_i meet their deadlines.

More-Less: *ML* adopts the periodic task model [17] for update transactions whose derived deadlines are not larger than their periods. In *ML*, we only consider synchronous transactions whose first jobs all start at time 0 and there are three constraints to follow for transactions τ_i ($\forall i, 1 \leq i \leq m$) [10]:

- *Validity constraint:* the sum of the period and relative deadline of transaction τ_i is less than or equal to \mathcal{V}_i , i.e.,

$$P_i + D_i \leq \mathcal{V}_i \quad (1)$$

- *Deadline constraint:* the period of an update transaction 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 transactions, the *Deadline Monotonic* scheduling algorithm is used to schedule the transactions. 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 transactions based on *Shortest Validity First* (SVF), and ties are resolved in favor of transactions with larger execution time (C_i). It assigns deadlines and periods to τ_i as follows:

$$D_i = f_{i,0}^{ml} - r_{i,0}^{ml}, \quad (2)$$

$$P_i = \mathcal{V}_i - D_i, \quad (3)$$

where $f_{i,0}^{ml}$ and $r_{i,0}^{ml}$ are finishing and sampling times of the first job of τ_i , respectively. Note that in a synchronous system, $r_{i,0}^{ml} = 0$ and the first job's response time is the worst-case response time in *ML*. We use the superscript *ml* to distinguish the finishing and sampling times in *ML* from those in *DS-FP*.

DS-FP: *ML* is pessimistic on the deadline and period assignment. This is because it uses a periodic task model and the relative deadline D_i is equal to the worst-case response time of the

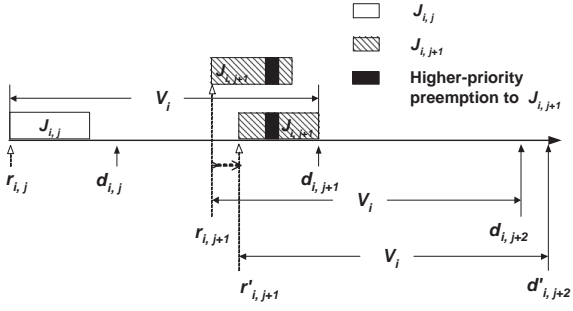


Fig. 3. Illustration of *DS-FP* scheduling: $r_{i,j+1}$ can be shifted to $r'_{i,j+1}$ without violating the **validity constraint**.

transaction. According to the *validity constraint* in *ML*, the larger the deadline D_i , the smaller the period P_i . To increase the separation of two consecutive jobs (and thus reduce the update workload), *DS-FP* adaptively derives the relative deadline and release time of one job from its previous job and preemptions from higher-priority transactions. Given release time $r_{i,j}$ of job $J_{i,j}$ and deadline $d_{i,j+1}$ of job $J_{i,j+1}$,

$$d_{i,j+1} = r_{i,j} + \mathcal{V}_i \quad (4)$$

guarantees that Eq.1 can be satisfied, as depicted in Figure 3. Correspondingly, Eq. 5 follows directly from Eq. 4:

$$(r_{i,j+1} - r_{i,j}) + (d_{i,j+1} - r_{i,j+1}) = \mathcal{V}_i. \quad (5)$$

If $r_{i,j+1}$ can be shifted onward to $r'_{i,j+1}$ along the time line in Figure 3, 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 guarantee Eq. 1.

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

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

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

where $R_{i,j+1}$ denotes the response time of $J_{i,j+1}$ deriving backwards from its deadline $d_{i,j+1}$. $\Theta_i(a, b)$ denotes the total *cumulative processor demands* made by all jobs of higher-priority transactions during the time interval $[a, b)$. Note that the schedule of all higher-priority jobs that are released prior to $d_{i,j+1}$ needs to be computed before computing $\Theta_i(r_{i,j+1}, d_{i,j+1})$.

Similar to *ML*, *DS-FP* also assigns priorities to transactions according to *SVF*. Readers are referred to [13] for the details of the *DS-FP* algorithm and Theorem II.1 states that *DS-FP* outperforms *ML* in terms of schedulability.

Theorem II.1: (Theorem 3.1 in [13]) Given a synchronous update transaction set \mathcal{T} with known C_i and \mathcal{V}_i ($1 \leq i \leq m$), if $(\forall i) f_{i,0}^{ml} \leq \frac{\mathcal{V}_i}{2}$ in *ML*, then

$$WCRT_i \leq f_{i,0}^{ml}$$

where $WCRT_i$ and $f_{i,0}^{ml}$ denote the worst-case response time of τ_i in *DS-FP* and *ML*, respectively.

C. Mode Change Protocols

In the literature, mode change protocols can be classified into synchronous and asynchronous protocols with regard to the way old and new-mode tasks are combined during the mode change. [18] proposes two synchronous protocols, one with periodicity and the other without. These protocols assume that old-mode tasks may be completed if they have outstanding execution when the MCR is issued. The old-mode tasks may or may not have further jobs released depending on whether the periodicity is to be satisfied during the mode change. The offset to the MCR is obtained by summing up the worst-case execution time of all old-mode tasks.

An analysis approach for mode changes on single processor system with rate-monotonic scheduling is introduced in [19]. This protocol is based on dynamic processor utilization and the rules of the priority ceiling protocol. [20] improves and extends [19] to deadline-monotonic scheduling and with the periodicity maintained. Worst-case response time analysis is given in [20] for each type of tasks during the mode change. [21] further generalizes [20] by introducing possible offsets for all the new-mode tasks, no matter whether they are changed or unchanged tasks thus relaxes the periodicity requirement. The timing analysis in [21] assumes a pre-determined offset for each task and [14] introduces a slightly different protocol and proposes an algorithm to calculate the offsets and achieve the tradeoff between the schedulability and the promptness.

All the analysis methods in [14], [19]–[21] are limited to strictly periodic task activation and [22] eliminates this restriction by allowing complex task activation patterns including periodic with jitter, periodic with burst and sporadic event models. [23] further improves [22] by supporting any event stream model and it can handle both the earliest deadline first (EDF) and fixed priority (FP) scheduling of tasks.

[24] studies the mode change problem from another direction and it aims at configuring tasks within a system judiciously so that task migrations and priority changes are minimized during mode changes.

Different from prior work, our work focuses on maintaining the temporal validity of the tasks which do not change during the mode switch by applying different scheduling policies in different modes depending on the runtime processor utilization. Our work applies to synchronous protocols and all the new-mode tasks are released with the same offset to the MCR. Unlike [18], our algorithm can *adjust* the schedule of the old-mode tasks between the MCR and the launching time of all the new-mode tasks, thus greatly improving the promptness while still maintaining the temporal validity.

III. PRELIMINARIES

A. Task and Mode Change Model

We model the operational dynamics in a real-time system as a series of different modes, M_0, M_1, M_2, \dots , and each mode M_k contains a fixed task set $\mathcal{T}_k = \{\tau_i\}_{i=1}^m$ with known C_i and \mathcal{V}_i for each τ_i ($1 \leq i \leq m$). In our model, a scheduling policy Ψ_k is applied to \mathcal{T}_k in mode M_k and following the general assumptions in the prior work, we assume that MCR is a sporadic event and

Symbol	Definition
X_i	Real-time data object i ($i = 1, \dots, m$)
τ_i	Update transaction updating X_i
$J_{i,j}$	The j^{th} job of τ_i ($j = 0, 1, 2, \dots$)
$R_{i,j}$	Response time of $J_{i,j}$
C_i	Computation time of transaction 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 of $J_{i,j}$
P_i	Period of transaction τ_i in ML
D_i	Relative deadline of transaction τ_i in ML
M_i	The i^{th} mode in the system ($i = 0, 1, 2, \dots$)
\mathcal{T}_i	The fixed task set in mode M_i
Ψ_i	The scheduling policy applied on \mathcal{T}_i
\mathcal{T}_k^c	The changed task set in mode change $M_k \rightarrow M_{k+1}$
\mathcal{T}_k^u	The unchanged task set in mode change $M_k \rightarrow M_{k+1}$
\mathcal{T}_k^-	The complete task set in mode change $M_k \rightarrow M_{k+1}$
\mathcal{T}_k^+	The new task set in mode change $M_{k-1} \rightarrow M_k$
t_{MCR}	The issue time of the mode change request (MCR)
t_L	The latency requirement of the MCR
t_w	The switch time in the mode change
$r_{i,j}^k$	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 processor demands from higher-priority transactions received by τ_i in interval $[a, b]$

TABLE I

Symbols and definitions.

it cannot occur during the mode transitions. There are four types of tasks in our model:

Complete tasks are tasks that are active in the old-mode but do not appear in the new-mode. They are allowed to run to complete under the old scheduling policy after the MCR with no new job released. They *cannot* be aborted for the purpose of maintaining its temporal validity.

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

Unchanged tasks are tasks that are persistent through the mode change. They are executed and released after the MCR under the old scheduling policy until the new scheduling policy takes the control. The switch point should be carefully selected to maintain the temporal validity during the transition.

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.

B. Notations and Definitions

We assume that $\mathcal{T}_k, \mathcal{T}_{k+1}$ are the two task sets before and after the mode change and they are schedulable under scheduling policies Ψ_k and Ψ_{k+1} respectively. Let \mathcal{T}_k^c and \mathcal{T}_k^u denote the changed and unchanged task set during the switch respectively. According to the definition in Section III-A, \mathcal{T}_k^c and \mathcal{T}_k^u are the only tasks that appear in both modes and we have $\mathcal{T}_k^c \cup \mathcal{T}_k^u = \mathcal{T}_k \cap \mathcal{T}_{k+1}$. We further have the complete task set $\mathcal{T}_k^- = \mathcal{T}_k - \mathcal{T}_k^c - \mathcal{T}_k^u$ and the new task set $\mathcal{T}_{k+1}^+ = \mathcal{T}_{k+1} - \mathcal{T}_k^c - \mathcal{T}_k^u$.

Following tradition, we denote by t_{MCR} the time when the MCR is issued and t_L the MCR latency requirement. That is, the mode change must be conducted in $[t_{MCR}, t_{MCR} + t_L]$. Our study in this paper focuses on the following scenario: A real-time system in mode M_k is initially controlled by Ψ_k . At

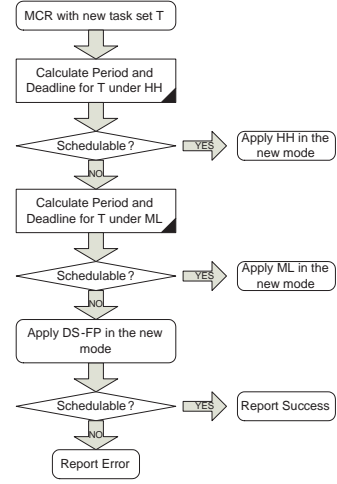


Fig. 4. Utilization-based Scheduling Selection

time t_{MCR} , the system is notified by a change of the task set and is requested to finish a scheduling switch before time $t_{MCR} + t_L$. After t_{MCR} , the tasks in \mathcal{T}_k^- are run to complete and tasks in $\mathcal{T}_k^c \cup \mathcal{T}_k^u$ are executed and released as normal under Ψ_k until a certain time point t_w ($t_{MCR} \leq t_w \leq t_{MCR} + t_L$) when the task set to be executed is changed to \mathcal{T}_{k+1} and a new policy Ψ_{k+1} is in control. The problem is how to choose Ψ_k and Ψ_{k+1} to achieve the tradeoff between data freshness and schedulability, and how to find the valid switch point that preserves the temporal validity of the tasks in $\mathcal{T}_k^c \cup \mathcal{T}_k^u$ during the transition.

In the case that a certain task $\tau_i \in \mathcal{T}_k^c$ whose validity interval is changed from V_i to V'_i ($V_i \neq V'_i$) during the switch, it is difficult to identify whether the temporal validity is satisfied during the switch because it's not specified which validity interval should be used. To avoid the ambiguity, we introduce the concepts of strict and weak temporal validity as follows.

Definition III.1: During the mode change from M_k to M_{k+1} and at the switch point t_w , strict temporal validity of $\tau_i \in \mathcal{T}_k^c$ is satisfied if, for the release time ($r_{i,j}^k$) of its latest job (say, the j^{th} job) that finishes before t_w under Ψ_k and the deadline ($d_{i,0}^{k+1}$) of its first job after t_w under Ψ_{k+1} , the difference between $r_{i,j}^k$ and $d_{i,0}^{k+1}$ does not exceed the smaller validity interval. That is, $d_{i,0}^{k+1} - r_{i,j}^k \leq \min\{V_i, V'_i\}$. Weak temporal validity is satisfied if the difference does not exceed the larger validity interval. That is, $d_{i,0}^{k+1} - r_{i,j}^k \leq \max\{V_i, V'_i\}$ \square

IV. UTILIZATION-BASED SCHEDULING SELECTION

Prior work on mode change protocols mostly assume that the task sets before and after the mode change are controlled by the same scheduling policy. However, this assumption does not always hold in the real-world scenarios. To maintain the temporal validity in a flexible real-time system, the first problem to be addressed is how to select the proper scheduling policy for each mode so that the corresponding task set is schedulable. As we have mentioned in Section I, our strategy is to apply the periodic scheduling policies when the imposed update workload is low as long as the task set is schedulable. Because they maintain higher data freshness with lower online scheduling overhead. We only have to switch to more sophisticated policies when the schedulability bounds for periodic policies are exceeded. In this

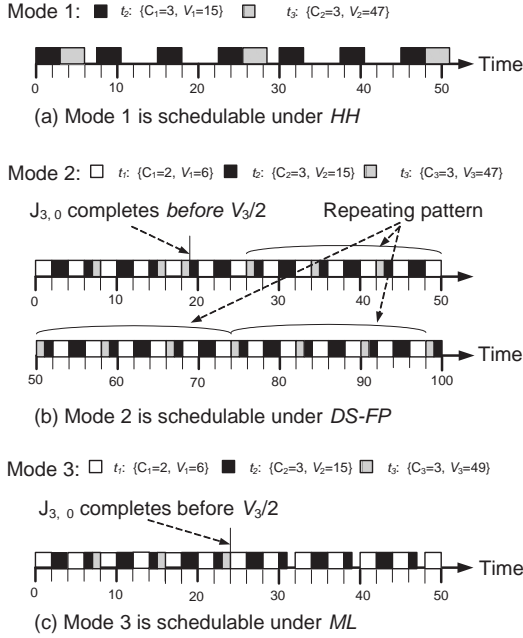


Fig. 5. Example of Utilization-based Scheduling Selection

way, we achieve maintaining the temporal validity by degrading the data freshness.

In this paper, we consider three candidate scheduling policies: *HH*, *ML* and *DS-FP*. *DS-FP* has the best schedulability among all three candidates but the worst data freshness. We choose the scheduling policy based on the imposed workload from the update transactions and the selection process is summarized in Figure 4. When the system is notified with a MCR, it will calculate the periods and deadlines of the new task set under *HH* and *ML*, respectively. Theorem 5 in [25] 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 adopt *DS-FP* when both *HH* and *ML* do not work. If the task set is not schedulable under *DS-FP*, the system will report error because there are no better scheduling algorithms available for maintaining temporal validity.

Example IV.1: Consider three consecutive modes, M_1, M_2 and M_3 in a real-time system which is shown in Figure 5. In M_1 , there are two transactions τ_2, τ_3 with computation times 3, 3 and validity intervals 15, 47, respectively. In M_2 , a new task τ_1 is added into the system with computation time 2 and validity interval 6. 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 0.525. This is below 0.828 which is the bound evaluated by Theorem 5 in [25]. 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 $V_3/2$ (that is 23.5). Thus, this new transaction set is not schedulable by either *ML* or *HH*. On the other hand, the same transaction set is schedulable by *DS-FP*, because the schedule pattern between time 26 and 50 repeats itself forever. In M_3 , with τ_3 's validity interval increased to 49, the new task set is not schedulable by *HH*, but schedulable under *ML* because the deadline constraint is satisfied. □

V. SCHEDULING SWITCH WITH VALIDITY CONSTRAINT

Even though both the old and new task sets are schedulable under the selected scheduling policies, 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 scheduling switch. To satisfy all these temporal validity constraints, the switch point should be carefully selected. In this section we first investigate two different switch scenarios, the clean switch and non-clean switch. Based on the clean switch scenario, we propose two algorithms for searching the proper switch points. They are the search-based switch (SBS) and adjustment-based switch (ABS). Some theoretical results are further presented which are related to the switch between *ML* and *DS-FP*.

A. Clean Switch vs. Non-clean Switch

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

For tasks in $\mathcal{T}_k^c \cup \mathcal{T}_k^u$, they are schedulable up until t_w . Ψ_{k+1} may be affected by their last jobs before t_w . Some scheduling policies are preconditioned upon the fixed starting times of the first jobs. For example all tasks start at time 0. Similar problems arise if a task in $\mathcal{T}_k^c \cup \mathcal{T}_k^u$ has outstanding execution at t_w . Should this job be considered the first job under 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; further more, it also interferes with Ψ_{k+1} just as those tasks in \mathcal{T}_k^- could do. In either case Ψ_{k+1} cannot have a clean start.

Example V.1: Figure 6 depicts a non-clean switch of simple periodic task sets in which all tasks start at time 0 and deadline equals period. $\mathcal{T}_{k+1} = \mathcal{T}_k = \{\tau_1 = (2, 4), \tau_2 = (3, 8)\}$, $\Psi_k = \Psi_{k+1} = HH$, and $t_w = 6$. In the figure, (a) is the schedule of Ψ_k plus what could have been after t_w ; (b) is the schedule of Ψ_{k+1} which treats t_w as time 0. In this run, τ_2 's outstanding job is run to the finish but preempted by τ_1 's first job after t_w . So it misses its deadline defined in Ψ_k . Interestingly all jobs after t_w meet deadlines in spite of this extra execution. □

The above observation is the trivial case; accordingly, we only study the switch cases where:

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

We call such a switch scenario a clean switch; otherwise we call it a non-clean switch. In this paper, our switch point searching algorithms will focus on the clean switch scenario. We will briefly talk about the extension to the non-clean switch scenario in Section VII.

B. Search-based Switch

Consider a mode change from M_k to M_{k+1} . For a task in $\mathcal{T}_k^c \cup \mathcal{T}_k^u$ such that its last job before t_w and first job after t_w

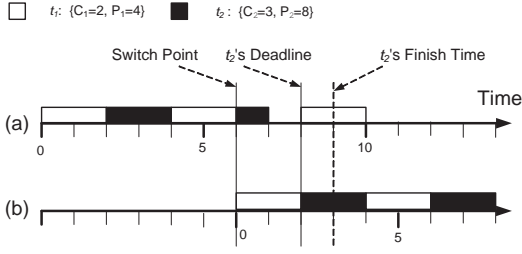


Fig. 6. Non-clean switch from HH to HH with outstanding execution (Failed)

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 V.2: Figure 7 depicts a clean switch of validity constrained task sets. $\mathcal{T}_k = \{\tau_1 = (4, 16), \tau_2 = (5, 26)\}$. We switch the same task set from $\Psi_k = DS-FP$ to $\Psi_{k+1} = HH$ at $t_w = 33$. Modeled in HH , $\mathcal{T}_{k+1} = \{\tau_1 = (4, 8), \tau_2 = (5, 13)\}$. In the figure, (a) is the schedule of Ψ_k plus what could have been after t_w ; (b) is the schedule of Ψ_{k+1} which treats t_w as time 0. For τ_1 , its last job before t_w starts at time 24 and its first job after t_w finishes at time 37. The distance is 13, which is less than \mathcal{V}_1 . 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 bigger than \mathcal{V}_2 . So τ_2 's validity constraint is violated during the switch. \square

Example V.2 exposes hidden real-time requirements that could be missed. After all, all tasks meet their real-time requirements before and after the switch. What else should we consider? Introducing validity constraint has many advantages. First, it enables us to model the same real-time application in both \mathcal{T}_k and \mathcal{T}_{k+1} , potentially different task models. Second, it abstracts out the real-time constraints during the switch so that Ψ_k and Ψ_{k+1} could be independently applied as they are originally devised. Third, it allows the same task to fluctuate as well. For example, the execution time of a task in \mathcal{T}_k^c may change to a different value after t_w .

We define a successful switch to be one that for all task $\tau_i, \tau_i \in \mathcal{T}_k^c \cup \mathcal{T}_k^u$, 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. This is also more realistic in practice. A real-time system normally tolerates some delay (t_L) in its adjustment to the mode change. Or a successful switch point could be pre-calculated and the switch be applied before the anticipated mode change occurs.

Next we study the properties of t_w . 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 $t_1 = t_2$. Since we are talking about clean switch, obviously t_w falls within some idle period and we have the following lemma.

Lemma V.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

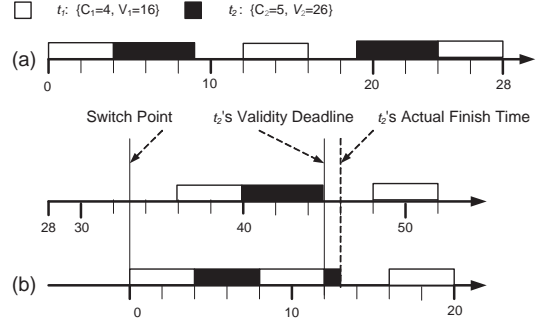


Fig. 7. Clean switch from $DS-FP$ to HH

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 $\mathcal{T}_k^c \cup \mathcal{T}_k^u$ will have an earlier finish time hence will meet the temporal validity constraint. \square

Note in Figure 7(b), 33 is not a successful switch point, but time from 28 to 32 are all successful switch points. This gives us the following search-based switch algorithm for t_w .

Alg 1 Search-based Switch 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  $\tau_i \in \mathcal{T}_k \cap \mathcal{T}_{k+1}$  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:     if  $f = true$  then
12:       return  $t$ ;
13: return no  $t_w$  exists;

```

As all schedulable schedules have a repeating pattern in discrete time system [26], if t_L , the latency requirement of the MCR is not explicitly given, Alg. 1 runs at most the length of the shortest pattern of the schedule, \mathcal{P}_L , and will eventually terminate. It will report failure if there is no successful switch point. Otherwise, if there is any successful switch point, then the algorithm will always return it.

Note in the above algorithm we do not require that the parameters of a task is the same before and after the switch, which means it can apply to the tasks both in \mathcal{T}_k^c and \mathcal{T}_k^u .

C. Adjustment-based Switch

Search-based switch (SBS) is straight forward and easy to be implemented for searching the proper switch point online. However, SBS restricts the switch point candidates only at the beginning of the idle periods in $[t_{MCR}, t_{MCR} + t_L]$. This constraint severely limits the number of possible candidates and reduces the promptness of the mode change.

We want to remove this restriction and take every time point in $[t_{MCR}, t_{MCR} + t_L]$ as the candidate for scheduling switch. However, this extension will put us in the non-clean switch scenario because there could be outstanding execution for certain

jobs at that time. In this section, we propose the adjustment-based switch algorithm (ABS) to address this problem. ABS converts the non-clean switch scenario to clean switch scenario through schedule adjustment. By schedule adjustment, we mean changing release times and deadlines of jobs. The basic idea of ABS is, at a certain time t ($t_{MCR} \leq t \leq t_{MCR} + t_L$), we push all the outstanding execution back to t . That is, all the unfinished jobs in \mathcal{T}_k must be finished by t in the adjusted schedule. Then the schedule in $[t_{MCR}, t]$ will be adjusted backwards from time t so that the adjusted schedule is valid for guaranteeing the validity constraints for all update transactions. Notice that if transaction τ_h is the highest priority transaction in \mathcal{T}_k whose schedule needs to be adjusted, then the schedule of all lower-priority transactions τ_i ($h < i \leq m$) in \mathcal{T}_k also needs to be adjusted due to the impact of release time and deadline adjustment of τ_h . After the schedule adjustment, ABS will release all the tasks in \mathcal{T}_{k+1} at time t and checks whether for each task $\tau_i \in \mathcal{T}_k^c \cup \mathcal{T}_k^u$, it can maintain the temporal validity during the scheduling switch.

Alg 2 Adjustment-based Switch 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:      $f = \text{ScheduleAdjustment}(\mathcal{T}_k, t_0, t)$ ;
8:     if  $f = \text{fail}$  then
9:       continue;
10:    else
11:      for each  $\tau_i \in \mathcal{T}_k \cap \mathcal{T}_{k+1}$  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:           $f = \text{fail}$ ;
17:        if  $f = \text{success}$  then
18:          return  $t$ ;
19: return no  $t_w$  exists;
```

The framework of the adjustment-based switch is presented in Alg.2. We denote by $I(a, b)$ the total number of idle slots between $[a, b]$, and $\Gamma_i(t)$ the outstanding execution of τ_i at time t . In Alg.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 execution. Line 7 is the core of the algorithm. The function *ScheduleAdjustment* (T, t_0, t) tries to push back all the outstanding execution of transaction set T at time t and adjust the schedule in $[t_0, t]$ to satisfy the temporal validity constraints. Alg. 3 presents the details of this adjustment process. If the adjustment is successful, line 13-18 further verifies whether the temporal validity is also maintained during the switch. Alg.2 sequentially checks each time slot in $[t_0, t_0 + t_L]$ and it returns the earliest proper switch point or reports failure when time $t_0 + t_L$ is reached.

Alg.3 summarizes the details of the schedule adjustment. In the algorithm, line 1 first identifies τ_h , the transaction with the highest priority in \mathcal{T}_k who has outstanding execution at time t . For each transaction 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.

Alg 3 ScheduleAdjustment (T, t_0, t)

Input: Transaction 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:  $h = \min_i \{i | \tau_i \in T \text{ and } \tau_i \text{ has outstanding execution at } t.\}$ 
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:      if ( $(r'_{i,j} < d_{i,j-1})$ ) then
17:         $d'_{i,j-1} = r'_{i,j}$ ;
18:      else
19:         $d'_{i,j-1} = d_{i,j-1}$ ;
20:       $j = j - 1$ ;
21:    else
22:      // No adjustment for this job
23:      if ( $d'_{i,j} - \Theta(t_0, d'_{i,j}) - C_i < t_0$ ) then
24:        return fail; // cannot adjust the schedule before  $t_0$ 
25:      else
26:        if ( $t_s \geq d'_{i,j}$ ) then
27:           $t_s = d'_{i,j}$ ;
28:          break;
29:        else
30:           $d'_{i,j-1} = d_{i,j-1}$ ;
31:           $j = j - 1$ ;
32:          if ( $(j = 0) \wedge (d'_{i,j} \neq d_{i,j})$ ) then
33:            return fail;
34: return adjusted  $S$  in  $[t_0, t]$  and  $\forall i, r'_{i,k_i}$ ;
```

Alg.3 adjusts the release time and deadline for each job of these transactions 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 temporal validity is still maintained after the adjustment. Failure will be reported if either of the conditions is not met. Similar checking will also be conducted in line 23 even when no adjustment is conducted. Line 16-20 adjusts the deadline of the previous job for further processing. If a successful adjustment cannot be achieved even when all the jobs are tested, line 34 will report failure. Otherwise, the successfully adjusted schedule will be returned.

Example V.3 shows a scenario where the adjustment-based switch outperforms the search-based switch.

Example V.3: Following the same task set as in Example V.2, Figure 8 depicts a scenario where search-based switch does not work while adjustment-based switch is successful. The MCR is issued at time 33 and its latency requirement is 12 which means the scheduling switch must be finished before time 45. The only switch candidate under search-based switch is time 33 but it does not satisfy the validity constraint for τ_2 during the transition.

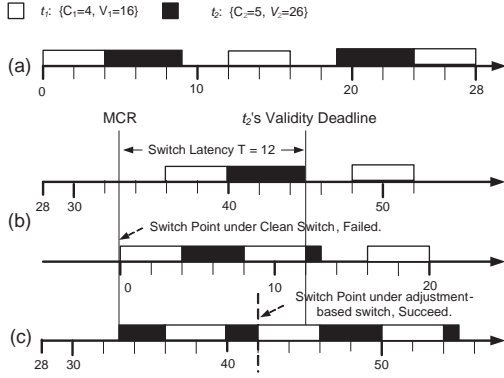


Fig. 8. An example of successful adjustment-based switch

However, if adjustment-based switch is applied at time 42, the outstanding execution of τ_2 is 3 and they 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. This distance is 22 which is smaller than V_2 . So the validity constraint during the switch is satisfied. \square

D. Properties of Switch between ML and DS-FP

In this section, we investigate further and look at the switch specifically between *ML* and *DS-FP*. *HH* can be taken as a special case of *ML* with $P_i = D_i = \frac{V_i}{2}$. Theorem V.1 and Theorem V.2 present two interesting properties of the switch between them.

Theorem V.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 $\mathcal{T}_k^c \cup \mathcal{T}_k^u$ is met during the switch. Let $\tau = (C, P)$ and t_w is 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 $(t_w - t) + D \leq P + D = \mathcal{V}$. \square

Note Theorem V.1 holds when $\mathcal{T}_k \neq \mathcal{T}_{k+1}$. It also holds if the execution time of tasks in $\mathcal{T}_k^c \cup \mathcal{T}_k^u$ changes during the switch as long as *ML* schedules both \mathcal{T}_k and \mathcal{T}_{k+1} . If validity interval also changes, we then have to go back to Alg. 1 for searching successful switch points to maintain strict temporal validity.

Theorem V.2: For \mathcal{T}_k and \mathcal{T}_{k+1} , if $\mathcal{T}_{k+1} \subseteq \mathcal{T}_k$, $\Psi_k = ML$ and $\Psi_{k+1} = DS-FP$, 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 by *ML*, so is its subset \mathcal{T}_{k+1} . So the worst-case execution time of τ under *ML* is no bigger than D . Furthermore, according to Theorem II.1, the worst-case execution time of τ under *DS-FP* is no bigger than D . So the first job of τ after t_w finishes 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 $(t_w - t) + D \leq P + D \leq \mathcal{V}$. \square

Note that theorem V.2 may not hold when $\mathcal{T}_{k+1} \not\subseteq \mathcal{T}_k$. Even if $\mathcal{T}_{k+1} \subseteq \mathcal{T}_k$, we cannot extend the result to cases where the

execution time or validity interval changes. 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 V.2.

VI. PERFORMANCE EVALUATION

This section presents simulation results of the online scheduling switch in flexible real-time systems. We present two sets of experiments for the performance evaluation. The first set of experiments focuses on the comparison between single scheduling policy and online utilization-based scheduling switch (*UBSS*). The second set compares the ability and efficiency of the two algorithms, search-based switch (*SBS*) and adjustment-based switch (*ABS*), for searching proper switch points. Our goal is to study the efficiency of the invented algorithms and verify that *UBSS* can maintain higher data freshness and significantly reduce the online scheduling overhead while satisfying the temporal validity constraints over the entire execution of the system.

A. Simulation Model and Parameters

Two categories of parameters are defined: system and update transaction parameters. For system configurations, we consider a single CPU, 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 20 and the validity length is uniformly distributed from 50 to 150 time units. It is assumed that each transaction updates one real-time data object, and its CPU time is uniformly distributed from 1 to 5 time units. Following the definition of [27], we define the *density factor* of a set of transactions \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 transaction sets, $S = \{\mathcal{T}_i\}_{i=1}^m$ under the policy Ψ , as $\frac{n}{m} \times 100\%$ where n is the number of schedulable task sets in S under Ψ . The primary performance metrics used in our experimental studies are the CPU utilization, the scheduling success ratio, the data staleness, the scheduling overhead and the switch latency.

B. Performance Improvement with UBSS

In this set of experiments, we simulate the fluctuation of the system as a sequence of 10 consecutive modes and each mode has a fixed duration of 20000 time units. The density factor for each mode is shown at the top of Figure 9. 200 task sets are randomly generated in each mode and the scheduling success ratios for *ML*, *DS-FP* and *UBSS* are evaluated respectively. The comparison is illustrated at the bottom of Figure 9. In the figure, our first important observation is, *DS-FP* and *UBSS* always have the same success ratio in different modes with varying density factors. The reason is at the beginning of each mode, *UBSS* will conduct a schedulability test and select the most proper policy for scheduling. If a task set is only schedulable under *DS-FP*, *UBSS* will choose *DS-FP* for the purpose of maximized schedulability. In Figure 9, we also observe that the success ratios of all three approaches drop along with the increase of the density factor while *DS-FP* and *UBSS* outperform *ML* persistently through the whole system. The success ratio of *ML* drops to 0.01 when the density factor climbs to 0.63 while *DS-FP* and *UBSS* can still maintain a 0.79 success ratio. All three approaches have the same success ratio when the density factor is below 0.55 because all task sets are schedulable under *ML* at that time.

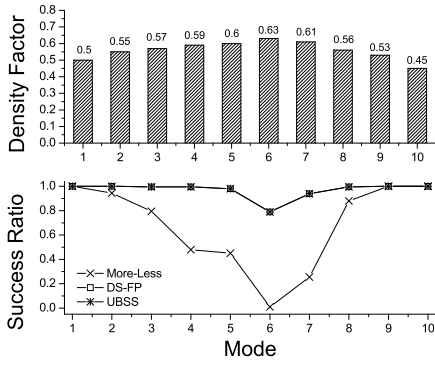


Fig. 9. Success Ratio vs. CPU Utilization

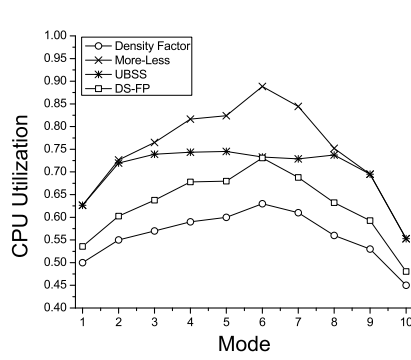


Fig. 10. Comparison of CPU Utilization

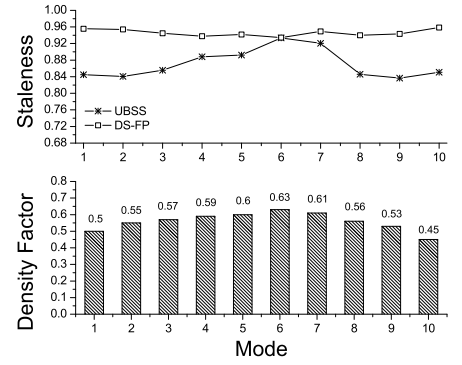


Fig. 11. Staleness with high system workload

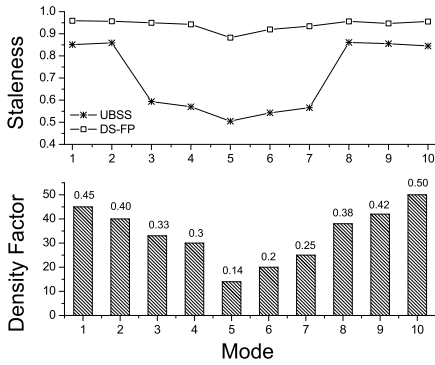


Fig. 12. Staleness with low system workload

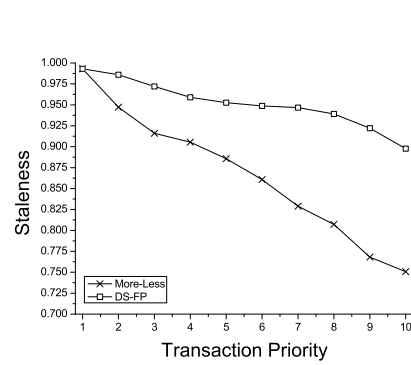


Fig. 13. Staleness vs. Transaction Priority

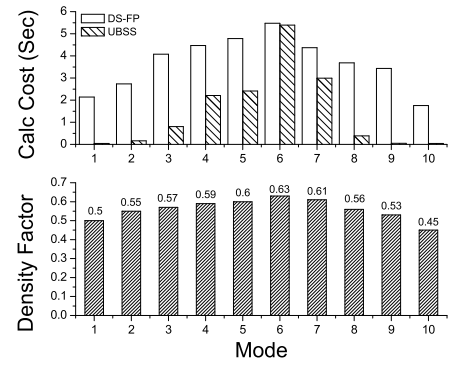


Fig. 14. Scheduling Overhead vs. CPU Utilization

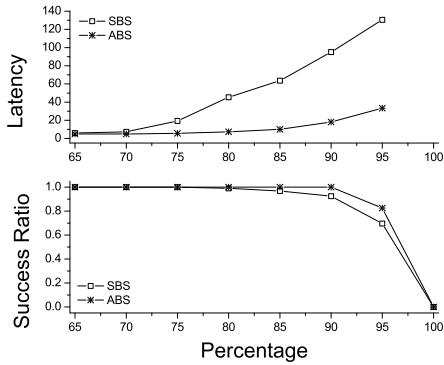


Fig. 15. Comparison of the Switch Latency

Figure 10 shows the comparison of the CPU utilization among three approaches. As mentioned in Section II, *DS-FP* can greatly reduce the CPU utilization compared with *ML* while still maintaining the temporal validity. This is verified in Figure 10 where the CPU utilization of *DS-FP* is consistently lower than *ML* and the difference reaches 15.7% when the density factor is 0.63 in mode 6. As observed in Figure 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 scheduling policy 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 control and achieve the maximized schedulability. Its CPU utilization will be close to *DS-FP* at that time.

Figure 11 and Figure 12 demonstrate *UBSS*'s improvement over *DS-FP* in maintaining higher data freshness with different system workloads. Suppose mode M_k contains the task set $\mathcal{T}_k = \{\tau_i\}_{i=1}^m$ and τ_i ($1 \leq i \leq m$) has released N_i update jobs by the end of M_k . We measure \bar{S}_k , the average data staleness of \mathcal{T}_k by summing up each update job ($J_{i,j}$)'s staleness at their finishing time ($f_{i,j}$) and divided by the total number of updates, i.e., $\bar{S}_k = \frac{\sum_{i=1}^m \sum_{j=0}^{N_i} S_t(f_{i,j})}{\sum_i N_i}$. Both Figure 11 and Figure 12 show that the average data staleness under *UBSS* is always lower than *DS-FP* which means the real-time data values under *UBSS* is persistently fresher. Figure 12 further shows that the improvement increases when the system workload decreases and it reaches around 40% when the density factor is 0.14 in mode 5. The main reason for this huge improvement is, when the system workload is low, the transaction set has a high possibility to be schedulable under *ML* or even *HH* where the deadline and period for each transaction are both assigned to be half of the validity length. This assignment greatly shortens the data staleness thus improve the data freshness. On the other hand, *DS-FP* always defers the release time of the update jobs as late as possible. This drives its data staleness close to 1 and both Figure 11 and Figure 12 show that the data staleness remains around 95% and is quite stable in the presence of the fluctuation of the system workload. Figure 13 helps us gain an insight into the comparison of data staleness between *ML* and *DS-FP* in mode 5. We have several observations in this figure. First, for each update transaction, the real-time data value under *ML* is consistently fresher than that of *DS-FP* and the transaction with lower priority has larger improvement; Second, transactions with lower priorities always have lower staleness no

matter which scheduling policy is employed.

Figure 14 compares the online scheduling overhead between *DS-FP* and *UBSS* in each mode. From the figure, we observe that with the same scheduling success ratio, *UBSS* can greatly reduce the online scheduling overhead especially when the density factor is low. For example, in mode 10, when the density factor is 0.45, the scheduling overhead of *DS-FP* is 1.75369 which is 41.2 times higher than that of *UBSS* (0.04257). This is because in those scenarios, *ML* will be in control most of the time and its online scheduling overhead is much lower than that of *DS-FP*.

C. Search-based Switch vs. Adjustment-based Switch

In this subsection, we compare the efficiency of the two algorithms for searching switch points from *DS-FP* to *ML* switch scenario. 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 transaction number ($m = 20$), 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 $\lceil m \times p\% \rceil$ transactions 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 2000 time units and compare the success ratio and switch latency between the two algorithms. We conduct 200 experiments for each point to get the average value.

In Figure 15, we observe that the switch success ratio under SBS and ABS are both decreasing when \mathcal{T}_{k+1} increases its size. This is because the more transactions \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 while not only searching passively, ABS always outperforms SBS in terms of the switch success ratio and the difference reaches 13% when the percentage p climbs to 95%. As \mathcal{T}_k is not schedulable under *ML*, when $p = 100\%$, \mathcal{T}_{k+1} is equal to \mathcal{T}_k and the success ratio for both of them drops to 0. Figure 15 also shows the comparison of the switch latency between SBS and ABS. We can see that ABS always has lower switch latency and the improvement keeps increasing when \mathcal{T}_{k+1} increases. This is because SBS restricts the switch candidates only at the beginning time slot of idle periods while ABS breaks this constraint. Through judicious schedule adjustment, ABS greatly increases the number of candidates for scheduling switch and potentially improve the switch latency.

In summary, it is demonstrated in our experimental results that ABS is more efficient than SBS in switch points searching during the mode change. It also shows that *UBSS* can maintain higher data freshness and significantly reduce the online scheduling overhead while still satisfying the temporal validity constraints.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we studied the problem how to maintain the temporal validity of real-time data in the presence of mode changes in flexible real-time systems. We proposed to use different scheduling policies in different modes and introduced two algorithms to search for proper switch points. We studied the properties of the switch point and provided some results on switching between two scheduling policies, *ML* and *DS-FP*. Extensive experiments are conducted to evaluate the algorithm performance and show that switch between different scheduling policies according to runtime

processor workload can significantly outperform a single fixed scheduling policy while only introduce limited switch overhead.

For future work, we will further investigate the properties of the scheduling switch for wider classes of scheduling policies. There are also several open questions for us to answer: 1) Suppose the MCR latency requirement is infinite, if the old and new task sets are schedulable under the old and new scheduling policies respectively, does there exist a proper switch point using SBS or ABS? 2) SBS and ABS are both synchronous algorithms and all the tasks in the new mode are released simultaneously. Can we design asynchronous algorithms? If so, how should scheduling switch be conducted?

REFERENCES

- [1] Krithi Ramamritham, "Real-time databases," *Distrib. Parallel Databases*, vol. 1, no. 2, pp. 199–226, 1993.
- [2] D. Locke, "Real-time databases: Real-world requirements," *Real-Time Database Systems: Issues and Applications*, 1997.
- [3] John A. Stankovic, Sang Hyuk Son, and Jorgen Hansson, "Misconceptions about real-time databases," *IEEE Computer*, vol. 32, 1999.
- [4] Lukasz Golab, Theodore Johnson, and Vladislav Shkapenyuk, "Scheduling updates in a real-time stream warehouse," *ICDE*, 2009.
- [5] Kam-Yiu Lam, Ming Xiong, Bi Yu Liang, and Yang Guo, "Statistical quality of service guarantee for temporal consistency of real-time data objects," in *RTSS*, 2004.
- [6] Krithi Ramamritham, "Where do time constraints come from and where do they go," *International Journal of Database Management*, 1996.
- [7] Xiaohui Song and J.W.S. Liu, "Maintaining temporal consistency: pessimistic vs. optimistic concurrency control," *TKDE*, 1995.
- [8] Ming Xiong, Rajendran Sivasankaran, John A. Stankovic, Krithi Ramamritham, and Don Towsley, "Scheduling transactions with temporal constraints: Exploiting data semantics," in *TKDE*, 1996.
- [9] Kyoung don Kang, Sang H. Son, John A. Stankovic, and Tarek F. Abdelzaher, "A qos-sensitive approach for timeliness and freshness guarantees in real-time databases," in *ECRTS*, 2002.
- [10] Ming Xiong and Krithi Ramamritham, "Deriving deadlines and periods for real-time update transactions," in *RTSS*, 1999.
- [11] Thomas Gustafsson and Jörgen Hansson, "Dynamic on-demand updating of data in real-time database systems," in *SAC*, 2004.
- [12] Ming Xiong, Song Han, and Kam-Yiu Lam, "A deferrable scheduling algorithm for real-time transactions maintaining data freshness," in *RTSS'05*.
- [13] Ming Xiong, Song Han, Kam-Yiu Lam, and Deji Chen, "Deferrable scheduling for maintaining real-time data freshness: Algorithms, analysis, and results," *IEEE Transactions on Computers*, 2008.
- [14] Jorge Real and Alfons Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-Time Systems*, 2004.
- [15] Shao-Juen Ho, Tei-Wei Kuo, and A. K. Mok, "Similarity-based load adjustment for real-time data-intensive applications," in *RTSS*, 1997.
- [16] A. Burns and R. Davis, "Choosing task periods to minimise system utilisation in time triggered systems," *Information Processing Letters*, 1996.
- [17] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, 1982.
- [18] Jorge Real, "Mode change protocols for real-time systems," *Ph.D. Thesis*.
- [19] Lui Sha, Ragunathan Rajkumar, John Lehoczky, and Krithi Ramamritham, "Mode change protocols for priority-driven preemptive scheduling," *Real-Time Systems*, 1988.
- [20] K. W. Tindell, A. Burns, and A. J. Wellings, "Mode changes in priority pre-emptively scheduled systems," in *RTSS*, 1992.
- [21] P. Pedro and A. Burns, "Schedulability analysis for mode changes in flexible real-time systems," *ECRTS*, 1998.
- [22] Rafik Henia and Rolf Ernst, "Scenario aware analysis for complex event models and distributed systems," *RTSS*, 2007.
- [23] N. Stoimenov, S. Perathoner, and L. Thiele, "Reliable mode changes in real-time systems with fixed priority or edf scheduling," *DATE*, 2009.
- [24] Paul Emberson and Iain Bate, "Minimising task migration and priority changes in mode transitions," *RTAS*, 2007.
- [25] C. L. Liu and James W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, 1973.
- [26] Song Han, Deji Chen, Ming Xiong, and A.K. Mok, "A schedulability analysis of deferrable scheduling using patterns," *ECRTS*, 2008.
- [27] Ming Xiong, Qiong Wang, and Krithi Ramamritham, "On earliest deadline first scheduling for temporal consistency maintenance," *Real-Time Systems*, 2008.