Real-Time Syst (2012) 48:387–429
DOI 10.1007/s11241-012-9150-4

Maintaining data temporal consistency in distributed real-time systems

Jiantao Wang · Song Han · Kam-Yiu Lam · Aloysius K. Mok

Published online: 14 April 2012

Abstract Previous works on maintaining temporal consistency of real-time data objects mainly focuses on real-time database systems in which the transmission delays (jitters) of update jobs are simply ignored. However, this assumption does not hold in distributed real-time systems where the jitters of the update jobs can be large and change unpredictably with time. In this paper, we examine the design problems when the More-Less (*ML*) approach (Xiong and Ramamritham in Proc. of the IEEE real-time systems symposium 1999; IEEE Trans Comput 53:567–583, 2004), known to be an efficient scheme for maintaining temporal consistency of real-time data objects, is applied in a distributed real-time system environment. We propose two new extensions based on *ML*, called Jitter-based More-Less (*JB-ML*) and Statistical Jitter-based More-Less (*SJB-ML*) to address the jitter problems. *JB-ML* assumes that in the system the jitter is a constant for each update task, and it provides a deterministic guarantee in temporal consistency of the real-time data objects. *SJB-ML* further relaxes this restriction and provides a statistical guarantee based on the given *QoS* requirements of the real-time data objects. We demonstrate through extensive simu-

The authors would like to thank Professor Sang Hyuk Son for his valuable suggestions on revising the paper and the discussion of the recent related works.

J. Wang · K.-Y. Lam (✉)
Dept. of Computer Science, City University of Hong Kong, Hong Kong, Hong Kong
e-mail: cskylam@cityu.edu.hk

J. Wang
e-mail: jiantwang2@student.cityu.edu.hk

S. Han · A.K. Mok
Dept. of Computer Science, University of Texas at Austin, Austin, USA

S. Han
e-mail: shan@cs.utexas.edu

A.K. Mok
e-mail: mok@cs.utexas.edu

lation experiments that both *JB-ML* and *SJB-ML* are effective approaches and they significantly outperform *ML* in terms of improving schedulability.

Keywords Real-time databases · Temporal consistency · Data freshness · Scheduling and jitters

1 Introduction

One of the important applications in distributed real-time systems is to continuously monitor the system environment status to respond to critical events. Example applications include coal mine monitoring (Li and Liu 2009), industrial process control (Song et al. 2008; Han et al. 2011) and road traffic control (Papageorgiou et al. 2005). In these systems, various sensors capture the current status of the physical entities in the system to generate sensor data periodically. These data are typical examples of real-time data objects which are associated with temporal validity to indicate that the sampled values are valid only for a time interval (Ramamritham 1993; Shanker et al. 2008; Ramamritham et al. 2004). A real-time data object is temporally consistent if its value "truly" reflects the current status of the corresponding physical entity. A new data value needs to be installed into a database to refresh the corresponding data object before the validity interval of the old value expires (Ramamritham 1993). Accessing invalid or stale data values can result in the inability to respond to environmental changes timely and even worse making wrong decisions (Xiong et al. 2010; Ramamritham et al. 2004; Golab et al. 2009; Amirijoo et al. 2006; Kang et al. 2004).

Although extensive research work has been devoted to the design of high-performance distributed real-time systems (Di Natale and Stankovic 1994; Lu et al. 2005; Shanker et al. 2008; Han et al. 2011), maintaining temporal consistency of real-time data objects in distributed real-time systems is still a challenging problem and has not received sufficient attention. On the other hand, the efficient methods proposed for maintaining temporal data validity in traditional real-time systems, such as Half-Half (*HH*) (Ho et al. 1997), More-Less (*ML*) (Xiong and Ramamritham 2004), EDF-based More-Less (ML_{EDF}) (Xiong et al. 2008b) and Deferrable Scheduling (*DS-FP*) (Xiong et al. 2008a), cannot be directly applied in distributed real-time systems since they are designed for systems where the transmission delays of update jobs are not considered. In distributed real-time systems, a newly generated data value is forwarded as an update job from a sensor or a task generator to a real-time controller through a network where the transmission delay (called jitter in the paper) cannot be simply ignored (Han et al. 2011). The controller maintains a database of real-time data objects to install data updates, process various types of control queries, and conduct data aggregation operations (Tan et al. 2009).

An extension of *ML* (Xiong and Ramamritham 2004) has considered the transmission delays of update jobs. However, it only assumes a simple transmission model and uses the maximum transmission delay of all the tasks to calculate the periods and deadlines for the tasks. In many distributed real-time systems, it is difficult to determine the maximum transmission delay of a set of update tasks, and the actual transmission delay of individual update job can be unbounded and not predictable due to transmission failures and the effect of dynamic network workloads.

As will be explained in a later section, variability in transmission delays makes the problem of maintaining temporal consistency in distributed real-time systems more difficult and the overhead for installing the updates more expensive. Even worse, delay variability can seriously affect the schedulability of the systems. To address these problems, in this paper, we first propose an extension of *ML*, called *Jitter-based More-Less* (*JB-ML*). *JB-ML* follows a similar approach as *ML* to assign the deadline and period for each update task to provide a deterministic guarantee in temporal consistency of real-time data objects. However, the assignment order is decided by the *Short Validity minus Jitter First* policy (*SV-JF*) (Zuhily and Burns 2007) instead of the *Shortest Validity First* (*SVF*) in *ML*. Based on *JB-ML*, with the introduction of the concept of statistical temporal consistency, we further propose another extension called *Statistical Jitter-based More-Less* (*SJB-ML*) to tolerate a certain degree of violation to the temporal validity constraints while still providing a statistical guarantee in temporal consistency. *SJB-ML* first determines the deadlines and periods of update tasks according to *reference jitter* values that are derived from the quality of services (*QoS*) required for the data objects, and then enforces an admission control test on each *late* job with the purpose to increase the number of late jobs completed before their deadlines and at the same time to prevent the late jobs affecting the scheduling of the remaining jobs.

The remainder of the paper is organized as follows. In Sect. 2, we briefly review previous research work on maintaining temporal consistency of real-time data objects. In Sect. 3, we describe the basic principles of *ML* and discuss its limitations in handling transmission delays. The details of the two proposed methods, *JB-ML* and *SJB-ML* are presented in Sects. 4 and 5 respectively. Section 6 presents our performance studies. We conclude the paper and discuss the future works in Sect. 7.

2 Related works

There has been extensive work in the literature on maintaining the temporal consistency of real-time data objects (Golab et al. 2009; Labrinidis and Roussopoulos 2001; Xiong and Ramamritham 1999, 2004; Xiong et al. 2005, 2006, 2010, 2008a; Han et al. 2009, 2008; Lam et al. 2004; Ahmed and Vrbsky 2000; Gustafsson and Hansson 2004; Xiang et al. 2008). Some of them are based on the periodic update model while the others use the sporadic update model in which no assumption is made on the generation and arrival patterns of update jobs. The methods based on the sporadic update model are mainly designed for soft real-time systems (Ramamritham et al. 2004; Ramamritham 1993; Amirijoo et al. 2006) and the main problem to be tackled is how to schedule update jobs in runtime to maximize the freshness of real-time data objects while minimizing their impacts to the execution of real-time transactions from applications. In Amirijoo et al. (2006), feedback control mechanisms were proposed to schedule update transactions together with user transactions from applications to deal with unpredictable workloads and transient overloading with the purpose to maintain the overall performance of the system within a specified QoS. It divides the workload into mandatory and optional, and tries to identify which ones should be scheduled using feedback control, based on QoS metrics. The mechanisms have been demonstrated to be effective for maintaining freshness in soft real-time systems such as the stock trading systems. Labrinidis and Roussopoulos (2001) studied the update scheduling problem for maintaining real-time information contained in dynamic web pages. Based on the popularity of the information, a quality-aware update scheduling algorithm was proposed. In Golab et al. (2009), based on the earliest deadline first (EDF) scheduling, a maximum benefit with look-ahead scheme was proposed to assign priorities to different updates to minimize the overall averaged staleness of real-time data objects. Qu and Labrinidis (2007) proposed a two-level scheduling algorithm called query update time sharing (QUTS). At the lower level, queries and updates have their own priority queues; at the higher level QUTS dynamically adjusts the share of CPU between the real-time queries and updates to maximize overall system profit. In Ahmed and Vrbsky (2000), Gustafsson and Hansson (2004), Xiang et al. (2008), various on-demand methods were proposed to reduce the number of unnecessary updates and thus minimize the CPU utilization for processing updates. The main weakness of these methods for hard real-time systems is that they do not consider the generations of update jobs and assume a given stream of updates whose arrival rate may be sporadic and uncontrollable. Thus, they cannot guarantee the temporal consistency by changing the periods for update job generations.

The periodic update model is mainly adopted for hard real-time systems where a guarantee in temporal consistency is required as accessing invalid data objects by user transactions or queries may generate incorrect results and the consequences could be similar to missing their deadlines and be catastrophic. Unlike the methods mentioned in above, the main problems to be studied are: (1) how to determine the period and deadline for each update task to maintain temporal consistency of each real-time data object; and (2) how to define a schedule such that the deadlines of all the update tasks can be guaranteed. One of the earliest proposals is the Half-Half (*HH*) method (Ho et al. 1997). In *HH*, both the period and relative deadline of an update task are set to be half of the validity interval of the data object to be updated. To further reduce the update workload, the More-Less (*ML*) approach (Xiong and Ramamritham 1999, 2004), which will be reviewed in Sect. 3, was proposed. *ML* applied the deadline monotonic scheduling (Leung and Whitehead 1982) to assign priorities to different update tasks. In contrast to *HH* and *ML*, the deferrable scheduling algorithm for fixed-priority transactions (*DS-FP*) (Xiong et al. 2005, 2008a) followed a sporadic update model. *DS-FP* exploited the semantics of temporal validity constraint of real-time data objects by judiciously deferring the sampling times of update jobs as late as possible, thus significantly outperformed *ML* in terms of reducing processor workload. To reduce the online scheduling overhead of *DS-FP*, (Xiong et al. 2006, 2010) proposed two extensions to produce a hyperperiod for *DS-FP* so that the schedule generated by repeating the hyperperiod infinitely will satisfy the temporal validity constraints of the real-time data objects.

The problem of temporal consistency maintenance using dynamic priority scheduling was examined in Xiong et al. (2008b). Based on a sufficient feasibility condition of EDF scheduling, it proposed ML_{EDF}, a linear time algorithm to solve the period and deadline assignment problem. To achieve an optimal solution in discrete systems, Xiong et al. (2008b) further proposed the OS_{EDF} algorithm by relaxing the deadline constraints to be arbitrary. A heuristic search-based algorithm, called HS_{EDF}, was also proposed to find the optimal solution more efficiently. In Li et al. (2011), an enhancement of *ML* is provided for multiprocessor systems.

Providing a complete guarantee in performance in real-time systems can be very expensive due to the existence of various unpredictable system factors. In some real-time systems, accessing some stale data values occasionally are tolerable. Amirijoo et al. (2006) proposed a framework for specification and management of QoS in real-time data management and transaction processing. Lam et al. (2004) proposed a family of Statistical More-Less (*SML*) algorithms to achieve the tradeoff between quality of services (QoS) of temporal consistency and the schedulability of systems. *SML* extended *ML* for the real-time database systems where the jobs from an update task may have high variation in computation time and having a certain degree of temporal inconsistency is acceptable. Han et al. (2009) investigated the cost of data freshness maintenance and online scheduling overhead in the presence of mode changes in real-time systems as a result of dynamic system workloads. It proposed to apply periodic scheduling policies when the imposed update workload was low to maintain high data freshness; when the update workload became high, it switched to more sophisticated algorithms to improve schedulability.

In this paper, we follow the periodic update model for update job generations and concentrate on the problems for the real-time systems which require a guarantee in temporal consistency. Unlike previous works, we try to resolve the problems due to jitters in update job transmissions in distributed real-time systems. Except Xiong and Ramamritham (2004), all the aforementioned works using the periodic update model assumed that the transmission delays are small enough to be ignorable for ease of analysis. However, this assumption is not practical in distributed real-time systems. Our work will relax this assumption and investigate the problem of maintaining real-time data temporal consistency with considerations on the variability in transmission delays. This relaxation makes the problem more challenging and the overhead for installing the updates more expensive. To our best knowledge, this is the first paper on using the periodic update model for maintaining temporal consistency of real-time data objects with considerations on variation of jitters in update job transmissions.

3 Backgrounds and preliminaries

In this section, we will first introduce the concept of real-time data temporal consistency. Then, we will present the well-known algorithm *ML* which is based on the periodic update model, and then discuss its limitations. For ease of reading, we summarize the symbols that will be used throughout the paper in Table 1.

3.1 Temporal consistency of real-time data

Real-time data objects are defined to represent the current status of physical entities in a real-time system. The act of measurement implements a function $\zeta_i \rightarrow X_i$ which maps the current status of the physical entity ζ_i to the value of a real-time data object

Table 1 Symbols and definitions

Symbol	Definition
X_i	Real-time data object i
Φ	The set of update tasks
τ_i	Update task for updating X_i
$J_{i,j}$	The $j+1$th job of task τ_i ($i=1,\dots,m$; $j=0,1,2,\dots$)
V_i	Validity interval of X_i
C_i	Worst-case execution time of task τ_i
D_i	Relative deadline of task τ_i
P_i	Period of task τ_i
R_i	Worst case response time of task τ_i
δ_i	Jitter of task τ_i
δ_{max}	The maximum jitter of all the tasks
Q_i^*	Statistical QoS requirement on temporal consistency of X_i
δ_i^*	Reference jitter of task τ_i to meet the QoS requirement Q_i^*
$\delta_{i,j}$	Jitter of $J_{i,j}$
$s_{i,j}$	Sampling time of $J_{i,j}$
$r_{i,j}$	Release time of $J_{i,j}$
$d_{i,j}$	Absolute deadline of $J_{i,j}$
$f_i(\delta)$	Probability density function of the jitters of τ_i
$F_i(\delta)$	Discrete cumulative density function of the jitters of τ_i

X_i in the database. Since the status of ζ_i can be highly dynamic, *e.g.*, the location of a fast moving object, the value of X_i may become invalid with the passage of time. To maintain the validity of a real-time data object X_i, update jobs are generated by a task periodically to capture the latest status of ζ_i. However, if the generation period for a job is short, the total update processing workload could be heavy. Thus, the main design issue is how to achieve a balance between maintaining the temporal validity of a set of real-time data objects and minimizing the total update workload. In database systems, this is called the data currency problem. Various best-effort approaches have been proposed in database systems research (Golab et al. 2009; Labrinidis and Roussopoulos 2001). However, different from conventional database systems, an essential concern of real-time systems is to provide a guarantee in data quality instead of maximizing the average currency of data accessed by applications.

The temporal consistency concept was proposed for defining the validity of real-time data objects (Ramamritham 1993). It is assumed that the maximum "rate of change" in the physical status of an entity can be estimated such that within any time interval of length not exceeding a given validity interval V_i, the status of the physical entity is considered by all applications to be adequately represented by any value of the data object within the time interval. We say that any two values of the data object are "similar" within the validity interval. A data object is temporally consistent if its value is determined within its validity interval from the last measurement.

Definition 1 A real-time data object X_i at time t is temporally consistent if, for its update job $J_{i,j}$, finished before t, the sampling time $s_{i,j}$ of $J_{i,j}$ plus the validity length V_i of the data object is not less than t, that is, $s_{i,j} + V_i \geq t$ (Ramamritham 1993).

It is obvious that the length of the validity interval for a real-time data object is application-dependent. Short intervals are for data objects corresponding to highly dynamic entities. The importance of temporal consistency is that if a data object accessed by a query is temporally consistent, the degree of currency of the data object is guaranteed. Another important property of this criterion for temporal validity is that it can be applied with the concept of data similarity (Kuo and Mok 1993) for resolving the problem in concurrency control between update jobs and user transactions. Under the concept of data similarity, the values from two successive update jobs for the same data object could be considered to be "similar" if the difference in their sampling times is smaller than the validity interval of the data object (Kuo and Mok 1993). If there are data conflicts amongst a user transaction and two update jobs, which are similar to each other, the serialization order amongst them can be adjusted to make the final schedule to be serializable (Bernstein et al. 1987; Lam and Kuo 2001). Therefore, it does not need to provide a concurrency control mechanism for resolving data conflicts between update jobs and user transactions (Kuo and Mok 1993; Kuo and Mok 1994).

3.2 Maintaining temporal consistency with Jitter

Data transmission delay is physically unavoidable for the jobs in distributed real-time systems. In this paper, the jitter of an update job $J_{i,j}$ is denoted by $\delta_{i,j}$, and $\delta_{i,j} = r_{i,j} - s_{i,j}$, where $r_{i,j}$ and $s_{i,j}$ are the release time and sampling time of $J_{i,j}$ respectively. The sampling time of $J_{i,j}$ is the time when it is created while the release time is the time when it arrives the controller and is ready for processing. Hereafter, transmission delay and jitter will be used interchangeably.

In distributed real-time systems, large jitters are possible and usually can not be ignored. In addition, they are always hard to predict and may change with the dynamic workload of the underlying network. Furthermore, for wireless networks, the jitter for transmitting a job is also highly affected by various environmental factors such as interferences, path loss and signal attenuation.

Since the jitters of different tasks and among the jobs of the same task could have different values, how to include them into the calculation of periods and deadlines for the update tasks is not a simple problem. Xiong and Ramamritham (2004), the extension of Xiong and Ramamritham (1999) uses the maximum jitter value among all the tasks in determining the deadlines and periods. This simple scheme, however, imposes serious limitations on system performance which will be elaborated in Sect. 3.3.

In this paper, we follow the WirelessHART mesh network model (Song et al. 2008; Han et al. 2011; Saifullah et al. 2011, 2010) as an example to discuss how to estimate the jitter values and the worst-case jitters. WirelessHART with standard number IEC 62591 is the first open wireless communication standard specifically designed

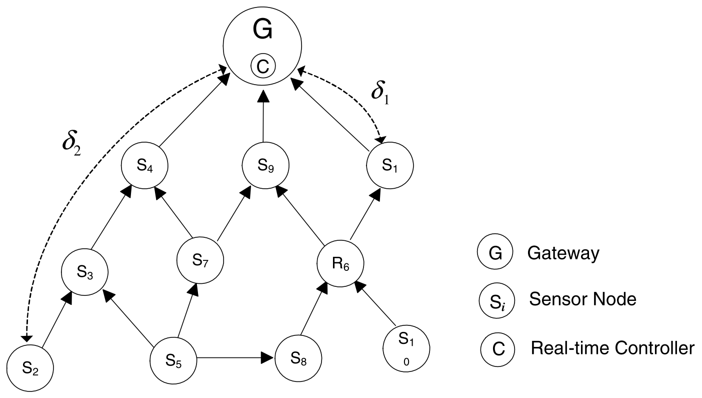


Fig. 1 A typical topology of WirelessHART networks

for process measurement and control applications. It is a real-time and reliable network communication protocol such that the performance of the network is more predictable. This assumption is valid for many industrial process control systems where the data link layers are mostly TDMA-based. Most of these data link layers have ACK and retry mechanisms. If the transmission is not successful after a given number of retries, the transmission will be taken as a failure. Thus, the communication delays for successful transmissions in such industrial wireless networks are bounded (Saifullah et al. 2010).

The topology of a typical WirelessHART network is depicted in Fig. 1. In the figure, S_i represents a sensor node which is attached to an industrial process and talks to the gateway G through a multi-hop wireless mesh. Each sensor node S_i has a running task τ_i. It generates update jobs periodically to relay the process data to the gateway based on a predefined scan period. The gateway G maintains a real-time controller C, which is responsible for scheduling the jobs released from all the update tasks. The jitter (transmission delay) of the jobs of task τ_i from S_i to the gateway is denoted by δ_i. If δ_i is small compared with its computation time (*e.g.*, S_1 is only one hop away from G and the communication is reliable without any additional delay or retry), then the jitter may be ignored and assumed to be zero. If δ_i cannot be ignored but its value is similar for the jobs from the same task (*e.g.*, S_2 is three hops away from G and the number of retries is assumed to be bounded for each hop.), then all the jobs of the same task may be assumed to have a constant jitter or bounded by a worst-case jitter. However, in practice, for the reliability purpose in industrial wireless mesh (Han et al. 2011), many applications require sensor nodes to have multiple routes to the gateway to tolerate link and node failures (*e.g.*, S_5 is configured with five different routes to G). For this reason, the jitter for individual job of an update task may change a lot with the variations of the network condition and the actual route and number of retries taken in the runtime. In these scenarios, the jitter δ_i is not a constant and the worst-case jitter could be unbounded.

As a summary, based on different assumptions on the jitter values, the problem of maintaining temporal consistency for real-time data objects can be classified into three cases:

- Case I: No jitter is considered. *i.e.*, $\delta_i = 0$ $(1 \le i \le m)$.
- Case II: The jitter of each task is a constant or bounded by a worst-case jitter value, and the jitters of different tasks are independent. *i.e.*, $\forall j, k, \delta_{i,j} = \delta_{i,k}$.
- Case III: The jitters of the jobs generated from the same task may vary. *i.e.*, $\exists j, k, \delta_{i,j} \neq \delta_{i,k}$, and the jitters are unpredictable and unbounded.

Case I was studied in *ML* (Xiong and Ramamritham 1999). The extension of *ML* (Xiong and Ramamritham 2004) proposed a simple scheme to deal with Case II by choosing the maximum jitter value among all the tasks to determine the deadlines and periods. It is the focus of this paper to provide a better solution for Case II and a solution for Case III.

3.3 The more-less method and its limitations

ML (Xiong and Ramamritham 1999, 2004) is an *off-line deterministic* method based on the periodic update model. *ML* is designed for systems where either: (1) separated computing resources are assigned for processing update jobs and applications such that the scheduling of update jobs will not be affected by the scheduling of application queries and transactions; or (2) the update jobs are assigned to higher priorities for execution comparing with other applications and user transactions. In the following discussions, we will concentrate on the second case. In addition, it is assumed that data synchronization delays due to concurrent accesses of shared real-time data objects between user transactions (*i.e.*, from applications) and update jobs are small and do not significantly affect the schedulability of the update jobs. An effective way to minimize the synchronization delay is to use the priority inheritance method (Sha et al. 1990) such that during the synchronization delay, the lower priority user transaction will be executed at the priority of the higher priority job which is waiting to access to the shared data object reading by the user transaction. The priority of the user transaction will resume to its own priority when it finishes reading the shared data object. Thus, the goal of *ML* is concentrated on the generation and scheduling of update tasks to provide a guarantee in temporal consistency of real-time data objects while minimizing the CPU utilization for job updating. To achieve this, in *ML*, there are three constraints to follow for each task τ_i $(1 \le i \le m)$:

- Validity constraint: the sum of the period P_i and relative deadline D_i of task τ_i is always no larger than V_i, *i.e.*, $P_i + D_i \le V_i$, as shown in Fig. 2. The absolute deadline $d_{i,j}$ of job $J_{i,j}$ is the sampling time $s_{i,j}$ plus D_i.
- Deadline constraint: the period P_i of task τ_i is assigned to be more than half of the validity length V_i, while D_i is less than half of V_i. For τ_i to be schedulable, D_i must be greater than or equal to C_i, the worst-case execution time of τ_i, plus the worst-case jitter among all the tasks, *i.e.*, $\delta_{max} + C_i \le D_i \le P_i$; and
- Schedulability constraint: for a given set of periodic tasks $\Phi = \{\tau_i\}(1 \le i \le m)$, the *deadline monotonic scheduling algorithm* (*DM*) (Leung and Whitehead 1982) is used to schedule them. Thus,

$$\delta_{max} + \sum_{j=1}^{i-1} n_{ij} \times C_j + C_i \le D_i$$

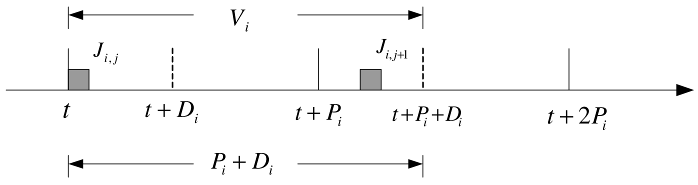


Fig. 2 An illustration of the more-less approach

where n_{ij} denotes the number of times task τ_j is executed before the first job of τ_i is completed, and δ_{max} is the worst-case jitter of all the tasks, *i.e.*, $\delta_{max} = max\{\delta_i\}(1 \le i \le m)$.

In calculating the deadlines and periods, *ML* follows the *Shortest Validity First* (*SVF*) policy to determine the assignment order of the tasks, *i.e.*, in the inverse order of validity length, and ties are resolved in favor of the task with smaller slack, *i.e.*, $V_i - C_i$ for τ_i.

Although *ML* has been shown to be effective in maintaining temporal data consistency, the following limitations restrict it from being directly applied to distributed real-time systems where jitters cannot be ignored:

- *ML* uses the worst-case jitter among all the tasks, δ_{max}, to determine the deadlines and periods. This unnecessarily increases the relative deadline, and thus reduces the period of each task. Therefore, the CPU utilization, $U = \sum_{i=1}^{m}(\frac{C_i}{P_i})$ will be increased accordingly.
- The increase in the worst-case response time of higher-priority tasks will lead to the increase in the relative deadlines of lower-priority tasks. This further increases the possibility of violations to the deadline constraint listed in above and can severely hurt the schedulability of the system.
- *ML* applies *DM* to schedule the update tasks. A new scheduling method, *deadline minus jitter monotonic* (*(D-J)-monotonic*) (Burns et al. 1995), was proven in Zuhily and Burns (2007) to outperform *DM* when jitters cannot be ignored.

4 Jitter-based more-less

In this section, we will introduce an enhancement of *ML*, called Jitter-based More-Less (*JB-ML*) for distributed real-time systems where different tasks have different jitter values (Case II in Sect. 3.2). Similar to *ML*, the objective of *JB-ML* is to determine the deadlines and periods for a set of update tasks such that the total CPU utilization is minimized while the temporal consistency of the set of real-time data objects is still guaranteed. Since *JB-ML* is an extension of *ML*, it follows the assumptions made in *ML*, *i.e.*, update tasks are executed at higher priorities and synchronization delays in accessing shared data objects between update jobs and user transactions from applications are small and do not affect the schedulability of the update tasks.

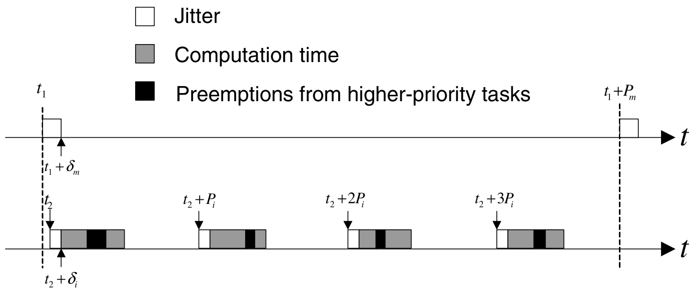


Fig. 3 Execution of τ_i between two consecutive jobs of τ_m

4.1 Theoretical preliminaries

Before introducing *JB-ML* formally, we first present several theoretical preliminaries. We denote the worst-case response time (*WCRT*) for any job of task τ_i by R_i, *i.e.*, the maximum response time of all released jobs, $J_{i,j}$s ($j \geq 0$). As in *ML*, the relative deadline D_i of τ_i is assigned to be its *WCRT* plus the jitter, *i.e.*, $D_i = R_i + \delta_i$. If a job of task τ_i with $WCRT = R_i$ can be finished before its deadline, all the other jobs of task τ_i can also be finished before their deadlines.

Lemma 1 *Consider a set of periodic tasks* $\Phi = \{\tau_i\}(1 \leq i \leq m)$ *with the sampling time of the first job* $s_{i,0}$, *jitter* δ_i *and the release time* $r_{i,k} = (s_{i,0} + \delta_i + k \times P_i)$ *for the kth job of* τ_i. *The WCRT of* τ_i, R_i, *occurs for the first job of* τ_i *when the first jobs of all the tasks are released simultaneously*, *i.e.*, $r_{i,0} = r_{j,0}, \forall i, j, 1 \leq i, j \leq m$.

Proof Let $\tau_1, \tau_2, \ldots, \tau_m$ denote a set of priority-ordered tasks with τ_m being the task with the lowest priority. Consider a particular job, $J_{m,l}(l \geq 0)$, of τ_m which is sampled at t_1 and released at $t_1 + \delta_m$. The next job, $J_{m,l+1}$, of τ_m to be sampled will be at $t_1 + P_m$. Suppose between t_1 and $t_1 + P_m$, the release times of task $\tau_i (i < m)$ are $t_2 + \delta_i, t_2 + \delta_i + P_i, t_2 + \delta_i + 2P_i, \ldots, t_2 + \delta_i + kP_i$, as shown in Fig. 3.

The shaded gray areas in Fig. 3 show the execution of the jobs of task τ_i in the time interval $[t_1, t_1 + P_m]$. These jobs may preempt the execution of $J_{m,l}$ unless $J_{m,l}$ is completed before $t_2 + \delta_i$, which is the earliest release time of τ_i's jobs in $[t_1, t_1 + P_m]$. The preemptions from τ_i will delay the completion time of $J_{m,l}$. Now, we will analyze how the response time of $J_{m,l}$ may change when the earliest release time $t_2 + \delta_i$ of task τ_i is varied. First, we assume that the completion time of $J_{m,l}$ is t_c and $t_2 + \delta_i = t_1 + \delta_m$. If $t_2 + \delta_i > t_1 + \delta_m$, the completion time of $J_{m,l}$ will be either unchanged or before t_c since $J_{m,l}$ may be executed in the time interval $[t_1 + \delta_m, t_2 + \delta_i]$. Thus, the response time of $J_{m,l}$ will be unchanged or smaller. If $t_2 + \delta_i < t_1 + \delta_m$, the beginning time of $J_{m,l}$ could be earlier as τ_i may be executed in time interval $[t_2 + \delta_i, t_1 + \delta_m]$. Thus, the completion time of $J_{m,l}$ will be equal to t_c or earlier. Therefore, the *WCRT* of task τ_m occurs when $t_2 + \delta_i = t_1 + \delta_m$, *i.e.*, the jobs from tasks τ_i and τ_m are released at the same time. Repeating the same argument for all $\tau_i, i = 1, 2, \ldots, m-1$, we conclude that the *WCRT* of task τ_m occurs when all the first jobs of higher-priority tasks (including task τ_m itself) are released simultaneously.

For any task τ_j $(1 \leq j < m)$, based on the above deduction, we can also conclude that the *WCRT* of τ_j occurs when all the first jobs of higher-priority tasks (including task τ_j itself) are released simultaneously. Therefore, the lemma is proved. □

Lemma 2 *For a set of periodic tasks* $\Phi = \{\tau_i\}$ $(1 \leq i \leq m)$ *with* $D_i \leq P_i$, *and having the same release time for their first jobs, the* $(D\text{-}J)$*-monotonic policy is an optimal fixed priority scheduling algorithm. This is true even if the task jitter is larger than zero. A task set is schedulable by* $(D\text{-}J)$*-monotonic if the first job of each task meets its deadline at the worst case as stated in Lemma* 1 (Zuhily and Burns 2007).

Lemma 1 implies that if jitters cannot be ignored, the *WCRT* happens when the first job of each task is released at the same time. Combining with Lemma 2, if the deadlines and periods of the task set can be derived by *JB-ML*, it will be schedulable.

Since $(D\text{-}J)$*-monotonic* is an optimal fixed priority scheduling algorithm for a set of tasks $\{\tau_i\}$ $(1 \leq i \leq m)$ with $D_i \leq P_i$ and jitters cannot be ignored, we adopt it in *JB-ML* for task scheduling.

4.2 *JB-ML* approach

4.2.1 Principles

Instead of using the maximum of the worst-case jitters among all the tasks as proposed in *ML*, in *JB-ML*, the worst-case jitter or simply call jitter δ_i of each task τ_i is included in the calculation of period and deadline for each task. The estimation of the worst-case jitter for a task could be based on the network configuration for update job transmissions. For example since we assume that the network is a TDMA-based mesh network such as the WirelessHART introduced in Fig. 1, the worst-case jitter for an end-to-end transmission between a sensor node and the controller can be estimated based on the number of hops between the sensor node and the controller and the maximum number of retries required for each transmission (Saifullah et al. 2011). Note that in real cases, the maximum number of retries could be unbounded due to various factors that may affect update job transmissions. This is the reason why we will propose a further enhancement of *JB-ML* called *SJB-ML* to resolve the problem in estimating the worst-case jitters.

In *JB-ML*, we assume that δ_i for each task is an independent constant (Case II in Sect. 3.2). Similar to *ML*, *JB-ML* is an *off-line deterministic* method. It determines the deadlines and periods for the tasks such that the following three constraints are satisfied:

- Validity constraint: $P_i + D_i \leq V_i$;
- Deadline constraint: $\delta_i + C_i \leq D_i \leq P_i$;
- Schedulability constraint: As the tasks are scheduled by $(D\text{-}J)$*-monotonic*, the following inequality constraint must hold:

$$\sum_{j=1}^{i-1} n_{ij} \times C_j + C_i \leq D_i - \delta_i \quad (1 \leq i \leq m)$$

where n_{ij} denotes the number of times task τ_j occurs before the first job of τ_i completes and $n_{1,j} = 0$.

Theorem 1 *Given a set of periodic tasks* $\Phi = \{\tau_i\}$ $(1 \le i \le m)$ *with deadlines and periods determined by* JB-ML, *the task set is schedulable and the temporal consistency of the data objects is guaranteed.*

Proof we need to prove that the three constraints of *JB-ML* can guarantee the schedulability of the tasks and temporal consistency of the data objects. Because of the schedulability constraint, the first job of each task can meet its deadline. The deadline and schedulability constraints combining with Lemma 1 ensure that the set of tasks can be scheduled by (*D-J*)-*monotonic* (Zuhily and Burns 2007). Since the set of tasks satisfies the validity constraint, the temporal consistency can also be guaranteed. Hence, the set of tasks is schedulable and the temporal consistency of the data objects is guaranteed. □

4.2.2 Assignment order policy in JB-ML

ML applies *SVF* to determine the assignment orders for calculating the deadline and period for each task in the task set. It was proved to be optimal such that the CPU utilization is minimized if jitters can be ignored and the following two restrictions are satisfied (Xiong and Ramamritham 2004):

$$\sum_{i=1}^{m} C_i \le \min\left(\frac{V_j}{2}\right) \quad (1 \le j \le m), \tag{1}$$

$$\begin{cases} V_1 \le V_2 \le \cdots \le V_m \\ \Delta C_{i+1,i} \le 2 \cdot \Delta V_{i+1,i} \quad (1 \le i \le m), \end{cases} \tag{2}$$

where $\Delta C_{i+1,i} = C_{i+1} - C_i$ and $\Delta V_{i+1,i} = V_{i+1} - V_i$. Since (*D-J*)-*monotonic* has been shown to give a better performance compared with *DM* when jitters are considered, we choose to use (*D-J*)-*monotonic* to assign priorities to schedule the released jobs at the real-time controller. Because the tasks may have very different jitter values, a task with a smaller validity interval may have a larger jitter comparing with a task with a larger validity interval. To make the assignment orders consistent with the priorities of the jobs from the tasks, in *JB-ML*, we use *SV-JF* instead of *SVF* for determining the assignment orders to calculate the deadlines and periods for the tasks, *i.e.*, in the inverse order of $V_i - \delta_i$, and ties are resolved by $V_i - \delta_i - C_i$. *SV-JF* guarantees a task that has a higher order in calculating the deadline will also has a smaller $D_i - \delta_i$, *i.e.*, higher priority in scheduling using (*D-J*)-*monotonic*. For example, considering tasks τ_i and τ_j with $V_i - \delta_i < V_j - \delta_j$, task τ_i has a higher order in calculating the deadline according to *SV-JF*. Note that the deadline of task τ_i is the sum of R_i and its worst-case jitter, *i.e.*, $D_i = R_i + \delta_i$. According to the *Schedulability Constraint*, we can get $R_i < R_j$, and furthermore, $D_i - \delta_i < D_j - \delta_j$, which means that τ_i has a higher priority in scheduling under the (*D-J*)-*monotonic* policy. Thus, the priorities of any tasks in calculating the deadline and in scheduling are consistent with each other.

Algorithm 1 Determine deadlines and periods in *JB-ML*

```
1: Input: A set of update tasks $\Phi=\{\tau_i\}_{i=1}^m$ ($m \geq 1$) with worst-case execution times $\{C_i\}_{i=1}^m$, validity
   intervals $\{V_i\}_{i=1}^m$, and jitters $\{\delta_i\}_{i=1}^m$ as well as a priority assignment order $\tau_1 \to \tau_2 \to \cdots \to \tau_m$.
2: Output: deadlines $\{D_i\}_{i=1}^m$ and periods $\{P_i\}_{i=1}^m$
3: /*Compute deadline and period for $\tau_0$*/
4: $D_0 = \delta_0 + C_0$;
5: $P_0 = V_0 - D_0$;
6: /*Compute $D_i$ and $P_i$ for $\tau_i$*/
7: for $i = 1$ to $m$ do
8:     $R_{i,0} = C_i$;
9:     repeat
10:        $D_i = R_{i,0}$;
11:        $R_{i,0} = C_i$;
12:        for $j = 1$ to $i - 1$ do
13:            /*calculate $R_{i,0}$ iteratively*/
14:            $R_{i,0} = R_{i,0} + \lceil \frac{D_i}{P_j} \rceil \times C_j$;
15:        end for
16:    until ($R_{i,0} == D_i$) or ($R_{i,0} + \delta_i > \frac{V_i}{2}$)
17:    if (($R_{i,0} + \delta_i$) > $\frac{V_i}{2}$) then
18:        return Abort;
19:    else
20:        $D_i = R_{i,0} + \delta_i$;
21:        $P_i = V_i - D_i$;
22:    end if
23: end for
24: return Successful;
```

4.2.3 JB-ML *algorithm*

Algorithm 1 shows how the deadlines and periods of the tasks are computed according to the orders determined from *SV-JF*. The algorithm is extended from the *ML* algorithm and the main difference is that the jitter δ_i for each task τ_i is used in calculation such that the three constraints of *JB-ML* can be satisfied.

4.3 Comparison of *JB-ML* with more-less

In this section, we will present the theoretical comparison between *ML* and *JB-ML*, and give the condition that *JB-ML* outperforms *ML* in minimizing CPU utilization. In the analysis, we assume that at least one jitter is not equal to the largest value δ_{max}, *i.e.*, $\exists i$, and $\delta_i \neq \delta_{max}$. Otherwise, if $\forall i$, $\delta_i = \delta_{max}$, the problem will be reduced to the one studied in Xiong and Ramamritham (2004).

Theorem 2 *Consider a set of periodic tasks in Case II. If* $\forall i, j$ $(1 \leq i < j \leq m)$ $V_i < V_j$ *and* $V_i - \delta_i < V_j - \delta_j$, *then the deadline assigned to any task using* ML *is larger than or equal to the deadline assigned to that task using* JB-ML.

Proof If $V_i < V_j$ and $V_i - \delta_i < V_j - \delta_j$ $(1 \leq i < j \leq m)$, the tasks will have the same priorities in both *ML* and *JB-ML*. We know that *ML* uses the largest jitter δ_{max} to calculate deadlines and periods, while *JB-ML* uses jitter δ_i for task τ_i. According to the schedulability constraints in *ML* and *JB-ML*, the deadlines of task τ_i derived from *ML* and *JB-ML* are, respectively:

$$D_i^{ml} = \delta_{\max} + \sum_{j=1}^{i-1}(n_{ij}^{ml} \times C_j) + C_i \quad (1 \le i \le m), \tag{3}$$

$$D_i^{jb} = \delta_i + \sum_{j=1}^{i-1}(n_{ij}^{jb} \times C_j) + C_i \quad (1 \le i \le m), \tag{4}$$

where n_{ij}^{ml} and n_{ij}^{jb} denote the number of times task τ_j occurs before the first job of τ_i is completed in *ML* and *JB-ML*, respectively. Mathematically, $n_{ij}^{ml} = \lceil \frac{D_i^{ml} - \delta_{max}}{P_j^{ml}} \rceil$ and $n_{ij}^{jb} = \lceil \frac{D_i^{jb} - \delta_i}{P_j^{jb}} \rceil$.

Assuming that τ_k $(1 \le k \le m)$ is the highest-priority task that satisfies $\delta_k < \delta_{max}$, *i.e.*, $\delta_i = \delta_{max}$ for $1 \le i \le k-1$. By reorganizing Eq. (3) and Eq. (4), we can get

$$R_i^{ml} = \sum_{j=1}^{i-1}\left(\left\lceil \frac{R_i^{ml}}{P_j^{ml}} \right\rceil \times C_j\right) + C_i \quad (1 \le i \le m), \tag{5}$$

$$R_i^{jb} = \sum_{j=1}^{i-1}\left(\left\lceil \frac{R_i^{jb}}{P_j^{jb}} \right\rceil \times C_j\right) + C_i \quad (1 \le i \le m), \tag{6}$$

where $R_i^{ml} = D_i^{ml} - \delta_{max}$ and $R_i^{jb} = D_i^{jb} - \delta_i$. R_i^{ml} (or R_i^{jb}) is the worst case response time of task τ_i in *ML* (or *JB-ML*). Note that in order to reduce the CPU utilization of the task set, the summation of the deadline and the period for each task should be equal to the validity interval in both *ML* and *JB-ML*, *i.e.*, $D_i^{ml} + P_i^{ml} = V_i$ and $D_i^{jb} + P_i^{jb} = V_i$ $(1 \le i \le m)$.

If $1 \le i \le k-1$, we have $\delta_i = \delta_{max}(1 \le i \le k-1)$. The calculation of the worst case response time for τ_i $(1 \le i \le k-1)$ in both *ML* and *JB-ML* is the same, thus we have $R_i^{ml} = R_i^{jb}$, $D_i^{ml} = D_i^{jb}$, and $P_i^{ml} = P_i^{jb}$ for all τ_i $(1 \le i \le k-1)$.

If $i = k$, we have $P_i^{ml} = P_i^{jb}$ for all τ_i $(1 \le i \le k-1)$ and the preemption from each task τ_i $(1 \le i \le k-1)$ on task τ_k is the same in *ML* and *JB-ML*. This implies that $R_k^{ml} = R_k^{jb}$, *i.e.*, $D_k^{ml} - \delta_{max} = D_k^{jb} - \delta_k$. Since $\delta_k < \delta_{max}$, we can get $D_k^{ml} > D_k^{jb}$ and $P_k^{ml} < P_k^{jb}$.

If $i = k+1$, because task τ_i $(1 \le i \le k-1)$ has the same period in *ML* and *JB-ML*, *i.e.*, $P_i^{ml} = P_i^{jb}$ $(1 \le i \le k-1)$, the preemption from these tasks on τ_{k+1} in *ML* will be equal to that in *JB-ML*. In addition, since task τ_k satisfies $P_k^{jb} > P_k^{ml}$, the preemption from task τ_k on task τ_{k+1} in *ML* is at least no smaller than that in *JB-ML*. Overall, the preemption from the previous k higher-priority tasks on task τ_{k+1} in *ML* is no smaller than that in *JB-ML*, *i.e.*, $R_{k+1}^{ml} \ge R_{k+1}^{jb}$. Thus we can derive that $D_{k+1}^{ml} - \delta_{max} \ge D_{k+1}^{jb} - \delta_{k+1}$ and $D_{k+1}^{ml} \ge D_{k+1}^{jb}$. The same analysis can be applied on the task τ_i $(k+1 < i \le m)$, and we can derive that $D_i^{ml} \ge D_i^{jb}$ and $P_i^{ml} \le P_i^{jb}$ for τ_i $(k+1 < i \le m)$.

Table 2 Parameters and results for Example 1

i	δ_i	C_i	V_i	*ML*		*JB-ML*	
				P_i	D_i	P_i	D_i
1	1	1	12	7	5	10	2
2	1	2	16	9	7	12	4
3	4	1	24	16	8	16	8

Table 3 The result of CPU utilization with increasing δ_3 for Example 1

Algorithm	δ_3					
	3	4	5	6	7	8
JB-ML	0.325	0.329	0.333	0.338	0.344	0.35
ML	0.384	0.423	0.483	N/A	N/A	N/A

This proves that if $\forall i, j$ $(1 \le i < j \le m)$ $V_i < V_j$ and $V_i - \delta_i < V_j - \delta_j$, $D_i^{ml} \ge D_i^{jb}$ holds for all τ_i $(1 \le i \le m)$. □

Corollary 1 *Consider a set of periodic tasks in Case II. If* $\forall i, j$ $(1 \le i < j \le m)$ $V_i < V_j$ *and* $V_i - \delta_i < V_j - \delta_j$, *and the task set is schedulable using both* ML *and* JB-ML, *then the CPU utilization obtained from* JB-ML *is lower than that from* ML.

Proof The total CPU utilization is:

$$U = \sum_{i=1}^{m} \left(\frac{C_i}{P_i} \right). \tag{7}$$

From Theorem 2, we know that $P_i^{ml} \le P_i^{jb}$ and at least one equality does not hold. According to Eq. (7), it is easy to see that the CPU utilization obtained from *JB-ML* is consistently lower than that obtained from *ML*. □

Example 1 Consider three tasks τ_1, τ_2, τ_3 with computation times 1, 2, 1, validity intervals 12, 16, 24, and jitters 1, 1, 4, respectively. Table 2 shows the relative deadlines and periods derived by *ML* and *JB-ML* respectively. As shown in Table 3, the CPU utilization of *ML* is 0.423, while that of *JB-ML* is only 0.329. Table 3 also shows the change in CPU utilization when the value for δ_3, the largest jitter among the three tasks, is varied. As shown in Table 3, the CPU utilization of *ML* is always higher than that of *JB-ML*. When δ_3 equals to or larger than 5, the task set cannot be scheduled in *ML*, but *JB-ML* still can schedule it.

5 Statistical Jitter-based more-less

In this section, taking *JB-ML* as a building block, we propose the Statistical Jitter-based More-Less approach (*SJB-ML*). In *SJB-ML*, all the assumptions of *ML* and *JB-ML* are followed except the assumption on the worst-case jitter. In *SJB-ML*, the jitter values for the tasks can be unbounded. Instead of guaranteeing a complete temporal consistency, *SJB-ML* aims at providing a good balance between statistical guarantee in temporal consistency for real-time data objects and the schedulability of update tasks.

5.1 Statistical guarantee in temporal consistency

Deterministic methods for maintaining real-time data temporal consistency, such as *ML* (Xiong and Ramamritham 1999) and its extension (Xiong and Ramamritham 2004), take the *WCRT* as the deadline for each task. However, if the jitters of the update jobs are not constant and even worse are unbounded, it will be difficult to have a good estimation on the *WCRT* and impossible to deterministically guarantee the temporal consistency of all the real-time data objects. Some of the update jobs may arrive too late to be scheduled and have to be dropped. On the other hand, in some real-time systems, it is generally tolerable to allow applications to read a certain percents of stale data, especially when the update cost for achieving deterministic temporal consistency is very high while the impact of missing some update deadlines is tolerable. For these systems, an approach for maintaining statistical temporal consistency of real-time data objects could be a better solution.

Definition 2 Q_i^* percents of statistical guarantee in temporal consistency for a real-time data object X_i is achieved if at least Q_i^* percents of jobs of τ_i can be completed before their deadlines over an arbitrarily long period of time.

Note that if the *QoS* is guaranteed at Q_i^*, the probability for n consecutive jobs of a task to miss their deadlines is $(1 - Q_i^*)^n$, which could be very small if n is large.

5.2 Principles of SJB-ML

In this section, we will discuss the principles and the details of *SJB-ML*. Unlike *ML* and *JB-ML*, *SJB-ML* is designed for distributed real-time systems where the jitters of the update jobs from the same task may vary (Case III in Sect. 3.2). Since the temporal consistency of real-time data objects and schedulability of update tasks can be seriously affected by the variation of jitters, the main challenge here is how to minimize the impacts of the varying jitters such that the required *QoS* of real-time data temporal consistency can be met and maximized. *SJB-ML* addresses this problem by adopting a mixed scheduling approach: (1) it defines an *off-line* schedule similar to *ML* and *JB-ML*; and (2) it maintains an *on-line* scheduler to adaptively handle the varying jitters of update jobs. The three main steps of *SJB-ML* as shown in Algorithm 2 are summarized in below, and their details will be elaborated in the next sub-section.

1. Construct the jitter distribution for each task τ_i based on the jitter values collected in a given time window;
2. Determine the *reference jitter* δ_i^* according to the required *QoS* Q_i^* for each data object X_i and then calculate the relative deadline D_i and period P_i for each task τ_i based on the *reference jitter* δ_i^*; and

Algorithm 2 Framework of the *SJB-ML* algorithm

```
1: Input: A set of update tasks $\Phi = \{\tau_i\}_{i=1}^m$ $(m \geq 1)$ with worst-case execution times
   $\{C_i\}_{i=1}^m$, validity intervals $\{V_i\}_{i=1}^m$, required QoS $\{Q_i^*\}_{i=1}^m$, and jitter distribution $\{F_i\}_{i=1}^m$
   as well as a priority assignment order $\tau_1 \to \tau_2 \to \cdots \to \tau_m$.
2: Output: A schedule of the task set $T$.
3: /*$Q$, $Q_e$ and $Q_l$ store the normal, early and late jobs.*/
4: $Q = Q_e = Q_l = null$;
5: /* Step 1: Construct the distribution of the jitters for each task $\tau_i$. */
6: Construct($F_i$) $(1 \leq i \leq m)$;
7: /* Step 2: Derive the reference jitter $\{\delta_i^*\}_{i=1}^m$ and calculate deadlines and periods
     according to Algorithm 1. */
8: Calc_Reference_Jitter($\{Q_i^*\}_{i=1}^m$, $\{f_i(\delta)\}_{i=1}^m$);
9: JB-ML($\{V_i\}_{i=1}^m$, $\{C_i\}_{i=1}^m$, $\{\delta_i^*\}_{i=1}^m$);
10: /* Step 3: Admit and schedule the jobs online. */
11: Schedule($Q_e$, $Q$, $Q_l$);
```

3. In runtime, execute an *admission control test* (*ACT*) on each late job such that a late job will be admitted only if it can be finished before its deadline without affecting the required *QoS* guaranteed to any data objects; and then schedule all the jobs using the (*D*-*J*)-*monotonic* scheduling.

Note that in runtime, it monitors the *QoS* of temporal consistency provided by each update task over a chosen long period of operation time. If the temporal consistency of a data object over the period of time is lower than or dropped close to the required *QoS*, a new distribution for the jitters will be constructed (Step 1), and then the reference jitter, the relative deadline and period of each task will also be recalculated (Step 2). As will be illustrated in the experiment section, the admission control test can effectively maximize the total *QoS* provided to be significantly higher than the required *QoS*, recalculation of jitter distribution is only required when there is a great change in jitter distribution continuously for a long period of time.

5.3 Algorithm details

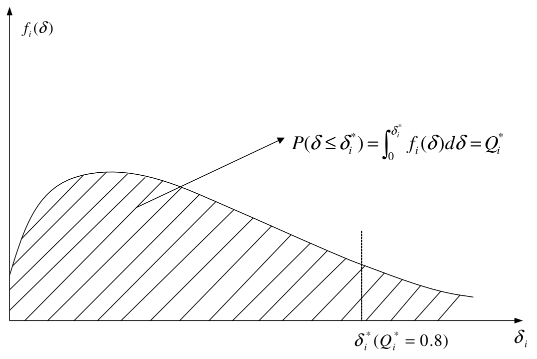
5.3.1 Determination of the reference Jitters, periods and deadlines

Figure 4(a) shows a probability density function (PDF) $f_i(\delta)$ to illustrate the jitter distribution for the jobs of task τ_i. If Q_i^* percents of the jobs from task τ_i have to be guaranteed to be completed before their deadlines, we determine a *jitter* δ_i^*, called the reference jitter, from Eq. (8) such that the percentage of the jobs whose jitters are no larger than reference jitter δ_i^* is Q_i^* percents.

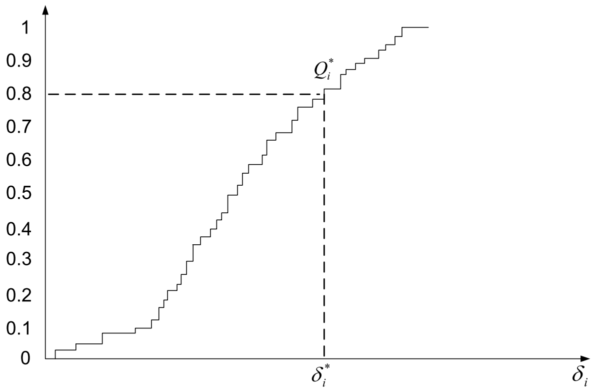
$$Q_i^* = \int_0^{\delta_i^*} f_i(\delta) d(\delta) \tag{8}$$

In practice it may not be easy to model the jitter and derive its probability density function (PDF) for each update task. Therefore, in *SJB-ML*, we construct the empirical cumulative distribution function (ECDF) $\hat{F}_i(\delta)$ instead of PDF $f_i(\delta)$ based on the

Fig. 4 Illustration of Jitter distribution



(a) PDF of the jitters of task τ_i



(b) Empirical CDF of the jitters of task τ_i

jitter values collected in the latest time window W with a fixed length of time L. The time window length L is chosen to be a large value to make sure that the constructed ECDF is stable and the required QoS can be satisfied. This will also help reduce the number of unnecessary ECDF computation and update. Then, according to the generalized inverse distribution function of $\hat{F}_i$ which will be described later, we derive the reference jitter δ_i^* satisfying the *QoS* requirement Q_i^* of task τ_i, as shown in Fig. 4(b). If all the jobs of task τ_i with jitters no larger than the reference jitter δ_i^* are completed before their deadlines, at least Q_i^* percents of τ_i's jobs can be guaranteed, and the required QoS is also guaranteed. Hereafter, we call the jobs whose jitters equal to δ_i^* as normal jobs. The jobs whose jitters are smaller than δ_i^* are called early jobs, and the jobs whose jitters are larger than δ_i^* are called late jobs.

Note that in the system initialization stage (when the system time t is 0), if we do not have the jitter values of the tasks for building the ECDF $\hat{F}_i(\delta)$ for each task, we may use the method proposed in *JB-ML* to obtain the maximum (worst-case) jitter δ_i for each update task τ_i and assign it to be the reference jitter δ_i^*. When the system time t is larger than L, we will construct the empirical CDF $\hat{F}_i(\delta)$ for δ_i based on the collected jitter values in the time window $(t - L, t]$. If we define for $y \in [0, 1]$, the generalized inverse distribution function of $\hat{F}_i$ is $\hat{F}_i^{-1}(y) = \inf_{x\in\mathbb{R}}\{\hat{F}_i(x) \ge y\}$, then the reference jitter δ_i^* will be calculated as the Q_i^* percentile, *i.e.*, $\hat{F}_i^{-1}(Q_i^*)$:

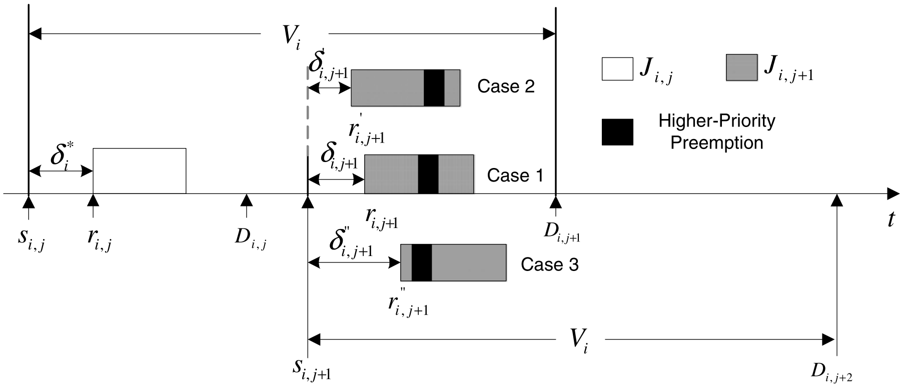


Fig. 5 An example of the early job, late job and normal job

$$\delta_i^* = \begin{cases} \delta_i & \text{if } t < L \\ \hat{F}_i^{-1}(Q_i^*) & \text{if } t \geq L \end{cases}$$

To reduce the computation and communication overheads, instead of updating $\hat{F}_i(\delta)$ and δ_i^* continuously, the controller records the accepted update jobs, which are completed before their deadlines, on the fly for each task. This value indicates the degree of temporal consistency provided for each task in the current operation period L. Once the percentage of accepted update jobs is lower than or dropped to be close to the given *QoS* Q_i^*, it will reconstruct the empirical CDF for δ_i by sorting the collected jitter values in the latest time window of size L in the ascending order and derive the new reference jitter as the Q_i^* percentile.

Line 9 in Algorithm 2 determines the period and relative deadline for each update task using the reference jitters obtained above. The calculation in *SJB-ML* follows the constraints defined in *JB-ML*, but it uses the reference jitter δ_i^* to replace δ_i for each task.

5.3.2 *Online scheduling of released jobs*

In Step 3 (Line 11), when an update job arrives (or called released), the scheduler at the real-time controller first decides if it is a normal job, a late job or an early job. Since the jitters of the jobs from a task may change with time, as shown in Fig. 5, the released jobs may be affected by or affect the execution of other released jobs. Case 1 shows that $J_{i,j+1}$ is a normal job, *i.e.*, $\delta_{i,j+1} = \delta_i^*$. According to *JB-ML*, $J_{i,j+1}$ can be finished before its deadline if the total preemption time from higher priority jobs is no more than that incurred in *JB-ML*. Case 2 shows an early job $J_{i,j+1}$, which is released at $r'_{i,j+1}$. It may preempt the lower-priority jobs that are to be executed during the time interval $[r'_{i,j+1}, r_{i,j+1}]$ and make them not schedulable. At the same time, $J_{i,j+1}$ may also be preempted by other higher-priority jobs and cannot be finished before its deadline. Case 3 shows a late job $J_{i,j+1}$. Its late arrival may also affect the execution of other lower-priority jobs.

Recall that we need to provide a statistical guarantee in temporal consistency for each data object X_i to be at least Q_i^*. To achieve this, we design a scheduler, in here,

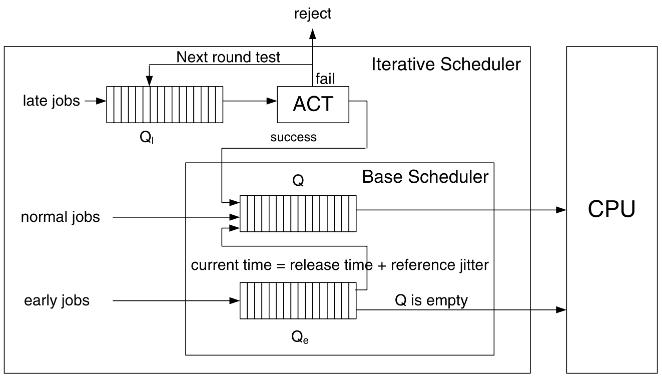


Fig. 6 The *base scheduler* and the *iterative scheduler* in *SJB-ML*

called the *Base Scheduler* which admits all normal jobs and early jobs but rejects all late jobs. If all the early and normal jobs of τ_i are admitted and can be completed before their deadlines, the required Q_i^* will be achieved. However, as explained in above example, some of the early and normal jobs may miss their deadlines due to preemptions from higher-priority early jobs. To resolve this problem, the *Base Scheduler* maintains two queues, Q_e for early jobs and Q for normal jobs. In Q_e and Q, the jobs are sorted according to their priorities. Each time, the scheduler selects a job from Q with the highest priority for execution. If Q is empty, the job in Q_e with the highest priority will be chosen. An early job will be moved from Q_e into Q if the current time t is equal to its release time as determined by its reference jitter, *i.e.*, $t = s_{i,j} + \delta_i^*$. Once it is moved into Q, its priority will be compared with the currently executing job. If its priority is higher, it will preempt the currently executing job for execution.

Although the *Base Scheduler* can achieve the required *QoS* of temporal consistency of a real-time data object as all its normal and early jobs are finished timely, some of the late jobs which may be able to be completed before their deadlines are rejected unnecessarily. Furthermore, postponing the release time of an early job $J_{i,j+1}$ from $r'_{i,j+1}$ to $r_{i,j+1}$ defers its finish time. In addition, the processor will be idle during the time interval $[r'_{i,j+1}, r_{i,j+1}]$ if only early jobs are waiting to be executed while Q is empty. To complete more jobs especially the late jobs to further improve the real-time data temporal consistency above the required *QoS*, as shown in Fig. 6, the scheduler maintains one more queue, Q_l for the late jobs, and adds an *Admission Control Test* (*ACT*) module to test the schedulability of the late jobs maintained in Q_l. If a late job can be scheduled, it will be admitted instead of being rejected immediately. We call the enhanced scheduler as the *Iterative Scheduler*.

In the *Iterative Scheduler*, the *ACT* performs a schedulability test on the late jobs in Q_l when a job switch is happened, i.e., the schedulability test is executed when either (1) the currently executing job is preempted by a higher-priority job; or (2) a new job arrives when the processor is idle. To minimize the testing cost, the test is performed on the jobs in Q_l with priority higher than the job that preempts the processor when a job switch is happened. These candidate jobs take the schedulability test according to their priorities determined by (*D-J*)-*monotonic*. Once any one of them passes the test, it will be moved into Q and the test will be stopped. After the schedulability test, the jobs in Q and Q_e are scheduled in the same way as in the *Base Scheduler*.

The schedulability test algorithm is summarized in Algorithm 3. For a late job $J_{i,j}$, if the total remaining execution time from the higher-priority jobs that have been released and maintained in Q, Q_e and Q_l plus C_i is larger than $d_{i,j} - r_{i,j}$, $J_{i,j}$ will fail to pass the schedulability test and will be rejected, *i.e.*, deleted from the queue Q_l (Line 5). Otherwise, the algorithm estimates the total preemption time from all higher-priority jobs that will be released before the deadline of $J_{i,j}$ with the assumption that all these jobs have a jitter equal to their reference jitters. If the estimated total preemption time plus C_i is smaller than the deadline of $J_{i,j}$, *i.e.*, the late job passes the test and $J_{i,j}$ will be moved into Q. Then, the late job will be treated as a normal job (Line 21). If the estimated total preemption time plus C_i exceeds the deadline of $J_{i,j}$, $J_{i,j}$ will fail to pass the schedulability test and has to wait for the next round of testing (as shown in Fig. 6) hoping that some of the higher-priority jobs may be rejected later (Line 19) and then it can be admitted.

It can be observed in Algorithm 3 that the total cost for performing the schedulability test is low and this will also be illustrated in the performance studies to be reported in the next section. The cost of the schedulability test for each late job consists of two parts: the cost for each schedulability test and the number of schedulability tests performed for each late job. Note that no schedulability test is necessary for early and normal jobs. As shown in Algorithm 3, the complexity of each schedulability test is $V_i/2 \cdot O(i)$ because the complexity of the *for loop* is $O(i)$ and the *repeat loop* may at most execute $d_{i,j} - r_{i,j}$ times, where $i (1 \le i \le m)$ is the task index and $d_{i,j} - r_{i,j} \le D_i \le V_i/2$. For each late job, at most D_i number of tests will be taken. The worst case happens if the late job takes a schedulability test at each time point from its release time to its absolute deadline. Normally, as shown in Table 5 in Sect. 6.3, the number of tests for each late job is low, around 3, on average.

6 Performance evaluation

In this section, we will report the important results obtained from our performance studies on *JB-ML* and *SJB-ML* as compared with *ML*. Section 6.1 describes the simulation model and experiment settings. Sections 6.2 and 6.3 report the performance of *JB-ML* and *SJB-ML* respectively. Section 6.4 reports the impacts of *ML*, *JB-ML* and *SJB-ML* on the performance of user transactions for a system using the update first method (Xiong and Ramamritham 2004) to co-schedule the update jobs and user transactions.

6.1 Simulation model and parameters

The simulation model was developed based on the network model introduced in Fig. 1. It consisted of a real-time controller and a fixed set of sensor nodes. The sensor nodes were connected to the controller through a wireless mesh network similar to the one shown in Fig. 1. Each sensor node ran an update task. It generated

Algorithm 3 Schedulability test by *ACT* on $J_{i,j}$

```
1: Input: job $J_{i,j}$ with $\delta_{i,j} > \delta_i^*$.
2: Output: if job $J_{i,j}$ passes the test, it is admitted for execution; otherwise, it will be rejected or waits
   for the next round of test.
3: $sum = \sum_{h=1}^{i-1} J_{h,k}.c_{remain}$; /* calculate the total remaining execution time of all released higher pri-
   ority jobs $J_{h,k}$s at $t$. */
4: if ($r_{i,j} + sum + C_i >= d_{i,j}$) then
5:     reject $J_{i,j}$ and return FALSE; /*The preemptions from the released higher-priority jobs make $J_{i,j}$
       to miss its deadline.*/
6: else
7:     $R_{i,j} = sum + C_i$;
8:     /*The repeat and while loop calculates the preemptions from the released and future (non-released)
       higher-priority jobs.*/
9:     repeat
10:        $R'_{i,j} = R_{i,j}$; /*Temporally keep $R_{i,j}$ for comparison.*/
11:        $R_{i,j} = sum + C_i$; /* Initialize $R_{i,j}$ for recompute it.*/
12:        /* Compute the preemptions from higher-priority tasks. */
13:        for each higher priority job $J_{h,k}$ ($h < i$) to be admitted in $[t, t + R_{i,j}]$ do
14:            $R_{i,j} = R_{i,j} + (\lceil \frac{R'_{i,j}}{P_h} \rceil - 1) \times C_h$;
15:        end for
16:        $d'_{i,j} = r_{i,j} + R_{i,j}$; /*$d'_{i,j}$ is the deadline of $J_{i,j}$ considering the preemptions from all the re-
           leased and future higher-priority jobs.*/
17:    until ($d'_{i,j} > d_{i,j}$ or $R'_{i,j} == R_{i,j}$)
18:    if $d'_{i,j} > d_{i,j}$ then
19:        wait for the next round of test and return FALSE;
20:    else
21:        accept $J_{i,j}$ and return TRUE;
22:    end if
23: end if
```

update jobs following a period determined by the adopted method, *i.e.*, *ML*, *JB-ML* or *SJB-ML*. The generated update jobs were forwarded to the controller to update the corresponding real-time data objects maintained at the controller. It was assumed that the controller was a single processor system and the real-time data objects were maintained in the main memory. Since each update job accessed to one data object, no concurrency control was required for resolving the data conflicts between update jobs. The length of each experiment run was at least 1,000 times of the longest validity interval amongst all the tasks so that the main results generated were stable, *i.e.*, further increase in the simulation length did not make any significant changes on the experiment results.

Table 4 summarizes the set of parameters and the baseline settings for the experiments. Note that in the experiments, we did not aim at studying the performance for a particular distributed real-time system. Instead, we wanted to illustrate the performance characteristics and schedulability of our proposed methods in achieving the performance goal of guaranteeing the temporal consistency of real-time data objects under different system settings. Thus, a wide range of workloads was used to test their performance by changing the number of update tasks as well as other important parameters such as the worst-case execution time of an update task and the distribution of the jitters. Furthermore, the values for the parameters were chosen according

Table 4 Experiment settings

Symbol	Definition	Default Value
	System Parameters	
N_Φ	Number of tasks	[50, 300]
V_i (ms)	Validity interval of τ_i	[1000, 15000]
C_i (ms)	Worst-case execution time of τ_i	[5, 15]
Q_i^*	Mean QoS requirement	0.75
	JB-ML Parameters	
μ_i (ms)	Mean value of jitter for all tasks	260
σ_i (ms)	Standard deviation of jitter	80
	SJB-ML Parameters	
μ_i' (ms)	Mean value of jitters for τ_i	260
σ_i' (ms)	Standard deviation of jitters for τ_i	80
θ_i	Scale of the Γ distribution of τ_i	0.5
k_i	Shape of the Γ distribution of τ_i	$\frac{E(\delta_{i,j})}{0.5}$
	User Transactions	
	Slack factor of the user transactions	8
	No. of user transactions arrived per second	4
	No. of objects accessed by a user transaction	[5, 15]
	CPU Time for each data access	5

to the settings used in Xiong and Ramamritham (1999, 2004) so that our results could be compared with the findings from them. The settings in Xiong and Ramamritham (2004) were defined according to the study of an air traffic control system in Locke (1997). As shown in Table 4, in the baseline setting, the number of real-time data objects (as well as the number of update tasks) in the system was varied from 50 to 300 and their validity intervals were chosen to be uniformly distributed between 1,000 and 15,000 ms. The worst-case execution time of an update task was uniformly distributed between 5 and 15 ms. In *JB-ML*, the worst-case jitter for the tasks was assumed to follow the normal distribution $\mathcal{N}(1/\mu_i, \sigma_i)$ such that the jitter values could be distributed in a relatively wide range and easily varied by changing the variance of the distribution. In *SJB-ML*, the jitter $\delta_{i,j}$ of a job from task τ_i followed the Gamma distribution $\Gamma(k_i, \theta_i)$ following the findings in Li and Mason (2007), and the expected value of the jitters $E(\delta_{i,j})$, for task τ_i equal to $k_i * \theta_i$. We fixed θ_i at 0.5 for all the tasks and $E(\delta_{i,j})$ followed the normal distribution $\mathcal{N}(1/\mu_i', \sigma_i')$.

In the third set of experiments, we included user transactions in the system to study the impacts of update tasks scheduling on the performance of the user transactions. They accessed to real-time data objects to get new values generated from update jobs. The number of data objects to be accessed by a user transaction was uniformly distributed between 5 to 15, and each data access took 5 ms. The user transactions were scheduled using the earliest deadline first (EDF) scheduling and the update first method (Xiong and Ramamritham 2004) was used to co-schedule the user transactions and update jobs. The slack factor determined the slack time for a user transaction before its deadline and was fixed at 8. The following equation was used to determine the deadline of a user transaction τ_i:

$$Deadline(\tau_i) = AT(\tau_i) + (ET(\tau_i) \times SlackFactor) \tag{9}$$

where $AT(\tau_i)$ and $ET(\tau_i)$ are the arrival time and total execution time of τ_i, respectively. Note that in the simulation model, similar to Xiong and Ramamritham (2004), we did not consider the concurrency control between update jobs and user transactions as the conflicts were resolved by the application of the concept of data similarity (Kuo and Mok 1993). It was also assumed that the synchronization delay in accessing shared data object was small and could be ignored as all the data objects were resided in the main memory and the problem of priority inversion was resolved by the use of the priority inheritance method (Sha et al. 1990).

6.2 Experiment 1: Comparison between *JB-ML* and *ML*

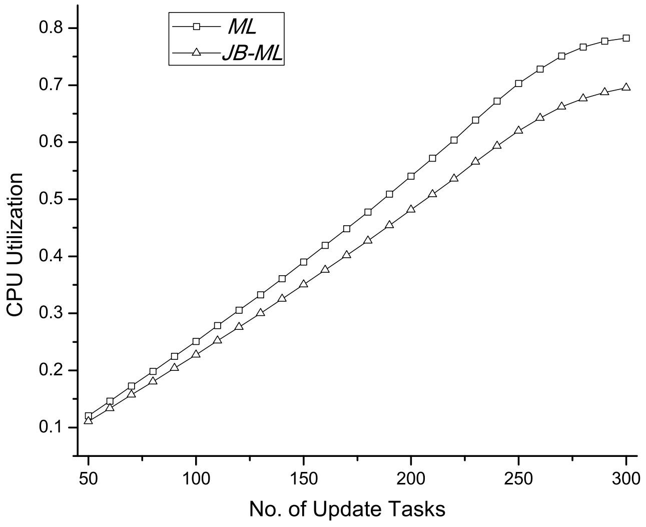
In this set of experiments, we compared the performance of *JB-ML* with *ML* in terms of the CPU utilization and schedulability under different update workloads by changing the number of update tasks in the system. In addition, we also evaluated the impacts of different jitter distributions and worst-case execution times of an update task on their performance.

Figure 7(a) shows the CPU utilization of *JB-ML* and *ML* where the jitters of the tasks follows the normal distribution $\mathcal{N}(260, 80)$ and the worst-case execution time of an update task C_i is uniformly distributed from 5 to 15. Consistent with our theoretical analysis shown in Sect. IV-C, as shown in Fig. 7(a), the CPU utilization of *JB-ML* is consistently lower than that of *ML*, and the improvement is greater when the number of tasks is larger. The difference in CPU utilization reaches 10% when the number of tasks is more than 150. The lower CPU utilization of *JB-ML* is due to longer update generation periods of the tasks. Consistent with the results shown in Fig. 7(a), when the worst-case execution time of an update task is reduced to 2 to 8, the CPU utilization of *JB-ML* is still consistent lower than that of *ML* as shown in Fig. 7(b).

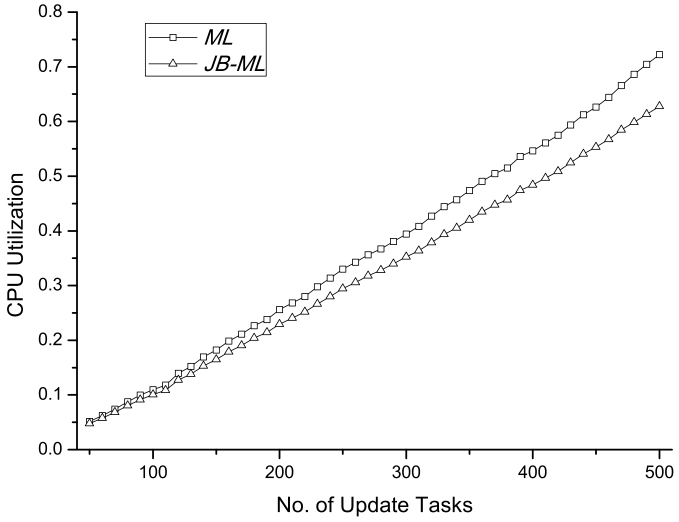
Figure 8 shows the comparison in CPU utilization between *JB-ML* and *ML* when different jitter distributions and worst-case execution times are used. We use ΔU (Eq. (10)) to denote the relative difference in the CPU utilization between *ML* and *JB-ML*.

$$\Delta U = \frac{U^{ml} - U^{jbml}}{U^{ml}} \tag{10}$$

Consistent with the results shown in Fig. 7, as shown in Fig. 8, the utilization of *JB-ML* is always lower than that of *ML* for different distributions of jitters and worst-case execution times of an update task. It can also be observed in Fig. 8 that if the distribution of the jitters is more dispersive, the improvement on the CPU utilization of *JB-ML* is more significant. For instance, ΔU is about 1% when the distribution is $\mathcal{N}(260, 10)$ while ΔU is about 10% when the jitter distribution is $\mathcal{N}(260, 80)$. Note that if σ_i is larger, the jitters of the jobs from the same task set is more dispersive.



(a) $C_i \in [5, 15]$

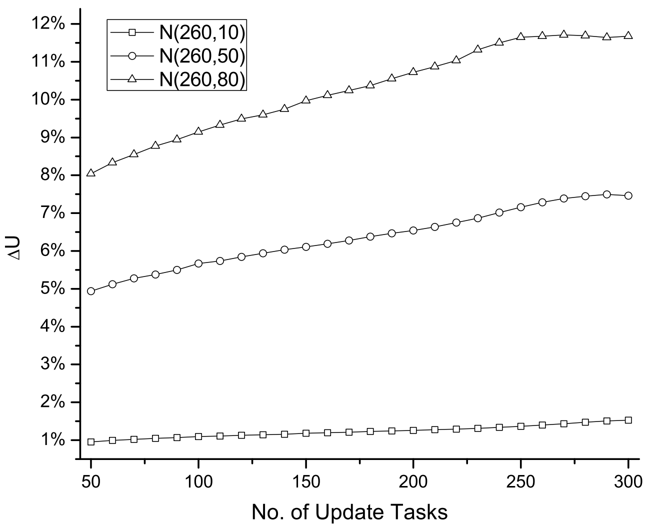


(b) $C_i \in [2, 8]$

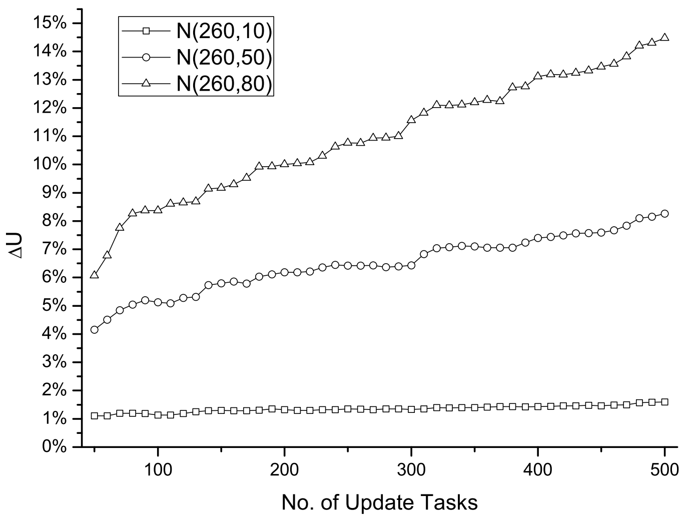
Fig. 7 CPU utilization vs. no. of update tasks

The results show that *JB-ML* can effectively reduce the total CPU utilization for processing update jobs to provide a deterministic guarantee in the temporal consistency of the real-time data objects.

To show the better schedulability of *JB-ML*, we randomly generated 600 task sets for each task set of size from 50 to 300 tasks. As shown in Fig. 9, when the number of tasks is less than 230, both *JB-ML* and *ML* can schedule all the randomly generated 600 task sets. However, when the number of tasks is more than 250, the performance



(a) $C_i \in [5, 15]$



(b) $C_i \in \{2, 8\}$

Fig. 8 ΔU vs. no. of update tasks

of *ML* drops dramatically in particular if the jitter distribution is more dispersive. For example, when the jitter distribution is $\mathcal{N}(260, 80)$ and number of tasks equals to 300, *ML* can only schedule about 16% of the generated task sets while the percentage of successfully scheduled task sets by *JB-ML* remains close to 75%. The results show that *JB-ML* is less affected by the variation of the jitters compared with *ML*.

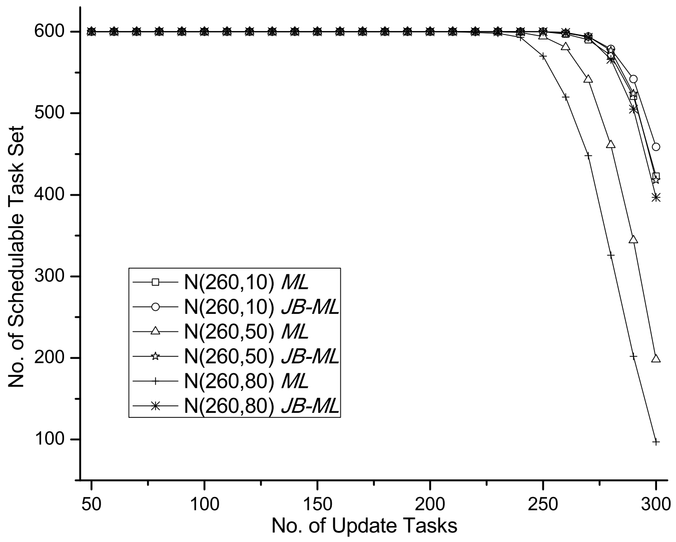


Fig. 9 Schedulability vs. no. of update tasks

6.3 Experiment 2: Performance of *SJB-ML*

In this set of experiments, we evaluated the performance of *SJB-ML*. If the jitter of a job is unbounded, it is very difficult to define a good schedule using *ML* and *JB-ML*. Even if a schedule can be obtained, the performance will be very poor as the maximum jitter of a task could be very large and consequently the update generation periods have to be very short. Thus, in the experiments, we concentrated on studying the effectiveness of *SJB-ML* using the *Iterative Scheduler* as compared with the *Base Scheduler* in achieving the required *QoS*. The performance of the *Base Scheduler* was used as a *baseline* to demonstrate the effectiveness of the *admission control test (ACT)* in the *Iterative Scheduler* in improving the overall performance of the system by admitting more late jobs and at the same time to complete them before their deadlines.

The parameter settings for Experiment 2 were similar to those used in Experiment 1, except that the jitter values of the update jobs from each task τ_i were varied following the Gamma distribution $\Gamma_i(k, \theta)$. The default required *QoS* Q_i^* for each data object was set to be 0.75, and two more *QoS*, 0.85 and 0.95, were used for comparison. The main metrics used in the experiments were the accept ratio (Eq. (11)), the consistency ratio (Eq. (12)) and the largest number of continuous rejected jobs, which was a measure of the distribution of the temporal consistency of the data objects. Note that if the temporal inconsistency of two data objects are similar, a smaller value of the largest number of continuous rejected jobs in general indicates that the temporal consistency of the data object is distributed relatively more even. In the experiments, we used N_i^- and N_i^+ to denote the number of rejected late jobs and the number of accepted jobs of τ_i, respectively. We used t_i to denote the total time duration that the temporal consistency of data object X_i was maintained. T was the length of the simulation time. Other performance measures were the CPU utilization and number of schedulability tests performed.

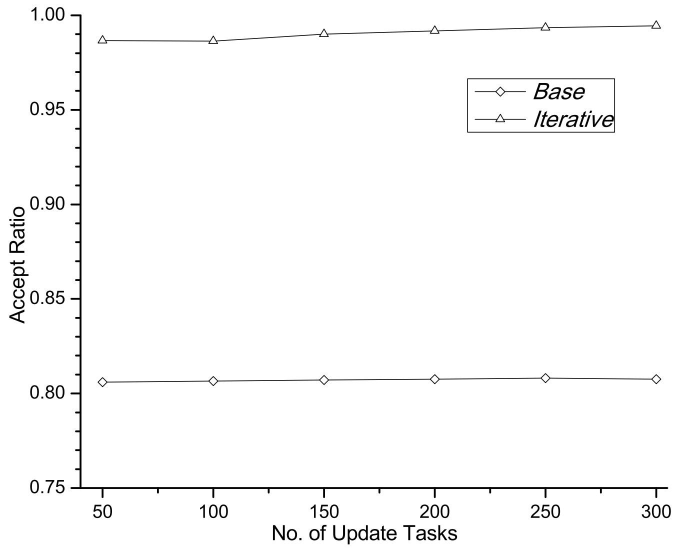


Fig. 10 Accept ratio vs. no. of update tasks

$$accept\ ratio = \frac{1}{m}\sum_{i=1}^{m}\frac{N_i^{+}}{N_i^{-}+N_i^{+}} \tag{11}$$

$$consistency\ ratio = \frac{1}{m}\sum_{i=1}^{m}\frac{t_i}{T} \tag{12}$$

Figures 10, 11 and 12 show the accept ratios, the consistency ratios and the CPU utilization for the *Iterative Scheduler* and the *Base Scheduler*, respectively, where the required *QoS* is 0.75. In Fig. 10, it can be observed that the *Iterative Scheduler* always gives a higher accept ratio than that of the *Base Scheduler*. Similar results on the consistency ratio are shown in Fig. 11. The higher accept ratio and consistency ratio of the *Iterative Scheduler* indicate that the admission control test (ACT) is effective in admitting more late jobs to provide a higher degree in temporal consistency of the data objects. An interesting observation from Figs. 10 and 11 is that the consistency ratio is slightly higher than the corresponding accept ratio especially for the *Base Scheduler*. This is because although we use the *WCRT* of a task to determine the periods for generating update jobs, in most cases, a job can be completed well before its *WCRT*. Therefore, if a job misses its deadline and the corresponding data object becomes invalid, the invalid period of the data object in most cases will be shorter than the *WCRT* of the job. Another interesting observation from Fig. 11 is that the consistency ratio of the *Base Scheduler* increases with number of tasks. It is because the consistency ratio of the *Base Scheduler* increases with decreasing task priority as shown in Fig. 13. For example, as shown in Fig. 13, when the number of tasks is 300, the consistency ratios of the higher priority tasks (*i.e.*, those tasks with ID < 50) are around 0.82 while that of the lower priority tasks (ie task ID > 250) are around 0.87. The higher consistency ratios of the lower priority tasks are due to the

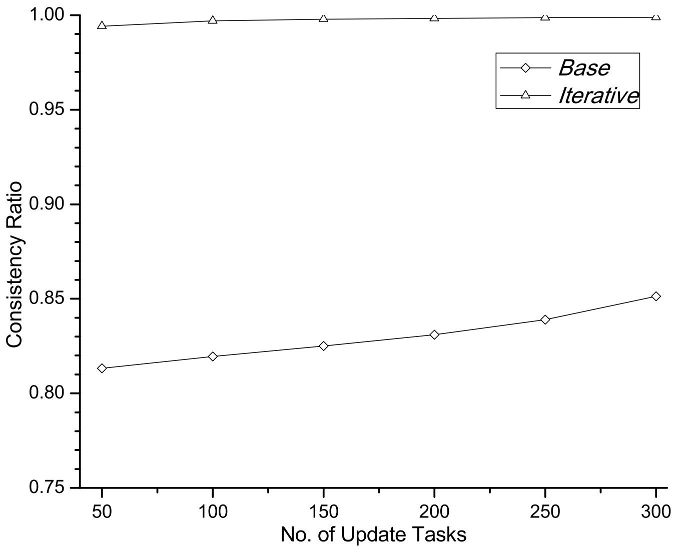


Fig. 11 Consistency ratio vs. no. of update tasks

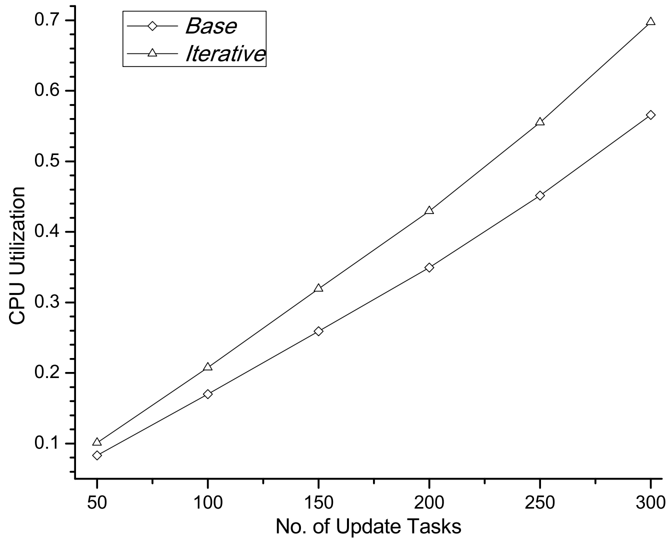


Fig. 12 CPU utilization vs. no. of update tasks

reservation of more slack times for them in the calculation using the *WCRT*. The higher consistency ratios of the lower priority tasks make the consistency ratio of the whole system increase gradually with the number of tasks. As a result of accepting more jobs for processing, the *Iterative Scheduler* has a higher CPU utilization than the *Base Scheduler* as shown in Fig. 12, and it can achieve a higher degree in temporal consistency of the data objects.

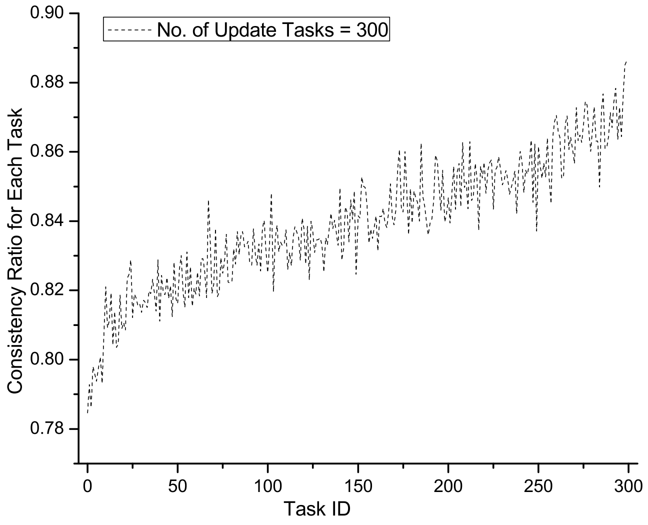


Fig. 13 Consistency ratio vs. task ID

Although the *Iterative Scheduler* gives a better performance compared with the *Base Scheduler*, it is important to examine the amount of overhead for accepting more jobs. We used the number of schedulability tests performed as a measure of the overhead. In Table 5, we summarize the number of schedulability tests per released job, the number of schedulability tests per late job and the number of schedulability tests per accepted late job for different number of tasks and required *QoS*. As shown in the table, their values increase with number of tasks and decrease with required *QoS*. The number of tests per late job has a slightly lower value compared with the number of tests per accepted late job as most of the late jobs are accepted for processing in the schedulability tests. When the number of tasks is 50 and the required *QoS* is 0.95, both the number of tests per late job and the number of tests per accepted late job are smaller than 1.5. Although the values raise to around 6 when the number of tasks is 300 and the required *QoS* is 0.75, the improvement in temporal consistency of the data objects are significant making them close to 100% as shown in Fig. 11. Since the number of tests per job (including early, normal and late jobs) is smaller than 0.5 for most cases, the admission control test in the *Iterative Scheduler* can effectively accept more late jobs to achieve a higher degree in temporal consistency of the data objects without incurring a heavy testing cost.

Figure 14 shows the average number of schedulability tests performed for a late job of each task. It is interesting to see that the average number of tests decreases with task priority. For example, as shown in Fig. 14, when the required *QoS* is 0.75, the numbers of schedulability tests of the higher priority tasks (*i.e.*, those tasks with ID $<$ 50) are between 1 to 2 while that of the lower priority tasks (*i.e.* task ID $>$ 250) are between 4 to 6.5. The average number of tests for each task decreases with required *QoS*. For example, when the required *QoS* is 0.95, the average number of tests for different tasks are mostly lower than 1.5 even for the higher priority tasks. It is because if the required *QoS* is higher, the number of late jobs will be smaller.

Table 5 No. of schedulability tests

QoS	0.75			0.85			0.95		
No. of update tasks	50	150	300	50	150	300	50	150	300
No. of tests per job	0.43	0.8	1.21	0.198	0.414	0.83	0.06	0.132	0.19
No. of tests per late job	1.43	3.24	5.85	1.32	2.76	4.54	1.21	2.35	3.6
No. of tests per acc. late job	1.69	3.63	6.33	1.56	3.10	4.94	1.35	2.55	3.83

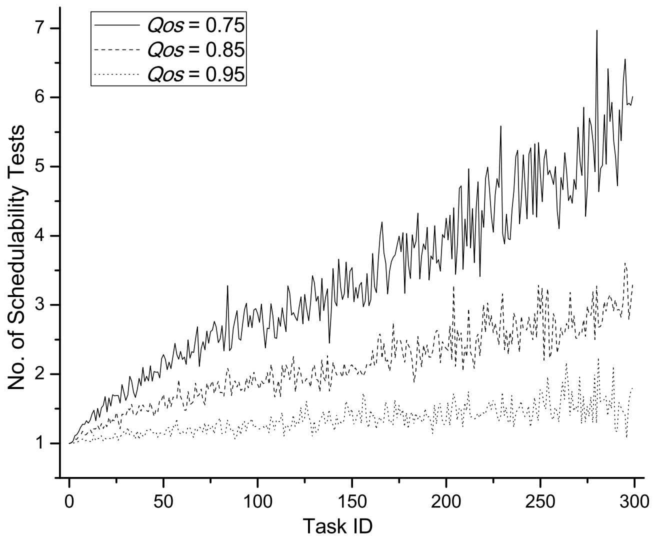


Fig. 14 No. of schedulability tests vs. task ID

Figure 15 shows the comparison of the largest number of continuous rejected jobs between the *Base Scheduler* and the *Iterative Scheduler*. It can be observed that in the *Iterative Scheduler*, except τ_1, the largest numbers of continuous rejected jobs for all the tasks are equal to or close to zero. However, in the *Base Scheduler*, the largest numbers of continuous rejected jobs vary from 4.5 for the higher priority tasks to about 3 for the lower priority tasks. For τ_1, the largest number of continuous rejected jobs is the same in both *Base Scheduler* and *Iterative Scheduler*. It is because all late jobs of τ_1 are rejected as the amount of slack times allocation to them are zero in calculating their *WCRT*. The results show that with the *Iterative Scheduler* even if the temporal consistency cannot be achieved, the degree of temporal inconsistency will be more evenly distributed over the time.

Figures 16 and 17 show the consistency ratios and the CPU utilization of the *Iterative Scheduler* with different *QoS* requirements when the number of tasks is varied. Consistent with the results shown in Fig. 11, the consistency ratios are always maintained to a value close to 1 for different required *QoS* even when the number of tasks is large. Similarly, the CPU utilization is about the same for different *QoS* as the

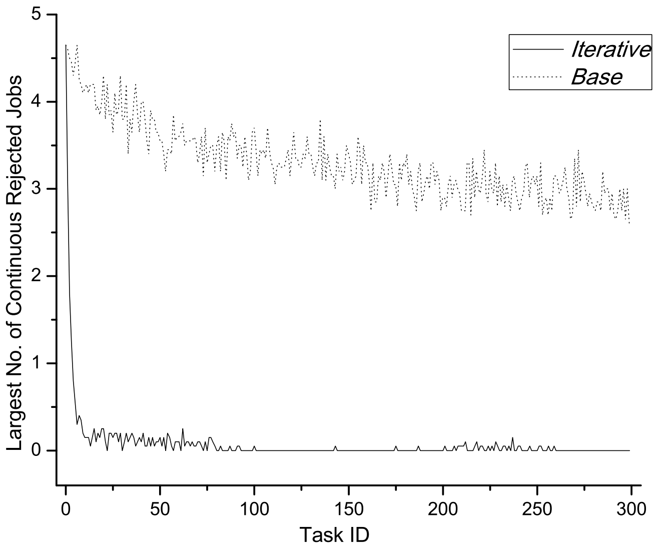


Fig. 15 Largest no. of continuous rejected jobs vs. task ID

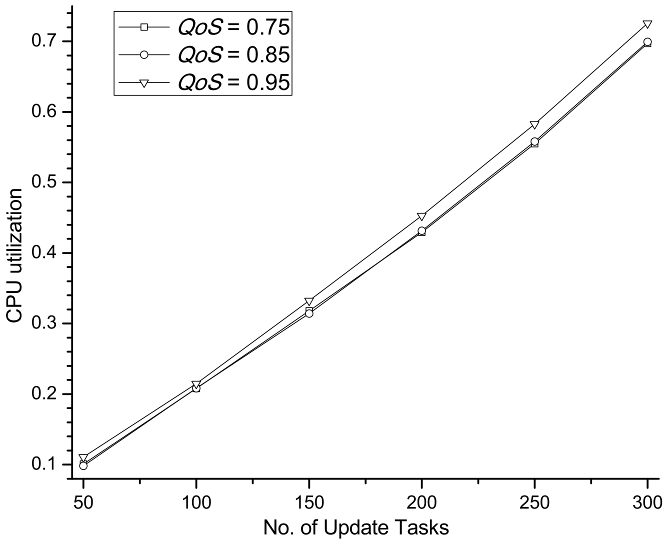


Fig. 16 CPU utilization vs. no. of update tasks

number of accepted jobs and achieved temporal consistency of the data objects are similar for different required *QoS*.

As a summary, *SJB-ML* can effectively handle the situations that the jitters of update jobs are varied in run-time. The experiment results presented in this section support the believe that *SJB-ML* can provide a statistical guarantee in temporal consistency of real-time data objects based on given *QoS* requirements. The *Iterative*

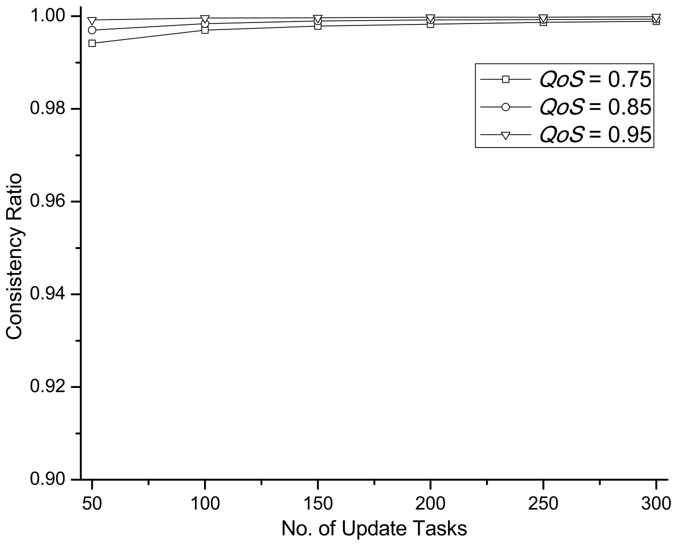


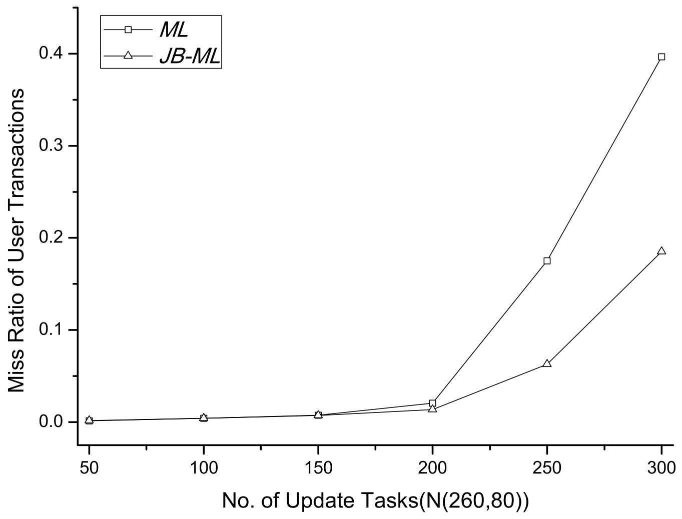
Fig. 17 Consistency ratio vs. no. of update tasks (*iterative scheduler*)

Scheduler can effectiveness accept more jobs to achieve a better temporal consistency of the real-time data objects compared with the *Base Scheduler* and the overhead for the schedulability tests depending on the number of tasks and required *QoS*.

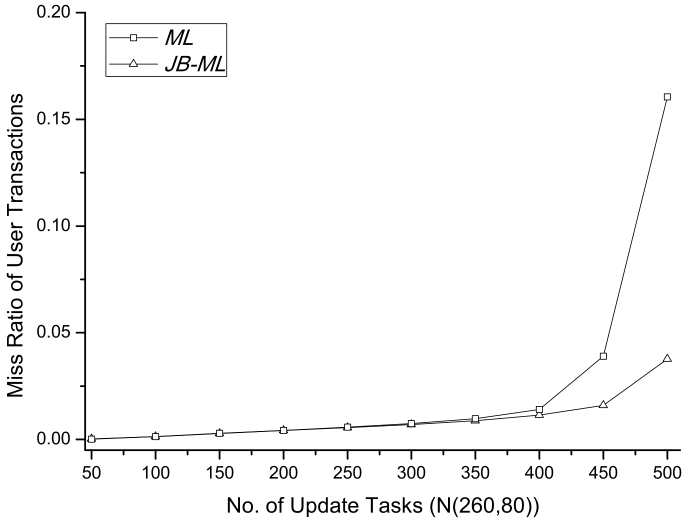
6.4 Experiment 3: Co-scheduling with user transactions

In this set of experiments, we studied the impacts of *ML*, *JB-ML* and *SJB-ML* on the performance of user transactions for a system using the update first method to co-schedule update jobs and user transactions. The user transactions were associated with deadline constraints on their completion times. Meeting the deadlines and providing temporally consistent data objects for their executions were two important requirements in supporting effective real-time applications. Thus, in this set of experiments, we measured the miss ratio of the user transactions as an indicator of the impacts of *ML*, *JB-ML* and *SJB-ML*, on the performance of the user transactions. It was defined as the total number of user transactions whose deadlines were missed over the total number of user transactions processed.

As shown in Fig. 18(a), consistent with our expectation, the miss ratios of both *ML* and *JB-ML* increase with the number of update tasks. It is because increasing the number of update tasks increases the update workload, as shown in Fig. 19(a). Since the update jobs are executed at higher priorities compared with the user transactions, the probability of missing their deadlines becomes higher when the update workload is heavier. Comparing Figs. 18(b) with (a), we can see that if the worst-case execution time C_i is larger, the update workload will be heavier. Therefore, the miss ratio of user transactions will be higher. Another important observation from Fig. 18 is that the miss ratio of *ML* is always



(a) $C_i \in [5, 15]$

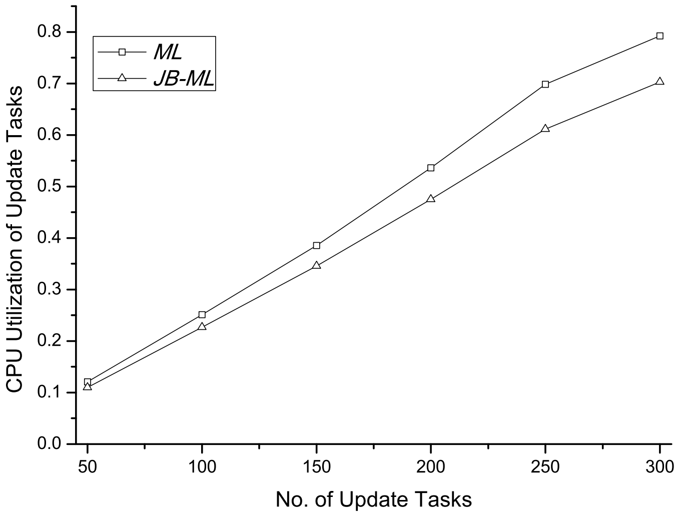


(b) $C_i \in \{2, 8\}$

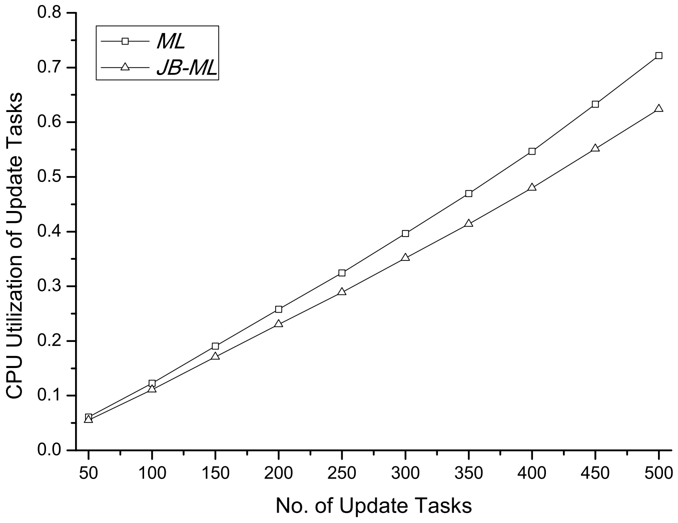
Fig. 18 Miss ratio vs. no. of update tasks

higher than that of *JB-ML*. For example, as shown in Fig. 18(a), when the number of update tasks is 300, 40 percents of user transactions miss the deadlines in *ML* while the miss ratio of user transactions is around 20 percents in *JB-ML*. It is because the CPU utilization resulted from *ML* is higher than that from *JB-ML*.

Figures 20(a) and (b) show the miss ratios of the *Base Scheduler* and the *Iterative Scheduler* when C_i is uniformly distributed in [5, 15] and [2, 8], respectively.



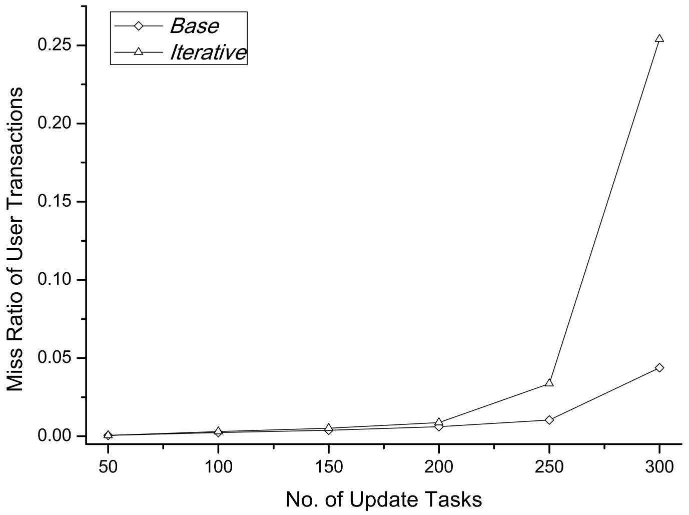
(a) $C_i \in [5, 15]$



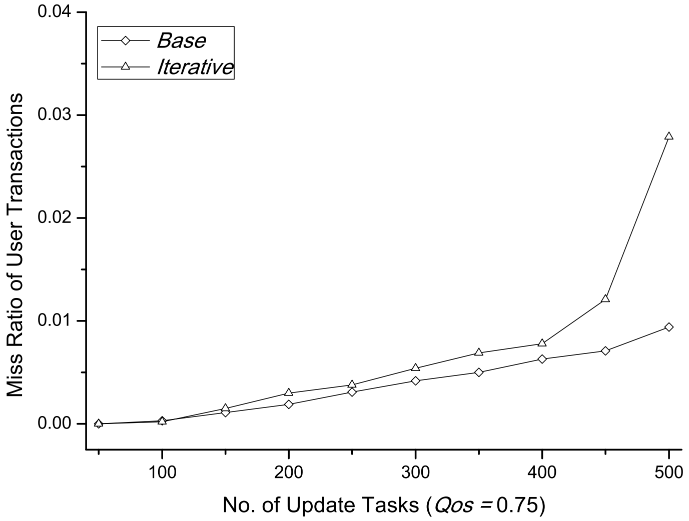
(b) $C_i \in \{2, 8\}$

Fig. 19 CPU utilization vs. no. of update tasks

As can be observed in Fig. 20, the miss ratios of both *Base Scheduler* and *Iterative Scheduler* remain zero when the number of update tasks is not large, e.g., smaller than 150 in Fig. 20(a). However, when the number of update tasks is large (e.g., more than 250), some user transactions miss their deadlines due to heavy update workload, as shown in Figs. 21(a) and (b). As shown in Fig. 20(b), the total CPU utilization of all the update jobs is more than 70 percents for the *Iterative Scheduler* when the number of update tasks is 500. It is important to note in the figure that the miss ratios of the *Base Scheduler* are lower than that of the *Iterative Sched-*



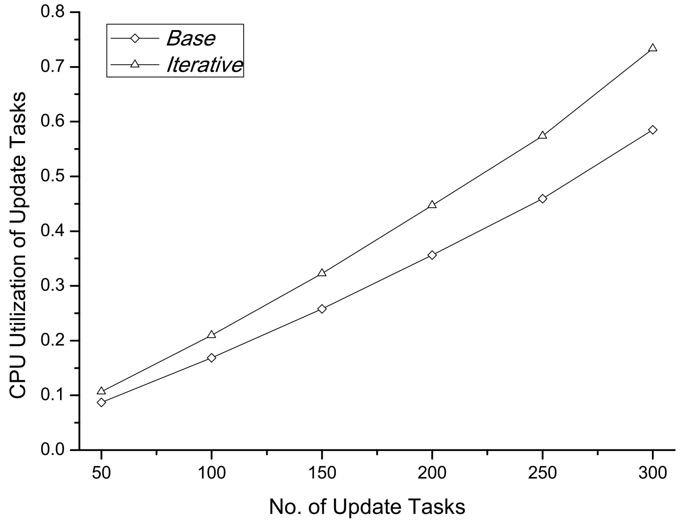
(a) $C_i \in [5, 15]$



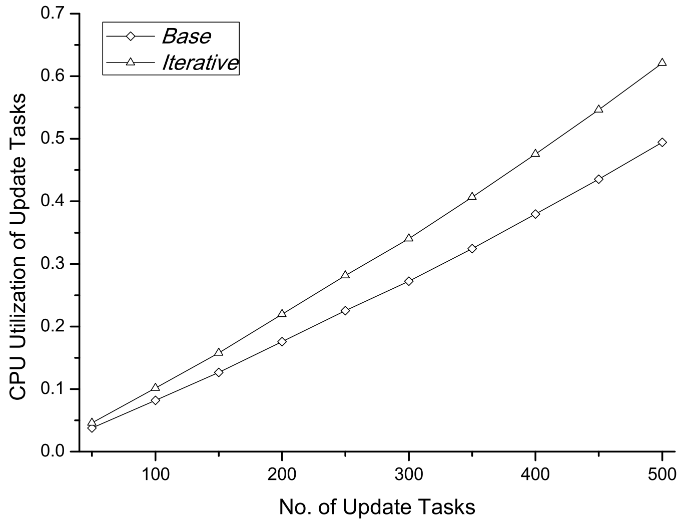
(b) $C_i \in \{2, 8\}$

Fig. 20 Miss ratio vs. no. of update tasks

uler when the update task workload is very heavy. It is because the *Iterative Scheduler* accepts more update jobs in particular the late jobs to maximize the temporal consistency of the real-time data objects. The tradeoff of higher temporal consistency from the *Iterative Scheduler* is that the total update workload becomes higher and the impacts to user transactions is more significant. As shown in Fig. 22, the achieved QoS (consistency ratio) from the *Iterative Scheduler* is significantly much higher than that from the *Base Scheduler*. It is close to 100 percents for differ-



(a) $C_i \in [5, 15]$



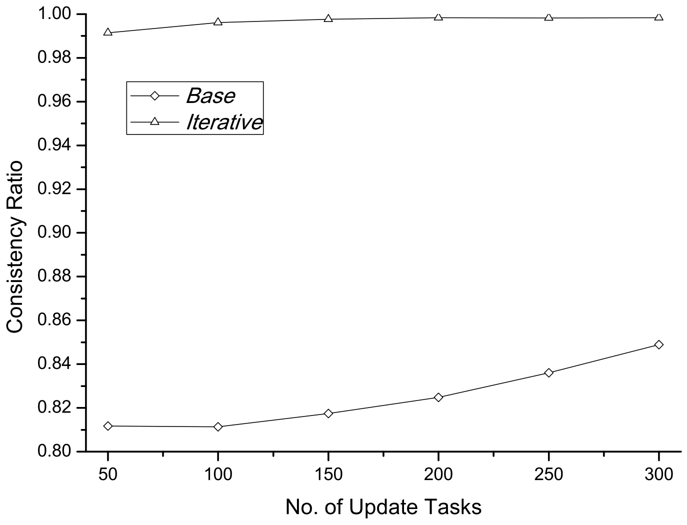
(b) $C_i \in \{2, 8\}$

Fig. 21 CPU utilization vs no. of update tasks

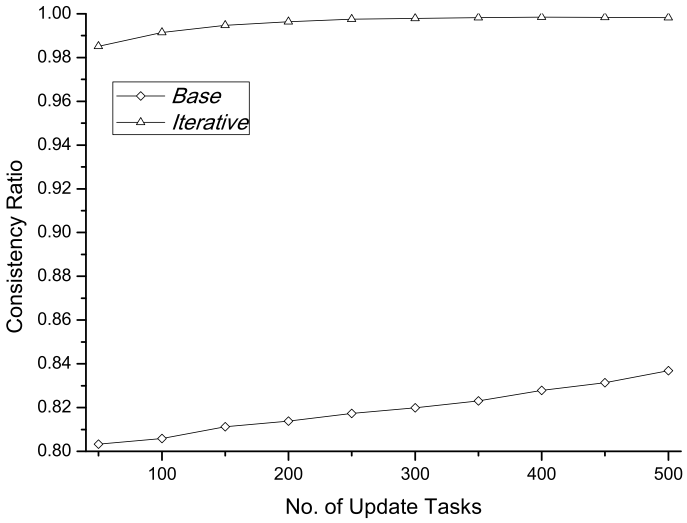
ent number of update tasks while that from the *Base Scheduler* is around 80 percents.

7 Conclusions and future works

This paper studies the problem of how to maintain temporal consistency of real-time data in distributed real-time systems where transmission delays cannot be simply ig-



(a) $C_i \in [5, 15]$



(b) $C_i \in \{2, 8\}$

Fig. 22 CPU utilization vs. no. of update tasks

nored. The variation in jitter values can seriously affect the schedulability of the systems and the total cost for installing the update jobs. However, the previous works, such as *More-Less*, did not provide a thorough study on this problem and only used an oversimplified assumption to handle the jitter problem. In this paper, based on *More-Less* (*ML*), we propose two extensions to deal with the jitter problems, called Jitter-based More-Less (*JB-ML*) and Statistical Jitter-based More-Less (*SJB-ML*). *JB-ML* assumes that different tasks may have different jitters, and it provides a deterministic guarantee in temporal consistency of the real-time data. *SJB-ML* further assumes that the jitters may vary among different jobs from the same task, and provides a statistical guarantee based on given *QoS* requirements of the real-time data objects. It is shown through our theoretical analysis and extensive experiments that both *JB-ML* and *SJB-ML* can significantly outperform the *More-Less* scheme in terms of improving the schedulability while still maintaining the required real-time data temporal consistency.

Most of the previous works on this topic concentrate on the scheduling of update jobs by assuming the use of the update first method for co-scheduling with user transactions such that the scheduling of update jobs will not be affected by the scheduling of user transactions. However, the update first method may not be very effective in maximizing the performance of user transactions. Thus, an important future work is to study efficient co-scheduling algorithms such that the temporal consistency of real-time data objects can be maintained and at the same time the response time constraints of user transactions can be satisfied.

References

Ahmed QN, Vrbsky SV (2000) Triggered updates for temporal consistency in real-time databases. Real-Time Syst 19(3):209–243

Amirijoo M, Hansson J, Son SH, Member S (2006) Specification and management of QoS in real-time databases supporting imprecise computations. IEEE Trans Comput, 304–319

Bernstein P, Hadzilacos V, Goodman N (1987) Concurrency Control and Recovery in Database Systems, vol 370. Addison-wesley, New York

Burns A, Tindell K, Wellings AJ (1995) Effective analysis for engineering real-time fixed priority schedulers. IEEE Trans Softw Eng 21:475–480

Di Natale M, Stankovic JA (1994) Dynamic end-to-end guarantees in distributed real-time systems. In: Proc. of the IEEE real-time systems symposium, pp 216–227

Golab L, Johnson T, Shkapenyuk V (2009) Scheduling updates in a real-time stream warehouse. In: Proc. of the IEEE international conference on data engineering, pp 1207–1210

Gustafsson T, Hansson J (2004) Data management in real-time systems: a case of on-demand updates in vehicle control systems. In: Proc. of the IEEE real-time and embedded technology and applications symposium, pp 182–191

Han S, Chen D, Xiong M, Mok AK (2008) A schedulability analysis of deferrable scheduling using patterns. In: Proc. of the euromicro conference on real-time systems

Han S, Chen D, Xiong M, Mok AK (2009) Online scheduling switch for maintaining data freshness in flexible real-time systems. In: Proc of the IEEE real-time systems symposium

Han S, Zhu X, Mok AK, Chen D, Nixon M (2011) Reliable and real-time communication in industrial wireless mesh networks. In: Proc of the IEEE real-time and embedded technology and applications symposium

Ho SJ, Kuo TW, Mok AK (1997) Similarity-based load adjustment for real-time data-intensive applications. In: Proc. of IEEE real-time system symposium, pp 144–153

Kang KD, Son SH, Stankovic JA (2004) Managing deadline miss ratio and sensor data freshness in real-time databases. IEEE Trans Knowl Data Eng **16**(10)

Kuo TW, Mok AK (1993) Ssp: a semantics-based protocol for real-time data access. In: Proc of the IEEE real-time systems symposium, pp 76–86

Kuo TW, Mok AK (1994) Using data similarity to achieve synchronization for free. In: 11th IEEE workshop on real-time operating systems and software, pp 112–116

Labrinidis A, Roussopoulos N (2001) Update propagation strategies for improving the quality of data on the web. In: Proc of the international conference on very large databases, pp 391–400

Lam KY, Kuo TW (2001) Real-time database systems: architecture and techniques. Kluwer Academic Amsterdam

Lam KY, Xiong M, Liang B, Guo Y (2004) Statistical quality of service guarantee for temporal consistency of real-time data objects. In: Proc of the IEEE real-time systems symposium

Leung J, Whitehead J (1982) On the complexity of fixed-priority scheduling of periodic real-time tasks. Perform Eval 2:237–250

Li H, Mason L (2007) Estimation and simulation of network delay traces for voip in service overlay network. In: Proc of the international symposium on signals, systems and electronics, pp 423–425

Li J, Chen JJ, Xiong M, Li G (2011) Workload-aware partitioning for maintaining temporal consistency upon multiprocessor platforms. In: Real-time systems symposium

Li M, Liu Y (2009) Underground coal mine monitoring with wireless sensor networks. ACM Trans Sens Netw 5(2):1–29

Locke D (1997) Real-time databases: real-world requirements. In: Bestavros A, Lin KJ, Son SH (eds) Real-time database systems: issues and applications. Kluwer Academic, Amsterdam pp 83–91

Lu C, Wang X, Koutsoukos X (2005) Feedback utilization control in distributed real-time systems with end-to-end tasks. IEEE Trans Parallel Distrib Syst, 550–561

Papageorgiou M, Diakaki C, Dinopoulou V, Kotsialos A, Wang Y (2005) Review of road traffic control strategies. Proc IEEE 91(12):2043–2067

Qu H, Labrinidis A (2007) Preference-aware query and update scheduling in web-databases. In: Proc of the IEEE international conference on data engineering, pp 356–365

Ramamritham K (1993) Real-time databases. Distrib Parallel Databases 1:199–226

Ramamritham K, Son SH, Dipippo LC (2004) Real-time databases and data services. Real-Time Syst 28(2):179–215

Saifullah A, Xu Y, Lu C, Chen Y (2010) Real-time scheduling for wirelesshart networks. In: Real-time systems symposium

Saifullah A, Xu Y, Lu C, Chen Y (2011) End-to-end delay analysis for fixed priority scheduling in wirelesshart networks. Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium

Sha L, Rajkumar R, Lehoczky JP (1990) Priority inheritance protocols: an approach to real-time synchronization. IEEE Trans Comput 39(9):1175–1185

Shanker U, Misra M, Sarje AK (2008) Distributed real time database systems: background and literature review. Distrib Parallel Databases 23(2):127–149

Song J, Han S, Mok AK, Chen D, Lucas M, Nixon M, Pratt W (2008) WirelessHART: applying wireless technology in real-time industrial process control. In: Proc. of the IEEE real-time and embedded technology and applications symposium, pp 377–386

Tan R, Xing G, Liu B, Wang J (2009) Impact of data fusion on real-time detection in sensor networks. In: Proc. of the IEEE real-time systems symposium, pp 323–332

Xiang J, Li GH, Xu HJ, Du XK (2008) Data freshness guarantee and scheduling of update transactions in RTMDBS. In: Proc of the international conference on wireless communications, networking and mobile computing, pp 1–4

Xiong M, Ramamritham K (1999) Deriving deadlines and periods for real-time update transactions. In: Proc of the IEEE real-time systems symposium

Xiong M, Ramamritham K (2004) Deriving deadlines and periods for real-time update transactions. IEEE Trans Comput 53:567–583

Xiong M, Han S, Lam KY (2005) A deferrable scheduling algorithm for real-time transactions maintaining data freshness. In: Proc of the IEEE real-time systems symposium, pp 27–37

Xiong M, Han S, Chen D (2006) Deferrable scheduling for temporal consistency: schedulability analysis and overhead reduction. In: Proc of the IEEE international conference on embedded and real-time computing systems and applications

Xiong M, Han S, Lam KY, Chen D (2008a) Deferrable scheduling for maintaining real-time data freshness: algorithms, analysis, and results. IEEE Trans Comput

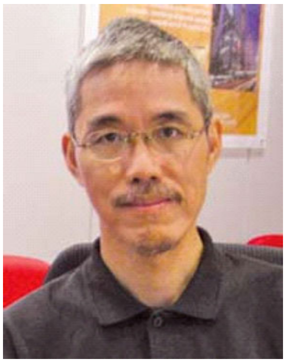
Xiong M, Wang Q, Ramamritham K (2008b) On earliest deadline first scheduling for temporal consistency maintenance. Real-Time Syst 40(2):208–237

Xiong M, Han S, Chen D, Lam KY, Feng S (2010) Desh: overhead reduction algorithms for deferrable scheduling. Real-Time Syst 44(1–3):1–25

Zuhily A, Burns A (2007) Optimal $(d - j)$-monotonic priority assignment. Inf Proces Lett 103(6):247–250

Jiantao Wang received the BS degree in computer science from University of Science and Technology of China (USTC), Peoples Republic of China, in 2009. He is currently working toward the PhD degree in the Department of Computer Sciences at City University of Hong Kong. His research interests include real-time systems, real-time data management, and flash-based database systems. He is a student member of the IEEE.

Song Han received the BS degree in computer science from Nanjing University, Peoples Republic of China in 2003 and the MPhil degree in computer science from City University of Hong Kong in 2006. He is currently a PhD candidate in the Department of Computer Science at the University of Texas at Austin. His research interests include cyber-physical systems, real-time and embedded systems, database systems and wireless networks. He is a student member of the IEEE.



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

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, where he has been a member of the faculty of the Department of Computer Sciences since 1983. He has performed extensive research on computer software systems and is internationally known for his work in real-time systems. He is a past chairman of the Technical Committee on Real-Time Systems of the IEEE and has served on numerous national and international research and advisory panels. His current interests include real-time and embedded systems, robust and secure network centric computing, and real-time knowledge-based systems. In 2002, Dr. Mok received the IEEE Technical Committee on Real-Time Systems Award for his outstanding technical contributions and leadership achievements in real-time systems. He is a member of the IEEE.