1999

Deriving Deadlines and Periods for Real-Time Update Transactions

Ming Xiong
University of Massachusetts - Amherst

Krithi Ramamritham
University of Massachusetts - Amherst

Deriving Deadlines and Periods for Real-Time Update Transactions

Ming Xiong & Krithi Ramamritham[*]
Department of Computer Science, University of Massachusetts, Amherst, MA 01003
Email: {xiong, krithi}@cs.umass.edu

Abstract

Typically, temporal validity of real-time data is maintained by periodic update transactions. In this paper, we examine the problem of period and deadline assignment for these update transactions such that (1) these transactions can be guaranteed to complete by their deadlines and (2) the imposed workload is minimized. To this end, we propose a novel approach, named More-Less principle. By applying this principle, updates occur with a period which is more than the period obtained through traditional approaches but with a deadline which is less than the traditional period. We show that the More-Less principle is better than existing approaches in terms of schedulability and the imposed load. We examine the issue of determining the assignment order in which transactions must be considered for period and deadline assignment so that the resulting workloads can be minimized. To this end, the More-Less principle is first examined in a restricted case where the Shortest Validity First (SVF) order is shown to be an optimal solution. We then relax some of the restrictions and show that SVF is an approximate solution which results in workloads that are close to the optimal solution. Our analysis and experiments show that the More-Less principle is an effective design principle that can provide better schedulability and reduce update transaction workload while guaranteeing data validity constraints.

1 Introduction

A real-time database (RTDB) is composed of real-time objects which are updated by periodic sensor transactions. An *object* in the database models a real world entity, for example, the position of an aircraft. A real-time object is one whose state may become invalid with the passage of time. Associated with the state is a temporal validity interval. To monitor the states of objects faithfully, a real-time object must be refreshed by a sensor transaction before it becomes invalid, i.e., before its temporal validity interval expires. The actual length of the temporal validity interval of a real-time object is application dependent. Sensor transactions are generated by intelligent sensors which periodically sample the value of real-time objects. When sensor transactions arrive at RTDBs with sampled data values, their updates are installed and real-time data are refreshed. So one of the important design goals of RTDBs is to guarantee that temporal data remain fresh, i.e., they are always valid. Therefore, efficient design principles are desired to guarantee the freshness of temporal data in RTDBs while minimizing the workload resulting from periodic sensor transactions.

In this paper, we propose the *More-Less* principle, a design principle which maintains the freshness of temporal data while reducing the workload incurred by periodic sensor transactions. It is shown that the *More-Less* principle outperforms traditional approaches in terms of sensor transaction schedulability and imposed workload. Using the *More-Less* principle, transactions are considered in a given order and their periods and deadlines are assigned. So an important issue is to determine the order so that the imposed transaction workload can be minimized. It is demonstrated, through both analysis and experiments, that *Shortest Validity First (SVF)* is an efficient assignment order to minimize workload for update transactions.

This paper is organized as follows: Section 2 reviews traditional approaches and introduces the intuition underlying the *More-Less* principle. The *More-Less* principle is formally introduced in Section 3, and compared with a traditional approach. We also examine the issue of determining the assignment order. Specifically, we propose and analyze *Shortest Validity First (SVF)*, an efficient transaction assignment order to minimize workload. An application of the *More-Less* principle is discussed in Section 4. Experimental results are presented in Section 5. We conclude the paper in Section 6.

2 Design Principles

In this section, traditional approaches for maintaining temporal validity, namely *One-One* and *Half-Half* principles, are reviewed, then the *More-Less* principle is introduced through an example. Formal definitions of some of the often-used symbols are given in Table 1.

We assume a simple execution semantics for periodic transactions: a transaction must be executed once every period. However, there is no guarantee on when an instance of

[*] Research supported in part by the National Science Foundation Grant IRI-9619588 and CDA-9502639. Krithi Ramamritham is also affiliated with India Institute of Technology, Bombay.

Symbol	Definition
X_i	Temporal Data i
τ_i	Periodic sensor transaction updating X_i
J_{ij}	The jth instance of τ_i
R_{ij}	Response time of the jth instance of τ_i
C_i	Computation time of transaction τ_i
α_i	Validity interval length of X_i
L_i	Validity interval slack of transaction τ_i, i.e., $L_i = \alpha_i - C_i$.
P_i	Period of transaction τ_i
D_i	Relative deadline of transaction τ_i
$\tau_i \rightarrow \tau_j$	In an assignment order, transaction τ_i precedes transaction τ_j.
U_{ij}	Given an assignment order $\tau_i \rightarrow \tau_j$ of two adjacent transactions τ_i and τ_j, CPU utilization of τ_i and τ_j. $U_{ij} = \frac{C_i}{P_i} + \frac{C_j}{P_j}$

Table 1. Symbols and definitions.

a periodic transaction is actually executed within a period.

2.1 One-One Principle

According to the first principle, the period and relative deadline[1] of a sensor transaction have to be equal to the data validity interval. Because the separation of the execution of two consecutive instances of a transaction can be more than the validity interval, data can become invalid under the *One-One* principle. So this principle can not guarantee the freshness of temporal data in RTDBs.

Example 2.1: Consider Figure 1: A periodic sensor transaction τ_i with deterministic execution time C_i refreshes temporal data X_i with validity interval α_i. The period P_i and relative deadline D_i of τ_i are assigned the value α_i. Suppose $J_{i,j}$ and $J_{i,j+1}$ are two consecutive instances of sensor transaction τ_i. Transaction instance $J_{i,j}$ samples data X_i with validity interval $[T, T+\alpha_i]$ at time T, and $J_{i,j+1}$ samples data X_i with validity interval $[T+\alpha_i, T+2\alpha_i]$ at time $T+\alpha_i$. From Figure 1, the actual arrival time of $J_{i,j}$ and finishing time of $J_{i,j+1}$ can be as close as $2C_i$, and as far as $2P_i$, i.e., $2\alpha_i$ when the period of τ_i is α_i. In the latter case, the validity of data X_i refreshed by $J_{i,j}$ expires after time $T+\alpha_i$. Since $J_{i,j+1}$ can not refresh data X_i before time $T+\alpha_i$, X_i is invalid from $T+\alpha_i$ until it is refreshed by $J_{i,j+1}$, just before the next deadline $T+2\alpha_i$. $\square$

2.2 Half-Half Principle

In order to guarantee the freshness of temporal data in RTDBs, the period and relative deadline of a sensor transaction are typically set to be less than or equal to one-half of the data validity interval [10]. In Figure 1, the farthest distance (based on the arrival time of a periodic transaction instance and the finishing time of its next instance) of two consecutive sensor transactions is $2P_i$. If $2P_i \le \alpha_i$, then the freshness of temporal data X_i is guaranteed as long as instances of sensor transaction τ_i does not miss their deadlines. Recent work [6] on similarity-based load adjustment also adopts this principle to adjust periods of sensor transactions based on similarity bound.

[1]The relative deadline of a transaction = transaction deadline - transaction arrival time.

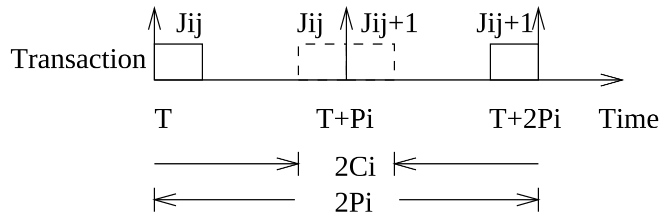

Figure 1. Extreme execution cases of periodic sensor transactions

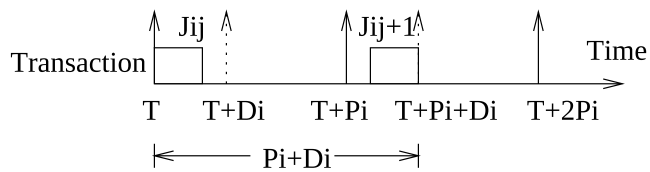

Figure 2. Illustration of *More-Less* principle

Unfortunately, even though data freshness is guaranteed, this design principle at least doubles the sensor transaction workload in the RTDBs compared to the *One-One* principle. Next, we introduce a new principle which guarantees the freshness of temporal data but incurs much less workload compared to the *Half-Half* principle.

2.3 *More-Less* Principle: Intuition

The goal of the *More-Less* principle is to minimize sensor transaction workload while guaranteeing the freshness of temporal data in RTDBs. For simplicity of discussion, we assume that a sensor transaction is responsible for updating one temporal data item in the system. In *More-Less*, the period of a sensor transaction is assigned to be *more* than *half* of the validity interval of the temporal data updated by the transaction, while its corresponding relative deadline is assigned to be *less* than *half* of the validity interval of the same data. However, the sum of the period and relative deadline is always equal to the length of the validity interval of the data updated. Consider Figure 2. Let $P_i > \frac{\alpha_i}{2}$, $C_i \le D_i < P_i$ where $P_i + D_i = \alpha_i$. The farthest distance (based on the arrival time of a periodic transaction instance and the finishing time of its next instance) of two consecutive sensor transactions J_{ij} and J_{ij+1} is $P_i + D_i$. In this case, the freshness of X_i can always be maintained if sensor transactions make their deadlines. Obviously, load incurred by sensor transaction τ_i can be reduced if P_i is enlarged (which implies that D_i is shrunk.). Therefore, we have the constraints $C_i \le D_i < P_i$ and $P_i + D_i = \alpha_i$ which aim at minimizing the workload of periodic transaction τ_i.

Principle	D_i	P_i	Utilization
One-One	α_i	α_i	$U_o = \frac{C_i}{\alpha_i} = \frac{1}{5}$
Half-Half	$\frac{\alpha_i}{2}$	$\frac{\alpha_i}{2}$	$U_h = \frac{2C_i}{\alpha_i} = \frac{2}{5}$
More-Less	$\frac{\alpha_i}{3}$	$\frac{2\alpha_i}{3}$	$U_m = \frac{3C_i}{2\alpha_i} = \frac{3}{10}$

Table 2. Comparison of three principles

Example 2.2: Suppose there is temporal data X_i with validity interval α_i in a uniprocessor RTDB system. τ_i updates X_i periodically. Our goal is to assign proper values to P_i and D_i given C_i and α_i so that CPU utilization resulting from sensor transaction τ_i can be reduced. Suppose $C_i = \frac{1}{5}\alpha_i$, possible values of P_i, D_i and corresponding CPU utilization according to the three different design principles are shown in Table 2. □

Only *Half-Half* and *More-Less* can guarantee the freshness of temporal data X_i if all the sensor transactions complete before their deadlines. We also notice that $U_o < U_m < U_h$. Intuitively, if $P_i = \frac{N-1}{N}\alpha_i$, then $D_i = \frac{1}{N}\alpha_i$, where $N \geq 2$. The freshness of temporal data in RTDBs is guaranteed if all sensor transactions complete before their deadlines. In such a case, we also notice that $U_m = \frac{NC_i}{(N-1)\alpha_i}$ and $U_o \leq U_m < U_h$. Theoretically, if $N \to \infty$, $U_m \to U_o$.

Unfortunately, how close U_m can get to U_o depends on C_i since $D_i \geq C_i$ implies $\frac{\alpha_i}{C_i} \geq N$. As N increases, relative deadlines become shorter and sensor transactions are executed with more stringent time constraints.

Therefore, given a set of sensor transactions in RTDBs, we need to find periods and deadlines of update transactions based on the temporal validity intervals of data such that the workload of sensor transactions is minimized and the schedulability of the resulting sensor transactions is guaranteed. The *More-Less* principle achieves this, as shown in the next section.

3 *More-Less*: Analysis and Results

In this section, we formally introduce the *More-Less* principle with three constraints: *Validity Constraint*, *Deadline Constraint* and *Schedulability Constraint*. We then show that the schedulability of transactions and data freshness are guaranteed by *More-Less*. To understand the advantages of *More-Less*, we then compare *More-Less* with *Half-Half* and show that *More-Less* is superior to *Half-Half* in terms of schedulability and for minimizing CPU utilization. We show that *assignment order*, i.e., the order in which periods and deadlines are assigned has an important impact on schedulability and CPU utilization of solutions derived from *More-Less*. Therefore, to find an optimal assignment order for *More-Less*, we investigate the issues of assignment order with the aid of a concept named *partitioning*. We show that *Shortest Validity First (SVF)*, an assignment order proposed in this paper, results in an optimal solution under certain restrictions. With the relaxation of some of the restrictions, it is proved that SVF produces an approximate solution within a certain bounded range of optimal solutions in general. SVF is shown to be a good heuristic solution in many applications, especially, where *validity interval lengths are much larger than transaction computation times*.

3.1 The Design Principle

From here on, $\mathcal{T} = \{\tau_i\}_{i=1}^m$ refers to a set of sensor transactions $\{\tau_1, \tau_2, .., \tau_m\}$ and $\mathcal{X} = \{X_i\}_{i=1}^m$ refers to a set of temporal data $\{X_1, X_2, .., X_m\}$ where X_i $(1 \leq i \leq m)$ is associated with a validity interval of length α_i: transaction τ_i $(1 \leq i \leq m)$ updates the corresponding data X_i. C_i, D_i and P_i $(1 \leq i \leq m)$ denote execution time, relative deadline, and period of transaction τ_i, respectively. Our goal is to determine P_i and D_i such that all the sensor transactions are schedulable and CPU utilization resulting from sensor transactions is minimized.

Although dynamic-priority scheduling is in general more effective than fixed-priority scheduling, they are also more difficult to implement and hence can incur higher system overhead than fixed-priority scheduling. Moreover, for many applications, it is possible to implement fixed-priority algorithms in the hardware level by the use of priority-interrupt mechanism. Thus, the overhead involved in scheduling tasks can be reduced to a minimal level [9]. Given this, we study fixed-priority scheduling algorithms in this paper.

For convenience, we use terms transaction and task interchangeably in this paper.

First, consider the longest response time for any instance of a periodic transaction τ_i where the response time is the difference between the transaction initiation time $(I_i + KP_i)$ and the transaction completion time where I_i is the offset within the period.

Theorem 3.1: For a set of periodic tasks $\mathcal{T} = \{\tau_i\}_{i=1}^m$ with task initiation time $(I_i + KP_i)$ $(K = 0, 1, 2, ...)$, the longest response time for any instance of τ_i occurs for the first instance of τ_i when $I_1 = I_2 = ... = I_m = 0$. [7] □

For $I_i = 0$ $(1 \leq i \leq m)$, the tasks are *in phase* because the first instances of all the tasks are initiated at the same time. It should be noted that we only discuss *in phase* tasks in this paper. A time instant after which a task has the longest response time is called a *critical instant*, e.g., time 0 is a critical instant for all the tasks if those tasks are *in phase*.

Further, Leung and Whitehead [9] introduced a fixed-priority scheduling algorithm, *deadline monotonic* scheduling algorithm, in which task priorities are assigned inversely with respect to task deadlines, that is, τ_i has higher priority than τ_j if $D_i < D_j$ [9].

Theorem 3.2: For a set of periodic tasks $\mathcal{T} = \{\tau_i\}_{i=1}^m$ with $D_i \leq P_i$ $(1 \leq i \leq m)$, the optimal fixed priority scheduling algorithm is the *deadline monotonic* scheduling algorithm. A task set is schedulable by this algorithm if the first instance of each task after a *critical instant* meets its deadline. □

Since the *deadline monotonic* algorithm is an optimal fixed priority scheduling algorithm for a set of tasks $\{\tau_i\}_{i=1}^m$ with $D_i \le P_i$ $(1 \le i \le m)$, it is used to maintain the schedulability of periodic transactions in our approach.

The *More-Less* principle determines deadlines and periods of transactions such that the following three constraints are satisfied:

- *Validity Constraint:* $P_i + D_i \le \alpha_i$
- *Deadline Constraint:* $C_i \le D_i \le P_i$
- *Schedulability Constraint:* Without loss of generality, assume that for $i < j$, $\tau_i \to \tau_j$ (i.e., τ_i precedes τ_j when they are considered for deadline and period assignment[2]). Because the transactions are scheduled by the *deadline monotonic* algorithm, the following inequality constraint must hold: $\sum_{j=1}^{i}(n_{ij} \times C_j) \le D_i$ $(1 \le i \le m)$, where n_{ij} denotes the number of times transaction τ_j occurs before the first instance of τ_i completes. Therefore, $\sum_{j=1}^{i}(n_{ij} \times C_j)$ represents the response time of the first instance of τ_i. It is easy to see that for any i, $n_{ii} = 1$.

The next theorem proves the correctness of the *More-Less* principle.

Theorem 3.3: Given a set of update transactions $\mathcal{T} = \{\tau_i\}_{i=1}^m$ $(m \ge 1)$ with deadlines and periods determined by *More-Less*, the set of transactions is schedulable and data freshness is guaranteed.

Proof: We need to prove that the three constraints of the *More-Less* principle can guarantee the schedulability of transactions and freshness of data. Because of *schedulability constraint*, the first instance of every transaction can meet its deadline. Combined with the *deadline constraint*, it follows from Theorem 3.2 that the set of transactions can be scheduled by the *deadline monotonic* algorithm. Since transactions satisfy *validity constraint*, data freshness can also be guaranteed. Hence the set of transactions is schedulable and data freshness is guaranteed. □

Given the *More-Less* principle, the optimization problem we need to solve is a non-linear programming problem: Determine D_i and P_i such that

$$U_m = \sum_{i=1}^{m} \frac{C_i}{P_i}$$

is minimized subject to the three constraints above.

From the three constraints underlying *More-Less*, we know that $P_i \le \alpha_i - \sum_{j=1}^{i}(n_{ij} \times C_j)$. Let $P_i = \alpha_i - \sum_{j=1}^{i}(n_{ij} \times C_j) - \beta_i$ $(\beta_i \ge 0)$. Now we transform the problem to be an assignment order problem so that $U_m = \sum_{i=1}^{m} \frac{C_i}{\alpha_i - \sum_{j=1}^{i}(n_{ij} \times C_j) - \beta_i}$ is minimized where $\beta_i \ge 0$.

It is easy to see that if U_m is minimized, then $\beta_i = 0$ for all i $(1 \le i \le m)$ and $D_i = \sum_{j=1}^{i}(n_{ij} \times C_j)$ $(1 \le i \le m)$. Now we have

$$P_i = \alpha_i - \sum_{j=1}^{i}(n_{ij} \times C_j) \;\; (1 \le i \le m). \qquad (1)$$

In particular, if $D_i = P_i$ $(1 \le i \le m)$, the *More-Less* principle actually reduces to the *Half-Half* principle.

The crux of the problem then, is to determine an assignment order for a set of transactions such that U_m is minimized. This is left to be discussed later in Sections 3.3, and 3.4. Next, we investigate the issue of computing D_i and P_i with a given transaction order for a set of transactions with known computation times and validity intervals. The following algorithm describes how to compute deadlines and periods of transactions.

Input: A set of update transactions $\mathcal{T} = \{\tau_i\}_{i=1}^m$ $(m \ge 1)$ with CPU computation times $\{C_i\}_{i=1}^m$ and validity interval lengths $\{\alpha_i\}_{i=1}^m$ as well as an assignment order $\tau_1 \to \tau_2 \to \ldots \to \tau_m$.
Output: Deadlines $\{D_i\}_{i=1}^m$ and periods $\{P_i\}_{i=1}^m$.

Algorithm 3.1: Determine Deadlines and Periods according to *More-Less*

```
/* Compute the deadline and period of τ_1 */
D_1 = C_1;
P_1 = α_1 − D_1;
/* Compute D_i and P_i for the rest of the tasks in the
descending order of task priorities */
for i = 2 to m do
{
    R_i1 = C_i; /* Initiate R_i1, response time of J_i1 */
    do { /* Compute R_i1 iteratively */
       D_i = R_i1; /* Keep R_i1 for comparison */
       R_i1 = C_i; /* Initiate R_i1 to recompute it */
       /* Next, recompute R_i1 using D_i */
       for j = 1 to i − 1 do
         /* Account for the interference of higher
             priority tasks */
           { R_i1 = R_i1 + ⌈D_i / P_j⌉ C_j; }
    } while (R_i1 ≠ D_i) and (R_i1 ≤ α_i / 2);
    /* Computation of R_i1 stops if R_i1 does not
       change, or R_i1 exceeds α_i / 2 */
    if (R_i1 > α_i / 2)
    then abort; /* Unschedulable case */
    else P_i = α_i − D_i; /* Compute P_i */
}
```

The next example illustrates how Algorithm 3.1 derives deadlines and periods of transactions.

Example 3.1: A set of transactions is given in Table 3 with transaction numbers, computation times, and validity interval lengths. *Half-Half* and *More-Less* are applied

[2] We use assignment order and priority order interchangeably in the paper. For example, $\tau_i \to \tau_j$ also means τ_i has higher priority than τ_j.

i	C_i	α_i	*More-Less*		*Half-Half*
			D_i	P_i	$P_i(D_i)$
1	1	3	1	2	1.5
2	2	20	4	16	10

Table 3. Parameters and results for example 3.1

to the transaction set. The resulting deadlines and periods are computed from Algorithm 3.1 and shown in Table 3 with assignment order $\tau_1 \rightarrow \tau_2$, which is the same as the assignment order from the *rate monotonic* algorithm for the periods resulting from *Half-Half*. The CPU utilization for *More-Less* is $\frac{1}{2} + \frac{2}{16} = 0.625$, which is less than $\frac{1}{1.5} + \frac{2}{10} = 0.867$, the CPU utilization for *Half-Half*. □

Example 3.1 shows that *More-Less* can have lower CPU utilization than *Half-Half*. Given any set of transactions, does *More-Less* produce better schedulability than *Half-Half* ? This is answered in the affirmative next.

3.2 Comparison of *More-Less* and *Half-Half*

Theorem 3.4: If any set of update transactions $\mathcal{T} = \{\tau_i\}_{i=1}^m$ $(m \geq 1)$ can be scheduled to guarantee data freshness using any fixed priority scheduling algorithm based on periods derived from *Half-Half*, then it can also be scheduled by the *deadline monotonic* algorithm based on the *More-Less* principle.

Proof: If $m = 1$, it is trivial. Let us look at the case of $m > 1$. Without loss of generality, assume that transaction priorities are assigned in the order of $\tau_1 \rightarrow \tau_2 \rightarrow ... \rightarrow \tau_m$ by the *Half-Half* principle. Let us assume that the same priority order is retained by the *More-Less* principle. Let D_i^H and P_i^H denote the deadline and period of transaction τ_i in *Half-Half*, and D_i^M and P_i^M denote the deadline and period of transaction τ_i in *More-Less*, respectively. Also let R_{ij}^H and R_{ij}^M denote the response time of the jth instance of transaction τ_i in the *Half-Half* and *More-Less* principle, respectively. We know that $D_i^H = P_i^H = \frac{\alpha_i}{2}$. Since the set of transactions can be scheduled by a fixed priority scheduling algorithm based on *Half-Half*, we will prove that $D_i^H \geq D_i^M$ and $P_i^H \leq P_i^M$. This can be proved by induction.

- In case of $m > 1$, we know that $C_1 < D_1^H$. Otherwise, $C_1 = D_1^H$ implies that $C_1 = P_1^H$, i.e., τ_1 would consume 100% CPU and other transactions would not be scheduled. Since $R_{11}^H = C_1$ and $C_1 < D_1^H$, we know that $R_{11}^H < D_1^H$. Because $D_1^H = P_1^H = \frac{\alpha_1}{2}$, we have $R_{11}^H < \frac{\alpha_1}{2}$. Because τ_1 has the highest priority in the task set, we have $R_{11}^M = R_{11}^H$. Hence $D_1^M = R_{11}^M = C_1$, which is less than $\frac{\alpha_1}{2}$. According to *More-Less*, let $P_1^M = \alpha_1 - D_1^M$, which implies $P_1^M > \frac{\alpha_1}{2}$.
- Assume that for all $1 \leq j \leq i-1$, $D_j^M \leq D_j^H$ and $P_j^M \geq P_j^H$ hold. Then

i	C_i	α_i	*More-Less*		*Half-Half*
			D_i	P_i	$P_i(D_i)$
1	1	4	1	3	2
2	1	5	2	3	2.5
3	1	8	3	5	4
4	1	20	9	11	10

Table 4. Parameters and results of example 3.2

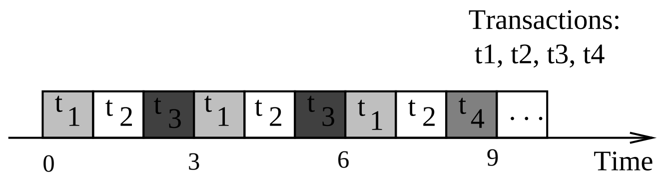

Figure 3. A solution produced by *More-Less*

$R_{i1}^H = C_i + \sum_{j=1}^{i-1}(\lceil \frac{R_{i1}^H}{P_j^H} \rceil C_j) > C_i + \sum_{j=1}^{i-1}(\lceil \frac{R_{i1}^H}{P_j^M} \rceil C_j)$. It is clear that $\exists(t \leq R_{i1}^H)$ such that $t = C_i + \sum_{j=1}^{i-1}(\lceil \frac{t}{P_j^M} \rceil C_j)$. Let $D_i^M = R_{i1}^M = t$. This implies that $D_i^M \leq R_{i1}^H$ because $t \leq R_{i1}^H$. Since $R_{i1}^H \leq D_i^H$, we have $D_i^M \leq D_i^H$, i.e., $D_i^M \leq \frac{\alpha_i}{2}$, which also implies $P_i^M \geq \frac{\alpha_i}{2}$ because $P_i^M = \alpha_i - D_i^M$. So $D_j^H \geq D_j^M$ and $P_j^H \leq P_j^M$ are true for $j = i$.

Therefore we can conclude that $D_i^M \leq P_i^M$ $(1 \leq i \leq m)$, and the first instance of τ_i $(1 \leq i \leq m)$ can make its deadline. It directly follows from Theorem 3.2 that the set of transactions with deadlines and periods derived from *More-Less* can be scheduled by *deadline monotonic*. □

From Theorem 3.4, if there is a feasible solution based on *Half-Half* for a set of transactions, there must be a feasible solution based on *More-Less*. However, the *converse* is not true. This is illustrated by Example 3.2.

Example 3.2: Transactions are listed in Table 4 with transaction numbers, computation times, validity interval lengths. *Half-Half* and *More-Less* are applied to the transaction set, and resulting deadlines and periods are shown in Table 4. It is clear from Table 4 that the transaction set resulting from *Half-Half* is non-schedulable because its CPU utilization is $\frac{1}{2} + \frac{1}{2.5} + \frac{1}{4} + \frac{1}{10} = 1.25 > 1.0$. However, transactions with periods resulting from *More-Less* is schedulable by assigning priorities $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$. In this case, the resulting CPU utilization is $\frac{1}{3} + \frac{1}{3} + \frac{1}{5} + \frac{1}{11} = 0.957$. Figure 3 shows that the first instance of every transaction in the set can meet its deadline, which indicates that the transaction set is schedulable according to Theorem 3.2. However, an assignment order $\tau_2 \rightarrow \tau_1 \rightarrow \tau_3 \rightarrow \tau_4$ under *More-Less* would not be able to produce a feasible solution. This indicates that assignment orders of transactions can significantly affect the schedulability of transactions. □

In addition, if any set of transactions can be scheduled to guarantee data freshness by any fixed priority scheduling algorithm based on *Half-Half*, there must be a solution based on *More-Less* with lower CPU utilization. It is clear from Theorem 3.4 that any conditions sufficient to guarantee the schedulability of a set of transactions using *Half-Half* must be sufficient to guarantee the schedulability using *More-Less*. The following lemma (see [14] for proof) gives a sufficient condition for schedulability of transactions based on *Half-Half*.

Lemma 3.1: Given any set of transactions $\mathcal{T} = \{\tau_i\}_{i=1}^m$ $(m \geq 1)$, if

$$\sum_{j=1}^{k} \lceil \frac{\alpha_k}{\alpha_j} \rceil C_j \leq \frac{\alpha_k}{2} (1 \leq k \leq m) \tag{2}$$

holds, then the set of transactions are schedulable by *More-Less*.

It should be noted that Eq. 2 is only a sufficient condition for feasibility test of scheduling a set of transactions based on *More-Less*, it is not the necessary condition. However, Eq. 2 is both a necessary and sufficient condition of scheduling a set of transactions with fixed priority scheduling algorithms based on *Half-Half*, that is, if Eq. 2 does not hold, a set of transactions is not schedulable based on *Half-Half*. However, it may still be schedulable using *More-Less*. As illustrated in example 3.2, assignment orders in *More-Less* may have significant impact on the schedulability of transactions. How to choose an appropriate assignment order to determine deadlines and periods remains a problem. An optimal assignment order is desirable for *More-Less* to guarantee schedulability and minimize CPU utilization of transactions.

3.3 *More-Less* Principle: An Optimal Solution in a Restricted Case

As far as we know, there is no known solution to solve the previous non-linear programming problem corresponding to producing optimal periods and deadlines under *More-Less*. The complexity arises from not only the non-linearity, but also the permutation of m transactions (i.e., the assignment order of the m transactions), which is $O(m!)$. If we enumerate all the permutations of m transactions to find the one with minimized CPU utilization, all $m!$ solutions have to be examined. It is obviously not efficient when the transaction set is large.

We now begin to examine the issue of finding optimal assignment order for *More-Less*. We first consider the problem with the following restriction:

Restriction (1): $\sum_{i=1}^m C_i \leq min(\frac{\alpha_j}{2})$ $(1 \leq j \leq m)$.

Under this restriction, the first instance of all transactions can complete before half of the shortest validity interval. Given any assignment order of transactions, this implies that no higher priority transactions can recur before the first instance of the lowest priority transaction completes. Otherwise, suppose J_{i2} (the 2nd instance of transaction τ_i) $(1 \leq i \leq m)$ is the first recurring instance, and it occurs at time t before the first instance of the lowest priority transaction completes. It implies that $t \geq P_i$. Because $P_i \geq \frac{\alpha_i}{2}$ according to *More-Less*, we have $t \geq \frac{\alpha_i}{2}$. Because not all the first instances from all transactions have completed yet, $t \leq \sum_{i=1}^m C_i$. Therefore we can conclude that $\frac{\alpha_i}{2} \leq \sum_{i=1}^m C_i$, which contradicts restriction (1). All the integers n_{ij} $(1 \leq i \leq m \ \& \ 1 \leq j \leq i)$ in the schedulability constraint are reduced to 1 under such a restriction. Due to the short execution time of sensor transactions and relatively long validity interval length in many real applications (e.g., avionics system in [6], air traffic control, aircraft mission processor, and spacecraft control in [8]), restriction (1) is reasonable in many cases. We discuss relaxing this condition later in the paper. In the rest of Section 3.3, we assume that restriction (1) holds. In the rest of the paper, we also assume that transactions are ordered so that $\tau_i \to \tau_j$ for $i < j$ unless specified otherwise.

3.3.1 *More-Less* Principle: Optimal Assignment Order for Two Transactions

To motivate our approach to determining the ordering of transactions, we first study the characteristics of a set of two transactions: τ_1 and τ_2. The question we are trying to answer is, which one should precede the other ? Two cases are examined:

1. $\tau_1 \tau_2$: $\tau_1 \to \tau_2$

$$\begin{cases} P_1 = \alpha_1 - C_1 \\ P_2 = \alpha_2 - (C_1 + C_2) \end{cases} \tag{3}$$

2. $\tau_2 \tau_1$: $\tau_2 \to \tau_1$

$$\begin{cases} P_2 = \alpha_2 - C_2 \\ P_1 = \alpha_1 - (C_1 + C_2) \end{cases} \tag{4}$$

In the above two cases, it should be noted that higher priority transaction only occurs once before the first instance of the lower priority transaction completes because restriction (1) holds. Let U_{12} and U_{21} denote the CPU utilization of transactions τ_1 and τ_2 in cases $\tau_1\tau_2$ and $\tau_2\tau_1$, respectively. Now we have

$$\begin{cases} U_{12} = \sum_{i=1}^2 \frac{C_i}{P_i} = \frac{C_1}{\alpha_1 - C_1} + \frac{C_2}{\alpha_2 - (C_1 + C_2)} \\ U_{21} = \sum_{i=1}^2 \frac{C_i}{P_i} = \frac{C_2}{\alpha_2 - C_2} + \frac{C_1}{\alpha_1 - (C_1 + C_2)} \end{cases} \tag{5}$$

Without loss of generality, assume we want to show that $U_{12} \leq U_{21}$. That is

$$\frac{C_1}{\alpha_1 - C_1} + \frac{C_2}{\alpha_2 - (C_1 + C_2)} \leq \frac{C_2}{\alpha_2 - C_2} + \frac{C_1}{\alpha_1 - (C_1 + C_2)} \tag{6}$$

We now study the conditions that satisfy Eq. 6.

Let L_i denote the validity interval slack of transaction τ_i, i.e., $L_i = \alpha_i - C_i$. Also let $\Delta\alpha_{ij} = \alpha_i - \alpha_j$, $\Delta L_{ij} = L_i -$

L_j, and $\Delta C_{ij} = C_i - C_j$. It is obvious that $\Delta\alpha_{ji} = -\Delta\alpha_{ij}$, $\Delta L_{ji} = -\Delta L_{ij}$, and $\Delta C_{ji} = -\Delta C_{ij}$. We now introduce the following theorem.

Theorem 3.5: If

1. $\frac{\alpha_i}{2} \geq C_i + C_j$ and $\frac{\alpha_j}{2} \geq C_i + C_j$ (i.e., restriction (1) holds).
2. $\Delta\alpha_{ji} \geq 0$ and $\Delta C_{ji} \leq 2\Delta\alpha_{ji}$, i.e., for any $\alpha_j \geq \alpha_i$, the increase of computation time is less than twice the increase in validity interval length,

then $U_{ij} \leq U_{ji}$. □

Theorem 3.5, proved in [14], has the following properties under restriction (1): *stability*, *transitivity* and *simplicity* as described below.

Definition 3.1: **Stability:** No matter how many transactions are assigned higher priority than two adjacent transactions τ_i and τ_j (i.e., no other transactions exist with priority between τ_i and τ_j), the ordering of τ_i and τ_j is *stable* which means U_{ij} is always less than U_{ji}.

Property 1. If transactions τ_i and τ_j satisfies the two conditions in Theorem 3.5, then the ordering of τ_i and τ_j is *stable*, i.e., $U_{ij} \leq U_{ji}$ always holds.

Proof: To prove that $U_{ij} \leq U_{ji}$ always holds, we need to prove that no matter how many transactions are assigned higher priority than τ_i and τ_j, $U_{ij} \leq U_{ji}$ always holds.

Suppose *k* transactions, $\tau_1, \tau_2, ..., \tau_k$, have been assigned higher priorities than τ_i and τ_j. The sum of their computation times is $\sum_{l=1}^{k} C_l = C$. Now we want $U_{ij} \leq U_{ji}$ to hold, i.e., $\frac{C_i}{\alpha_i - C - C_i} + \frac{C_j}{\alpha_j - (C + C_i + C_j)} \leq \frac{C_j}{\alpha_j - C - C_j} + \frac{C_i}{\alpha_i - (C + C_i + C_j)}$. Let $\alpha_i^* = \alpha_i - C$ and $\alpha_j^* = \alpha_j - C$, thus

$$\Delta\alpha_{ji}^* = \alpha_j^* - \alpha_i^* = \alpha_j - \alpha_i = \Delta\alpha_{ji}. \quad (7)$$

We know that $\Delta C_{ji} \leq 2\Delta\alpha_{ji}$ if $C = 0$ because of condition 2 in Theorem 3.5. Combined with Eq. 7, $\Delta C_{ji} \leq 2\Delta\alpha_{ji} \Leftrightarrow \Delta C_{ji} \leq 2\Delta\alpha_{ji}^*$, and this implies $\frac{C_i}{\alpha_i^* - C_i} + \frac{C_j}{\alpha_j^* - (C_i + C_j)} \leq \frac{C_j}{\alpha_j^* - C_j} + \frac{C_i}{\alpha_i^* - (C_i + C_j)}$ holds from Theorem 3.5. That is, $U_{ij} \leq U_{ji}$ holds. □

Definition 3.2: **Transitivity:** If τ_i preceding τ_j results in lower CPU utilization for transactions τ_i and τ_j (i.e., $U_{ij} \leq U_{ji}$), and τ_j preceding τ_k results in lower CPU utilization for transactions τ_j and τ_k (i.e., $U_{jk} \leq U_{kj}$), then τ_i preceding τ_k results in lower CPU utilization for transactions τ_i and τ_k (i.e., $U_{ik} \leq U_{ki}$).

Property 2. Transactions satisfying conditions in Theorem 3.5 maintain *transitivity*.

Proof: Given transactions τ_i, τ_j and τ_k, suppose $\Delta C_{ji} \leq 2\Delta\alpha_{ji}$ and $\Delta C_{kj} \leq 2\Delta\alpha_{kj}$. Because $2\Delta\alpha_{ki} = 2(\alpha_k - \alpha_i) = 2(\alpha_k - \alpha_j) + 2(\alpha_j - \alpha_i) = 2\Delta\alpha_{kj} + 2\Delta\alpha_{ji} \geq \Delta C_{kj} + \Delta C_{ji} = \Delta C_{ki}$, we have $U_{ik} \leq U_{ki}$. □

Property 3. Determining the conditions necessary from Theorem 3.5 for $U_{ij} \leq U_{ji}$ is computationally efficient because the computation of $\Delta\alpha_{ji}$ and ΔC_{ij} is *simple*.

Discussion

In Theorem 3.5, $\Delta\alpha_{ji} \geq 0$ and $\Delta C_{ji} \leq 2\Delta\alpha_{ji}$ include two cases:

1. $\Delta\alpha_{ji} \geq \Delta C_{ji}$, which implies $\alpha_i \leq \alpha_j$ and $\alpha_i - C_i \leq \alpha_j - C_j$, i.e., α_i and $\alpha_i - C_i$ order transactions in the *same* way.
2. $2\Delta\alpha_{ji} \geq \Delta C_{ji} \geq \Delta\alpha_{ji}$, which implies $\alpha_i < \alpha_j$, $\alpha_j - C_j \leq \alpha_i - C_i$, $(\alpha_i - C_i) - (\alpha_j - C_j) \leq \alpha_j - \alpha_i$, i.e., α_i and $\alpha_i - C_i$ do *not* order transactions in the same way.

τ_i preceding τ_j produces lower CPU utilization in the above two cases. Thus, $\alpha_i - C_i$ values of transactions may not produce the best assignment order. Said differently, the Least Slack First assignment algorithm may not produce the lowest utilization.

3.3.2 *More-Less* Principle: Optimal Ordering of m Transactions

To generalize the comparison of two transactions, we need to examine a set of transactions $\{\tau_i\}_{i=1}^{m}$ with $m > 2$. We first introduce the second restriction in this paper.

Restriction (2):

$$\begin{cases} \alpha_1 \leq \alpha_2 \leq ... \leq \alpha_m \\ \Delta C_{i+1,i} \leq 2\Delta\alpha_{i+1,i} \ (i = 1, 2, .., m-1) \end{cases}$$

The next theorem proposes an optimal solution under restrictions (1) and (2).

Theorem 3.6: Given a set of transactions $\mathcal{T} = \{\tau_i\}_{i=1}^{m}$, if restrictions (1) and (2) hold then an assignment order named *Shortest Validity First(SVF)*, which assigns orders to transactions in the *inverse* order of validity interval length and resolves ties in favor of a transaction with less *slack*[3], results in the optimal CPU utilization among all possible assignment orders of the *More-Less* principle.

Proof: We need to prove that the transaction ordering scheme from SVF results in the lowest CPU utilization. From restriction (2) and Theorem 3.5, we know that $U_{i,i+1} \leq U_{i+1,i}$ $(1 \leq i \leq m-1)$, and this is stable and transitive. Suppose there is an optimal assignment ordering K resulting from an order different from SVF. But that order can always be achieved by a sequence of swapping of priorities of two adjacent transactions in our SVF scheme. From the *stability* and *transitivity* of Theorem 3.5, we know that every swap of orders of two adjacent transactions in the SVF scheme would result in higher CPU utilization. Thus order K has higher CPU utilization than the SVF scheme. This contradicts the assumption that K is optimal. Therefore we have proved that transaction ordering scheme based on SVF results in the optimal CPU utilization. □

Example 3.3: In Table 5, a set of transactions satisfies restrictions (1) and (2), therefore an assignment order $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ results in an optimal solution for *More-Less*. *Half-Half* and *More-Less* are applied to the transaction set, respectively, and resulting deadlines and periods are shown in

[3]As in Table 1, slack L_i for transaction τ_i is defined as $\alpha_i - C_i$.

i	C_i	α_i	*More-Less*		*Half-Half*
			D_i	P_i	$P_i(D_i)$
1	1	8	1	7	4
2	1	10	2	8	5
3	1	12	3	9	6

Table 5. Illustration of an optimal solution

U_{123}	U_{132}	U_{213}	U_{231}	U_{312}	U_{321}
0.379	0.386	0.389	0.411	0.400	0.416

Table 6. CPU utilization of all possible orderings

Table 5. The resulting CPU utilization of the solution from *More-Less* is $\frac{1}{7}+\frac{1}{8}+\frac{1}{9}=0.379$. This is an optimal CPU utilization among all the priority assignments of *More-Less*, and it is much lower than CPU utilization of the solution from *Half-Half*, which is $\frac{1}{4}+\frac{1}{5}+\frac{1}{6}=0.62$. CPU utilizations of all possible assignment orders are listed in Table 6 in which U_{XYZ} represents utilization of assignment order $\tau_X \rightarrow \tau_Y \rightarrow \tau_Z$. We can see that SVF does result in the optimal CPU utilization in this case. □

The next example illustrates that SVF does not produce an optimal solution if restriction (2) does not hold.

Example 3.4: In Table 7, it is obvious that the set of transactions does not satisfy restriction (2) because $\frac{\Delta C_{21}}{\Delta \alpha_{21}} = \frac{4-1}{11-10} = 3 > 2$, although restriction (1) holds. Therefore an assignment order $\tau_1 \rightarrow \tau_2$ does not result in an optimal solution for *More-Less*. Resulting deadlines and periods from different assignment orders under *More-Less* are shown in Table 7. In this case, the resulting CPU utilization of SVF is $\frac{1}{9}+\frac{4}{6}=0.778$, and the other order results in a CPU utilization of $\frac{1}{5}+\frac{4}{7}=0.771$. □

So, clearly, when restriction (1) holds but restriction (2) does not hold, SVF is not an optimal solution. But it is interesting to note that SVF produces a CPU utilization which is close to the optimal in such situations. This is the issue that is examined next.

3.4 *More-Less* Principle: An Approximate Solution and Its Bound

In this subsection, we explore the implication of using SVF even when restriction (2) in Theorem 3.6 does not hold, but restriction (1) holds. We will show that SVF can provide a CPU utilization bounded within a certain range of that of the optimal solution. This is analyzed through the help of transaction *partitioning*, a powerful technique which can help derive the CPU utilization bound when using SVF as an approximation of the optimal assignment order.

Definition 3.3: Partition: Given a set of transactions $\mathcal{T} = \{\tau_i\}_{i=1}^m$ $(m \geq 1)$, if a transaction τ_k $(1 \leq k \leq m)$ is partitioned into n $(n > 1)$ independent subtransactions $\{\tau_{kj}\}_{j=1}^n$ with computation time $\{C_{kj}\}_{j=1}^n$ $(\sum_1^n C_{kj} =$

i	C_i	α_i	$\tau_1 \rightarrow \tau_2$		$\tau_2 \rightarrow \tau_1$	
			D_i	P_i	D_i	P_i
1	1	10	1	9	5	5
2	4	11	5	6	4	7

Table 7. SVF is non-optimal case

$C_k)$, and validity interval length $\{\alpha_{kj}\}_{j=1}^n$ $(\alpha_{kj} = \alpha_k)$, then the set of transactions $\{\tau_{kj}\}_{j=1}^n$ is a *partition* of τ_k, and the resulting set of transactions $\{\tau_i\}_{i=1}^{k-1} \cup \{\tau_{kj}\}_{j=1}^n \cup \{\tau_i\}_{i=k+1}^m$ is a *partition-transformed* set of the original transaction set.

It should be noted that *partition-transformation* is *transitive*. For example, if transaction set $\mathcal{T_B}$ is a partition-transformed set of transaction set $\mathcal{T_A}$, and transaction set $\mathcal{T_C}$ is a partition-transformed set of transaction set $\mathcal{T_B}$, then transaction set $\mathcal{T_C}$ is a partition-transformed set of transaction set $\mathcal{T_A}$.

We now investigate the impact of *partitioning* on CPU utilization of optimal solutions of a transaction set. We want to understand whether partitioning transactions into smaller subtransactions with shorter computation times would produce optimal solutions with lower CPU utilization. The following theorem holds even when restriction (1) is not satisfied.

Theorem 3.7: Given any set of transactions $\mathcal{T_O} = \{\tau_i\}_{i=1}^m$, a transaction τ_k $(1 \leq k \leq m)$ can be partitioned into n independent subtransactions $\{\tau_{kj}\}_{j=1}^n$ with $C_k = \sum_{j=1}^n C_{kj}$ and $(\alpha_{kj} = \alpha_k)$ $(1 \leq j \leq n)$. Let the partition-transformed transaction set be $\mathcal{T_P}$. Then for any solution generated by *More-Less*, the optimal CPU utilization of $\mathcal{T_P}$ is less than the optimal CPU utilization of $\mathcal{T_O}$.

Proof: For an optimal solution $\mathcal{S_O}^{opt}$ of $\mathcal{T_O}$ generated by *More-Less* with assignment order $\tau_1 \rightarrow \tau_2 \rightarrow ... \rightarrow \tau_m$, if a transaction $\tau_k \in \mathcal{T_O}$ $(1 \leq k \leq m)$ can be partitioned into n subtransactions $\tau_{k1}, ..., \tau_{kn}$, $\mathcal{T_O}$ is transformed into a transaction set $\mathcal{T_P} = \{\tau_1^{'}, .., \tau_{k-1}^{'}, \tau_{k1}^{'}, ..., \tau_{kn}^{'}, \tau_{k+1}^{'}, ..., \tau_m^{'}\}$ where $\tau_j^{'} = \tau_j$ $(j \neq k)$ and $\tau_{kj}^{'} = \tau_{kj}$ $(1 \leq k \leq n)$. Based on *More-Less*, we can obtain a feasible solution $\mathcal{S_P}$ from $\mathcal{S_O}^{opt}$ immediately with

$$\begin{cases} D_j^{'} = D_j (j < k) \\ D_{ji}^{'} < D_j (j = k \& 1 \leq i \leq n-1) \\ D_{ji}^{'} = D_j (j = k \& i = n) \\ D_j^{'} = D_j (j > k) \end{cases} \tag{8}$$

by assigning priorities in the order of $\tau_1^{'} \rightarrow ... \rightarrow \tau_{k-1}^{'} \rightarrow \tau_{k1}^{'} \rightarrow ... \rightarrow \tau_{kn}^{'} \rightarrow \tau_{k+1}^{'} \rightarrow ... \rightarrow \tau_m^{'}$. Thus, we know that

$$\begin{cases} P_j^{'} = P_j (j < k) \\ P_{ji}^{'} > P_j (j = k \& 1 \leq i \leq n-1) \\ P_{ji}^{'} = P_j (j = k \& i = n) \\ P_j^{'} = P_j (j > k) \end{cases} \tag{9}$$

We know that $\mathcal{T}_{\mathcal{P}}$ with above $\{D_i^{'}\}$ and $\{P_i^{'}\}$ can be scheduled because deadlines and periods are produced from a feasible solution, $\mathcal{S_O}^{opt}$. Considering that

$$U^{opt}_{\mathcal{T_O}} = \sum_{i=1}^{m} \frac{C_i}{P_i}, \tag{10}$$

and

$$U_{\mathcal{T_P}} = \sum_{i=1}^{k-1} \frac{C_i}{P_i} + \sum_{i=1}^{n} \frac{C_{ki}^{'}}{P_{ki}^{'}} + \sum_{i=k+1}^{m} \frac{C_i}{P_i}, \tag{11}$$

we know that $U^{opt}_{\mathcal{T_O}} > U_{\mathcal{T_P}}$. Because $U^{opt}_{\mathcal{T_P}}$, the optimal CPU utilization of $\mathcal{T_P}$, is less than or equal to $U_{\mathcal{T_P}}$, we can conclude that $U^{opt}_{\mathcal{T_O}} > U^{opt}_{\mathcal{T_P}}$. This proves the theorem. □

Theorem 3.7 is important because it says that a partition-transformed set can have lower optimal CPU utilization than the optimal CPU utilization of its original transaction set. Theorem 3.7 can be applied repeatedly to every transaction in $\mathcal{T_O}$. This generates a "finer" transaction set with even lower optimal CPU utilization. It is shown later in the paper that *partitioning* helps analyze *More-Less*.

Given a set of transactions $\mathcal{T}$ which satisfies restriction (1) but does not satisfy restriction (2), we can partition transactions which violate restriction (2) into a set of sub-transactions such that the partition-transformed transaction set $\mathcal{T_P}$ satisfies restriction (2). The optimal CPU utilization of the partition-transformed transaction set ($U^{opt}_{\mathcal{T_P}}$) can be obtained from Theorem 3.6, and this is less than the optimal CPU utilization of the original transaction set ($U^{opt}_{\mathcal{T}}$) as per Theorem 3.7. Thus, for any given solution $\mathcal{S}$ of $\mathcal{T}$ and its CPU utilization $U_{\mathcal{S}}$, $U_{\mathcal{S}} - U^{opt}_{\mathcal{T}} \le U_{\mathcal{S}} - U^{opt}_{\mathcal{T_P}}$ because $U_{\mathcal{S}} \ge U^{opt}_{\mathcal{T}}$.

Definition 3.4: Partition/Merge: Given any set of transactions $\mathcal{T} = \{\tau_1, \tau_2, .., \tau_m\}$ with $\alpha_1 \le \alpha_2 \le ... \le \alpha_m$, if restriction (1) holds but restriction (2) does not hold for $\mathcal{T}$, we can reconstruct the transaction set by partitioning the computation time of transactions so that restriction (2) holds.

1. *Partitioning of one transaction:* If there is one transaction τ_k with $\alpha_{k-1} \le \alpha_k$ and $C_k > C_{k-1} + 2(\alpha_k - \alpha_{k-1})$, in which case restriction (2) does not hold, we can partition the computation time C_k into n (n is a positive integer) parts that satisfies $\frac{C_k}{n} \le C_{k-1} + 2(\alpha_k - \alpha_{k-1})$ (which again implies $\Delta C_{k,k-1} \le 2\Delta\alpha_{k,k-1}$). We can consider τ_k to consist of a set of n subtransactions: $\mathcal{T}_{\tau_k} = \{\tau_{k1}, \tau_{k2}, .., \tau_{kn}\}$, in which $\alpha_{ki} = \alpha_k$ and $C_{ki} = \frac{C_k}{n}$ $(1 \le i \le n)$. We denote $\mathcal{T}_{\tau_k} = \mathcal{P}(\tau_k)$. Let us substitute the set of transactions $\tau_{k1}, \tau_{k2}, .., \tau_{kn}$ for transaction τ_k and form a new set of transactions $\mathcal{T}_k = \{\tau_1, .., \tau_{k-1}, \tau_{k1}, \tau_{k2}, .., \tau_{kn}, \tau_{k+1}, .., \tau_m\}$. If we assign orders of transactions in $\mathcal{T}_k$ according to SVF and derive periods based on Eq. 1, it is easy to see that $D_{ki} \le D_k$. that is, $P_{ki} \ge P_k$.
2. *Partitioning of more than one transaction:* If there are multiple adjacent transactions that do not satisfy restriction (2), they are partitioned in the same way and the set of old transactions is transformed into a set of new transactions $\mathcal{T_P} = \{\tau_1, \tau_2, .., \tau_{m_p}\}$ $(m_p \ge m)$. Transactions in $\mathcal{T_P}$ now satisfy restriction (2), thus the optimal solution of transaction set $\mathcal{T_P}$ can be achieved by applying theorem 3.6.

Merge (denoted as $\mathcal{P}^{-1}$) is the inverse function of *Partition*. If $\mathcal{T}_k = \mathcal{P}(\tau_k)$, then $\tau_k = \mathcal{P}^{-1}(\mathcal{T}_k)$.

Let $U^{opt}_{\mathcal{T}}$ and $U^{opt}_{\mathcal{T}_k}$ denote the optimal solution of $\mathcal{T}$ and $\mathcal{T}_k$, respectively. It is obvious that

$$U^{opt}_{\mathcal{T}_k} = \sum_{i=1, i\neq k}^{m'} \frac{C_i}{P_i} + \sum_{j=1}^{n} \frac{\frac{C_k}{n}}{P_{kj}}. \tag{12}$$

As per Theorem 3.7, $U^{opt}_{\mathcal{T}_k} \le U^{opt}_{\mathcal{T}}$. Applying Theorem 3.7 repeatedly to $\mathcal{T}$, we know that the CPU utilization of the optimal solution of $\mathcal{T_P}$, $U^{opt}_{\mathcal{T_P}}$, satisfies

$$U^{opt}_{\mathcal{T_P}} \le U^{opt}_{\mathcal{T}}. \tag{13}$$

Theorem 3.8: Given a set of transactions $\mathcal{T}$ which satisfies restriction (1), let $U^{opt}_{\mathcal{T}}$, $U^{opt}_{\mathcal{T_P}}$, and $U_{\mathcal{S}^*}$ denote the CPU utilization of an optimal solution of $\mathcal{T}$, the optimal solution of $\mathcal{T_P}$, and the approximate solution $\mathcal{S}^*$ of $\mathcal{T}$ derived from *Shortest Validity First (SVF)*, respectively. The following inequality holds:

$$U_{\mathcal{S}^*} \ge U^{opt}_{\mathcal{T}} \ge U^{opt}_{\mathcal{T_P}}. \tag{14}$$

Proof: $U_{\mathcal{S}^*} \ge U^{opt}_{\mathcal{T}}$ because $U^{opt}_{\mathcal{T}}$ is the optimal CPU utilization of the same set of transactions. We know $U^{opt}_{\mathcal{T}} \ge U^{opt}_{\mathcal{T_P}}$ from Eq. 13. So the theorem follows. □

Definition 3.5: CPU utilization bound with respect to the optimal solution : Given a set of transactions $\mathcal{T} = \{\tau_1, \tau_2, .., \tau_m\}$ and its optimal CPU utilization $U^{opt}_{\mathcal{T}}$, the CPU utilization bound of any solution $\mathcal{S}$ with respect to its optimal solution, $\mathcal{B}_{\mathcal{S}}$, is defined as

$$\mathcal{B}_{\mathcal{S}} = U_{\mathcal{S}} - U^{opt}_{\mathcal{T}}, \tag{15}$$

where $U_{\mathcal{S}}$ is the CPU utilization of solution $\mathcal{S}$.

Theorem 3.9: Given a set of transactions $\mathcal{T} = \{\tau_1, \tau_2, .., \tau_m\}$ with $\alpha_1 \le \alpha_2 \le ... \le \alpha_m$, suppose that $\mathcal{T}$ satisfies restriction (1) but not restriction (2). $\mathcal{S}^*$ is a solution from the *SVF* algorithm. Assume that Ψ is a set of subscripts of all the transactions in $\mathcal{T}$ that are partitioned in a *partition-transformation* after which the resulting set of transactions $\mathcal{T_P}$ satisfies restriction (2). The CPU utilization bound of $\mathcal{S}^*$ with respect to the optimal solution of $\mathcal{T}$, $\mathcal{B}_{\mathcal{S}^*}$, satisfies

$$\mathcal{B}_{\mathcal{S}^*} \le 2 \sum_{k\in\Psi} (\frac{C_k}{\alpha_k})^2. \tag{16}$$

Proof sketch: Let $U_{\mathcal{T}}^{opt}$, $U_{\mathcal{T}_{\mathcal{P}}}^{opt}$, and $U_{\mathcal{S}^*}$ denote the CPU utilization of an optimal solution of $\mathcal{T}$, the optimal solution of $\mathcal{T}_{\mathcal{P}}$ partition-transformed from $\mathcal{T}$, and the solution $\mathcal{S}^*$ from the *SVF* algorithm, respectively. It follows from Theorem 3.8 that $\mathcal{B}_{\mathcal{S}^*} = U_{\mathcal{S}^*} - U_{\mathcal{T}}^{opt} \leq U_{\mathcal{S}^*} - U_{\mathcal{T}_{\mathcal{P}}}^{opt}$. It is proved in [14] that $U_{\mathcal{S}^*} - U_{\mathcal{T}_{\mathcal{P}}}^{opt} \leq 2\sum_{k\in\Psi}(\frac{C_k}{\alpha_k})^2$. □

Theorem 3.9 says that the CPU utilization from SVF is within $2\sum_{k\in\Psi}(\frac{C_k}{\alpha_k})^2$ of that of an optimal solution if restriction (1) holds. In many real applications, e.g., the avionics application [6] discussed later in the paper, sensor transaction computation time is in the range of milliseconds, validity interval length is in the range of hundreds of milliseconds and seconds. Thus $(\frac{C_k}{\alpha_k})^2$ for a sensor transaction τ_k is about $\frac{1}{10^6}$ to $\frac{1}{10^4}$. The number of transactions which may belong to the transaction set Ψ is usually very limited. Therefore, this bound is actually very small and can be ignored in many situations, thus SVF becomes a near optimal solution.

The optimal solution for the general case of *More-Less*, i.e., when both Restrictions (1) and (2) are relaxed, is left as an open issue. However, as we shall show in Section 5, SVF is a good heuristic solution even in these situations.

4 *More-Less* Application: Similarity-Based Load Adjustment

In this section, we consider the similarity-based load adjustment [6] as an application of *More-Less*. The basic idea of similarity-based load adjustment is to skip the executions of transaction instances which produce similar outputs. The approach taken in [6] is to modify the execution frequencies of transactions such that only one instance of a transaction is executed for multiple periods. As a result, system workload is reduced. View *r-serializability* [6] is a criterion used to justify the correctness of transactions. Readers are referred to [4, 6] for details of *similarity* and view *r-serializability*.

In similarity-based load adjustment, a *similarity bound* is derived for each data object based on application semantics. Two write events of the same data objects are *similar* if their sampling times differ by an amount of time no greater than the similarity bound. In other words, write events on the same data occurs within similarity bound are interchangeable as input to a read without adverse effects. Therefore, some write or read events can be dropped in order to reduce system load without affecting data temporal correctness. Here, validity interval length is replaced by similarity bound to constrain the arrival time of a transaction instance and finishing time of its next instance.

Update and *View* principles are proposed in [6] to adjust the system load. Their *update* principle is based on the *Half-Half* principle. Based on *More-Less*, we derive new *update* and *view* principles to reduce the system load even further.

Suppose sb_j is the similarity bound for data object X_j. Any two conflicting write events on X_j occur within sb_j are interchangeable as input to a read event due to similarity. Suppose P_j, P_j^H and P_j^M be the periods of transaction τ_j refreshing X_j before load adjustment, after load adjusted by *Half-Half*, and after load adjusted by *More-Less*, respectively. Let D_j^H and D_j^M be the deadlines of transaction τ_j after load adjusted by the *Half-Half* and *More-Less* principle, respectively.

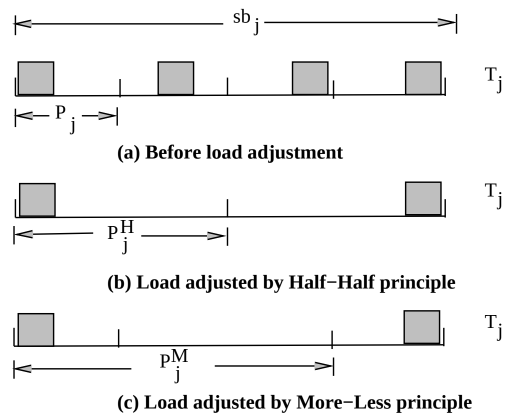

Figure 4. Update principles

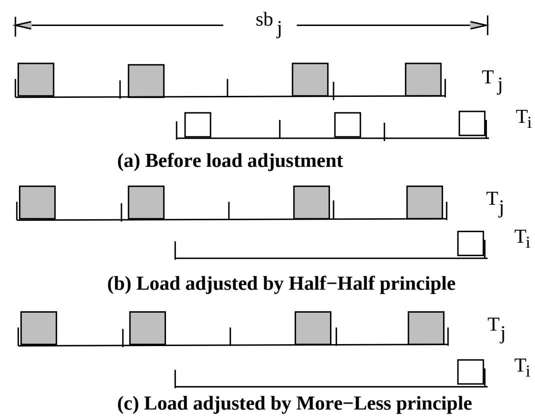

Figure 5. View principles

Update Principle: $P_j^M + D_j^M \leq sb_j$

In [6], the *Half-Half* principle is used to derive their *update* principle, which is $2P_j^H \leq sb_j$. However, our *update* principle derived from *More-Less* is $P_j^M + D_j^M \leq sb_j$. As shown in Figure 4, any read event will read from similar write events in both cases after load adjustment. In addition, because $D_j^M \leq \frac{sb_j}{2}$, we know that $P_j^M \geq \frac{sb_j}{2} \geq P_j^H$, which reduces the system utilization factor for τ_i by an amount of $\frac{C_j}{P_j^H} - \frac{C_j}{P_j^M}$ compared to the previous update principle. Therefore, update principle derived from *More-Less* reduces load even further without sacrificing similarity-based data correctness.

View Principle: $P_j^M + D_j^M + P_i^M \leq sb_j$

Suppose transaction τ_i with period P_i reads data object X_j. Let P_i^H and P_i^M denote the period of transaction τ_i adjusted by *Half-Half* and *More-Less*, respectively. *View* principle in [6] is defined as $2P_j^H + P_i^H \leq sb_j$. In contrast, our *view* principle from *More-Less* is defined

i	C_i	P_i	**Case 1**			**Case 2**			**Case 3**		
			P_i^M	D_i^M	P_i^H	P_i^M	D_i^M	P_i^H	P_i^M	D_i^M	P_i^H
1	1	3	12	3	6	9	3	6	3	3	3
2	2	5	5	5	5	10	10	10	15	15	15

Table 8. Parameters and results of example 4.1

as $P_j^M + D_j^M + P_i^M \leq sb_j$. As shown in Figure 5, $P_j^M + D_j^M + P_i^M$ is the maximum temporal distance among the write events which might be read by instances of τ_i and their representatives before and after load adjustment. Therefore, the *view* principle derived from the *More-Less* principle can guarantee similarity-based data correctness.

The following example clearly indicates that *update* and *view* principles derived from *More-Less* can reduce system load more than *update* and *view* principles from *Half-Half*.

Example 4.1: We use an example in [6] to illustrate the effectiveness of the *More-Less* principle. Suppose there are two periodic transactions τ_1 and τ_2 in a single processor environment. Their computation times and periods are given in Table 8. τ_1 periodically refreshes a data object X and τ_2 periodically reads the same data. The similarity bound sb_X of X is 22. According to *update* and *view* principles corresponding to the *More-Less* and *Half-Half* principles, the following inequalities must hold, respectively.

$$\begin{cases} P_1^M + D_1^M \leq 22 \\ P_1^M + D_1^M + P_2^M \leq 22 \end{cases} \tag{17}$$

$$\begin{cases} 2P_1^H \leq 22 \\ 2P_1^H + P_2^H \leq 22 \end{cases} \tag{18}$$

It is obvious that there are multiple solutions. Three different results after load adjustment are shown in Table 8. Let U_H and U_M denote the system CPU utilization after load adjustment based on the *Half-Half* and *More-Less* principles, respectively. In cases 1 and 2, $P_2^H = P_2^M = 5$ and $P_2^H = P_2^M = 10$, respectively. $U_H - U_M$, the difference in adjusted system load, is $\frac{1}{12}$ and $\frac{1}{18}$ in case 1 and 2, respectively. In case 3, $P_2^H = P_2^M = 15$, the system load adjusted from both principles are the same. This indicates that our update principle provides solutions with lower CPU utilization than the previous update principle. □

5 Experiments

In this section, experimental results are presented to quantitatively show that *More-Less* produces solutions with better schedulability and lower CPU utilization than the *Half-Half* principle. A set of update sensor transactions is generated randomly: computation time of a sensor transaction is uniformly generated from 5 to 15 milliseconds, and validity interval length of an object is uniformly generated from 4000 to 8000 milliseconds. These values are similar to the values used in the experiments of [6] and data presented in the study of air traffic control system in [8]. The number of sensor transactions are varied to change the workload in the system. For each data point presented in a figure, the experiments are run multiple times so that CPU utilizations shown have relative half-widths about the the mean of less than 5% at the 95% confidence interval.

The resulting CPU utilization generated from the *One-One*, *Half-Half* and *More-Less* with SVF ordering are presented in Figure 6. When the number of transactions is less than 200, the workload falls into the *restricted case*, i.e., restriction (1) is satisfied. This is because the sum of computation times of all the transactions is less than half of the minimum of all the validity interval lengths. It is observed that CPU utilization produced by *More-Less* is very close to that of *One-One*, and much less than that of the *Half-Half* principle. We would like to remind readers that *One-One* is used only as an artificial baseline – it does not guarantee the validity of temporal data. In this case, as we explained in Section 3.4, *More-Less* is very close to the optimal solution. This is clearly substantiated by the small difference in the CPU utilization between *One-One* and *More-Less*: CPU utilization of an optimal solution under *More-Less* should be between those for *One-One* and *More-Less*. When the number of transactions is more than 200, the workload falls into the *general case* because restriction (1) is *not* satisfied. In this case, we observe that CPU utilization of *More-Less* is still much less than that of *Half-Half*. However, the difference in CPU utilization of *One-One* and *More-Less* increases as system workload increases. The highest workload in our experiments is produced when the number of transactions is 375, and the corresponding CPU utilization under *One-One*, *Half-Half* and *More-Less* is about 65%, 130% and 92%, respectively. *Half-Half* can not produce a feasible solution when the number of transactions exceeds 300 because the corresponding CPU utilization exceeds 100%. But *More-Less* can still produce feasible solutions even when the number of transactions increases to 375.

In summary, when both the *Half-Half* and *More-Less* principles can be used to schedule a set of sensor update transactions, the *More-Less* principle can be used to produce solutions with much lower CPU utilization, thus more CPU capacity can be used by other transactions in the system. In addition, *More-Less* can be used to provide feasible solutions even when *Half-Half* can not be applied. In such situations, *More-Less* provides better *schedulability*.

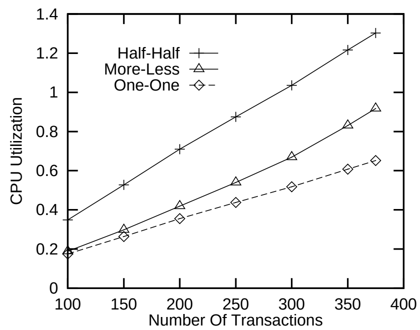

Figure 6. CPU utilizations from three principles

6 Conclusions

Database systems in which time validity intervals are associated with the data are discussed in [13, 12, 6, 5, 4, 2]. Such systems introduce the need to maintain data temporal consistency in addition to logical consistency.

A design methodology for guaranteeing end-to-end requirements of real-time systems is presented in [2]. Their approach guarantees end-to-end propagation delay, temporal input-sampling correlation, and allowable separation times between updated output values. However, their solution is based on the assumption that all the periodic tasks have harmonic periods. However, we do not make the assumption that all the periods are harmonic.

The work presented in our paper is also related to the work of [6]. but, as we showed, the schedulability of *More-Less* is better than *Half-Half* used in [6]. It is noted that *More-Less* guarantees a bound on the arrival time of a periodic transaction instance and the finishing time of the next instance. This is different from the *distance constrained scheduling*, a *dynamic* scheduling mechanism, which guarantees a bound of the finishing times of two consecutive instances of a task [3].

Very recently, we came across a paper by Burns and Davis [1] where SVF is proposed as a heuristic to determine periods. As we show in this paper, SVF in fact provides an optimal task assignment order when restrictions (1) and (2) are met and is a tight approximate ordering criterion when only restriction (1) is met.

In this paper, we examined the problem of deadline and period assignment in systems where data freshness should be guaranteed. *More-Less*, a novel principle based on the *validity constraint*, *deadline constraint* and *schedulability constraint* is proposed and analyzed. The solution for *More-Less* is constructed according to the *deadline monotonic* scheduling algorithm, which is the best algorithm for fixed priority scheduling. We proved the correctness of the *More-Less* principle, and its superiority to the traditional approach, the *Half-Half* principle. We further examined the issue of optimal assignment order under *More-Less* principle and found that *Shortest Validity First (SVF)* is an optimal order in situations in which both restrictions (1) and (2) hold. With the relaxation of restriction (2), we proved that SVF is an approximate solution within a certain bound of the optimal solutions. We showed, through both analysis and experiments, that this bound is tight in real world applications. We have also found in experiments that *More-Less* with SVF assignment order produces solutions with much better schedulability as well as lower CPU utilization than *Half-Half* even in general cases, i.e., when restriction (1) does not hold. However, the problem of searching for optimal assignment orders in the general case remains open.

References

[1] A. Burns and R. Davis, “Choosing task periods to minimise system utilisation in time triggered systems,” in *Information Processing Letters*, 58 (1996), pp. 223-229.

[2] R. Gerber, S. Hong and M. Saksena, “Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes,” *IEEE 15th Real-Time Systems Symposium*, December 1994.

[3] C. C. Han, K. J. Lin and J. W.-S. Liu, “Scheduling Jobs with Temporal Distance Constraints,” *Siam Journal of Computing*, Vol. 24, No. 5, pp. 1104 - 1121, October 1995.

[4] T. Kuo and A. K. Mok, “Real-Time Data Semantics and Similarity-Based Concurrency Control,” *IEEE 13th Real-Time Systems Symposium* , December 1992.

[5] T. Kuo and A. K. Mok, “SSP: a Semantics-Based Protocol for Real-Time Data Access,” *IEEE 14th Real-Time Systems Symposium* , December 1993.

[6] S. Ho, T. Kuo, and A. K. Mok, “Similarity-Based Load Adjustment for Static Real-Time Transaction Systems,” *18th Real-Time Systems Symposium*, 1997.

[7] C. L. Liu, and J. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *Journal of the ACM*, 20(1), 1973.

[8] Doug Locke, “Real-Time Databases: Real-World Requirements,” in *Real-Time Database Systems: Issues and Applications*, edited by Azer Bestavros, Kwei-Jay Lin and Sang H. Son, Kluwer Academic Publishers, pp. 83-91, 1997.

[9] J. Leung and J. Whitehead, “On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks,” *Performance Evaluation*, 2(1982), 237-250.

[10] K. Ramamritham, “Real-Time Databases,” *Distributed and Parallel Databases* 1(1993), pp. 199-226, 1993.

[11] K. Ramamritham, “Where Do Time Constraints Come From and Where Do They Go ?” International Journal of Database Management, Vol. 7, No. 2, Spring 1996, pp. 4-10.

[12] X. Song and J. W. S. Liu, “Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 5, pp. 786-796, October 1995.

[13] M. Xiong, R. M. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley, “Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics,” *IEEE 17th Real-Time Systems Symposium*, pp. 240-251, December 1996.

[14] M. Xiong, and K. Ramamritham, “Deriving Deadlines and Periods for Update Transactions in Real-Time Databases,” *Technical Report*, Computer Science Department, University of Massachusetts Amherst, 1999 (http://www-ccs.cs.umass.edu/rtdb/publications.html).