Abort-Oriented Concurrency Control for Real-Time Databases

Tei-Wei Kuo, *Member*, *IEEE*, Ming-Chung Liang, and LihChyun Shu, *Member*, *IEEE*

Abstract—There has been growing interest in the performance of transaction systems that have significant response time requirements. These requirements are usually specified as hard or soft deadlines on individual transactions and a concurrency control algorithm must attempt to meet the deadlines as well as preserve data consistency. This paper proposes a class of simple and efficient abort-oriented concurrency control algorithms in which the schedulability of a transaction system is improved by aborting transactions that introduce excessive blockings. We consider different levels of the aborting relationship among transactions and evaluate the impacts of the aborting relationship when the relationship is built in an online or offline fashion. We measure aborting overheads on a system running the LynxOS real time operating system. The strengths of the work are demonstrated by improving the worst-case schedulability of an avionics example [20], a satellite control system [7], and randomly generated transaction sets.

Index Terms—Real-time databases, concurrency control, transaction aborting, priority inversion.

1 Introduction

There has been growing interest in the performance of transaction systems that have significant response time requirements. These requirements are usually specified as hard or soft deadlines on individual transactions and a concurrency control algorithm must attempt to meet the deadlines as well as preserve data consistency.

A number of analytic and simulation studies on the performance of scheduling algorithms that meet the specified deadline requirements have been documented in the following literature, e.g., [3], [4], [5], [8], [9], [10], [11], [12], [13], [19], [21], [23], [24], [26], [28]. In these studies, some adopted conservative algorithms, e.g., [13], [26], to prevent any violation of data consistency from happening; some proposed optimistic algorithms, e.g., [9], [19], [23], [24], [28], which abort transactions whenever necessary, to maintain data consistency. In the meantime, various application semantics were explored to provide more flexibility in concurrency control of real-time data access [8], [10], [12], [21], [29]. There is always a trade-off in managing the blocking cost and aborting cost of transactions. Conservative algorithms are often self-restrained in the coordination of data access and seldom abort transactions. As a result, higher priority transactions often suffer from lengthy priority inversion. On the other hand, optimistic algorithms abort transactions whenever necessary to maintain data consistency. Lower priority transactions are often aborted repeatedly and miss their deadlines. Furthermore, application semantics, such as data similarity, may, in general, improve the consistency level of a transaction system, but they usually, at the same time, increase the maximum duration of priority inversion for transactions and reduce the schedulability of a transaction system in the worst case.

This paper explores the idea of transaction aborting in managing priority inversion. We provide a more precise mechanism in trading the aborting cost with the blocking cost of transactions. Different levels of aborting relationship among transactions are considered and the impacts of the aborting relationship are evaluated when the relationship is built in an online or offline fashion. The aborting overheads on a system running the LynxOS real time operating system were also measured. The contribution of this work is to provide a class of simple and efficient abort-oriented protocols and various ways in observing and managing the schedulability of a transaction system with respect to the aborting cost and the priority inversion problem. The strengths of the work are demonstrated by improving the worst-case schedulability of an avionics example [20], a satellite control system [7], and randomly generated transaction sets.

The rest of the paper is organized as follows: Section 2 summarizes the related work. Section 3 introduces the basic protocol and proves some important properties such as serializability preservation and maximum priority inversion time. Section 4 derives theorems and provides an analytic procedure to balance aborting cost and blocking cost. Section 5 extends the basic protocol into two of its variations. Section 6 provides the experimental results which compare the proposed protocols with other concurrency control algorithms. Some case studies are also provided. Section 7 is the conclusion.

- *T.-W. Kuo is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 106, ROC. E-mail: ktw@csie.ntu.edu.tw.*
- *M.C. Liang is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan 621, ROC.*
- *L. Shu is with the Department of Information Management, Chang Jung University, Tainan, Taiwan 711, ROC. E-mail: shulc@mail.cju.edu.tw.*

Manuscript received 15 Mar. 1999; revised 22 Aug. 2000; accepted 27 Nov. 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106024.

2 Related Work

In the past decades, there has been increasing demand for database systems with stringent response-time requirements. These requirements are usually specified as hard, soft, or firm deadlines on individual transactions. Missing the deadlines of *hard real-time transactions* may cause a catastrophe; missing the deadlines of *soft or firm real-time transactions* may only reduce system efficiency. *Firm real-time transactions* are aborted when they miss their deadlines, while a *soft real-time transaction* which misses its deadline may continue its execution if its execution still contributes value to the system.

A number of researchers have looked at the issues in both ensuring data consistency and guaranteeing timeliness of data accesses, e.g., [3], [9], [12], [13], [14], [19], [23], [24], [26], [27], [28]. The proposed algorithms, in general, can be classified as either conservative or optimisitic. Conservative algorithms [12], [26] prevent any violation of data consistency from happening. As a result, these algorithms are often self-restrained in the coordination of data access and seldom abort transactions. In particular, Sha, et al. [27] proposed the Read/Write Priority Ceiling Protocol by extending the well-known Priority Ceiling Protocol (PCP) [25] in incorporating the semantics of read and write locks. Lam et al. [13], [14] later revised the Read/Write Priority Ceiling Protocol by adopting the idea of dynamic adjustment of serializability order proposed by Lin and Son [19]. The schedulability of hard real-time read-only transactions is greatly improved.

On the other hand, optimistic algorithms [9], [19], [23], [24], [28] abort transactions whenever necessary to maintain data consistency. Haritsa et al. [9] and Abbott and Garcia-Molina [3] showed the superiority of the optimistic concurrency control protocol. Their performance metrics were based on the number of transactions that miss their deadlnes or were restarted. Nevertheless, their results did not quantify the abortion cost in calculating the schedulability of critical transactions. It should be noted that aborted transactions are often restarted and an abundance of system resources are wasted in servicing aborted transactions. Shu et al. [23], [24] proposed a series of abort-oriented protocols and demonstrated the strengths of transaction abortings in reducing priority inversion and in the improvement of the schedulability of critical transactions. The Abort Ceiling Protocol (ACP) [24] is the best among the protocols proposed in [23], [24] and is derived from PCP [25], [27]. In addition to the priority ceilings defined for data objects [25], [27], an abort ceiling is defined for each transaction to allow a higher-priority transaction to abort a lower-priority transaction which blocks the higher-priority transactions if the abort ceiling of the lower-priority transaction is smaller than the priority of the higher-priority transaction. A nontrivial mechanism is proposed to determine when and how to set the abort ceiling of each transaction. Takada and Sakamura [28] also proposed a concurrency control protocol which allows the switchings of the aborting status of transactions to better manage the blocking and aborting costs of transactions.

3 Basic Aborting Protocol (BAP)

3.1 Overview

The basic idea of the basic aborting protocol is that, when a higher priority transaction is blocked by a lower priority transaction due to resource competition, the higher priority transaction aborts the lower priority transaction if the lower priority transaction is abortable. If not, the higher priority transaction is blocked by the lower priority transaction. Whether a transaction is abortable is determined by an offline schedulability analysis described in Section 4.4. The underlying idea of the analysis is that, when a lower priority transaction introduces excessive blocking to a higher priority transaction such that the higher priority transaction may miss its deadline in the worst case, the lower priority transaction is abortable.

We assume that a transaction system consists of a fixed set of firm real-time transactions and every transaction has a fixed priority. A *transaction* is the template of its instances; a transaction instance is a sequence of read and write operations on data objects. An instance of a transaction is scheduled for every request of the transaction. There is no constraint in determining the priority of a transaction for the protocols proposed in this paper. However, for the schedulability analysis in Section 4.4, the rate monotonic priority assignment [16] is assumed. Transactions are classified as abortable or nonabortable in an offline fashion. (Please see Section 4.4.) Before a transaction can access a data object, the transaction must first obtain a lock on the semaphore that guards the data object. When a transaction terminates (commits or is aborted), it must release all of its locks. In other words, no transaction may hold a lock on any data object when it is initiated for a request. A main-memory-resident database is assumed. We are interested in the context of uniprocessor priority-driven preemptive scheduling. The collection of data objects which may be read or written by each transaction is known. We consider critical real-time systems with a well-defined workload, such as an avionics example [20] and a satellite control system [7], in which less than hundreds of data objects exist.

Transactions are required to adopt a delayed write procedure. For each data object updated by a transaction, the update is done in the local area of the transaction and the actual write of the data object is delayed until the commit time of the transaction. As a result, transactions do not release locks of semaphores until they commit or are aborted. The delay write procedure eases the aborting process and avoids cascading aborting.

Now, we will state our notation.

Notation:

- $\tau_{i,j}$ denotes the *j*th instance of transaction τ_i. p_i and c_i are the period and worst-case computation time of transaction τ_i, respectively. If transaction τ_i is aperiodic, p_i is the minimal separation time between its consecutive requests. When there is no ambiguity, we use the terms "transaction" and "transaction instance" interchangeably.
- $R_{i,j}$ denotes the *j*th request of transaction τ_i. A transaction instance is initiated for each request of transaction τ_i. We say that a request is aborted if its

corresponding transaction instance is aborted. Once a transaction instance is aborted, it may be restarted or terminated as required by the selected scheduling algorithm. When there is no ambiguity, $\tau_{i,j}$ denotes the transaction instance of request $R_{i,j}$, regardless of whether it is restarted.

- There is a unique semaphore S_i that guards every data object O_i. Before a transaction accesses a data object, the transaction must lock the guarding semaphore of the data object.
- The kth critical section of a transaction instance $\tau_{i,j}$ is denoted as $z_{i,j,k}$ and corresponds to the code segment between the kth locking operation and its corresponding unlocking operation. We assume in this paper that critical sections are properly nested.[1]

For the purposes of this paper, we will use the terms "resource" and "data object" interchangeably. (A data object is one kind of resource.) We will also use the terms "process" and "transaction" synonymously.

3.2 Definition

The Basic Aborting Protocol (BAP) is an integration of the two-phase locking (2PL) protocol, Priority Ceiling Protocol (PCP) [25], and a simple aborting algorithm.[2] As assumed in [25], we are interested in the context of uniprocessor priority-driven preemptive scheduling and every transaction has a fixed priority. The priority ceiling PL_i of each semaphore S_i is equal to the highest priority of transactions which may use S_i. We now present the definitions of BAP:

1. A transaction instance, which has the highest priority among all ready transaction instances, is assigned the processor. The transaction instance can preempt the execution of any transaction instance with a lower priority, whether or not the priorities are assigned or inherited. (Priority inheritance will be defined later.)
2. When a transaction instance $\tau_{i,j}$ attempts to lock a semaphore S_k that guards the data object O_k, the lock request will be granted if the priority of $\tau_{i,j}$ is higher than the priority ceilings of all semaphores currently locked by transaction instances other than $\tau_{i,j}$; otherwise, a rechecking procedure for the lock request is done as follows: If all of the transaction instances—other than $\tau_{i,j}$ that locked semaphores with priority ceilings higher than the priority of $\tau_{i,j}$—are abortable, then $\tau_{i,j}$ may abort all of the transaction instances that lock such semaphores and obtain the new lock. (When a transaction instance is aborted, it releases all of the locks it has.) Otherwise, $\tau_{i,j}$ will be blocked. Aborted transaction instances are assumed to restart immediately after their abortings. Note that such aborted transaction instances have lower priorities than $\tau_{i,j}$ does.[3] Let S^* be the semaphore with the highest priority ceiling of all semaphores currently locked by transaction instances other than $\tau_{i,j}$. If $\tau_{i,j}$ is blocked because of S^*, $\tau_{i,j}$ is said to be blocked by the transaction instance that locked S^*.
3. A transaction instance $\tau_{i,j}$ uses its assigned priority unless it locks some semaphores and blocks higher priority transaction instances. If a transaction instance blocks a higher priority transaction instance, it inherits the highest priority of the transaction instances blocked by $\tau_{i,j}$. When a transaction instance releases a semaphore, it resumes the priority it had at the point of obtaining the lock on the semaphore. When a transaction instance is aborted, all transaction instances which inherit its priority must reset their priorities according to the definition of priority inheritance. The priority inheritance is transitive.[4]
4. No transaction instance is allowed to obtain any new lock after it releases any locks.[5]

Apparently, BAP is exactly the same as PCP except that BAP offers higher priority transaction instances a chance to abort lower priority transaction instances and BAP requires transaction instances to lock semaphores in 2PL fashion. We refer readers to [25] for detailed definitions of PCP.

Example 1: A BAP Schedule. We illustrate BAP by an example. Suppose there are three transactions τ_H, τ_M, and τ_L in a single processor environment. Let τ_H, τ_M, and τ_L have computation requirements 5, 5, and 7, respectively, and periods 11, 19, and 22, respectively. Suppose τ_H is the only transaction using semaphore S_1 and transactions τ_M and τ_L share semaphore S_2. Let τ_H, τ_M, and τ_L have priorities 3, 2, and 1, respectively (3 is the highest priority level). Suppose transaction τ_L is abortable and transactions τ_H and τ_M are nonabortable. According to the Basic Aborting Protocol (BAP), semaphores S_1 and S_2 have priority ceilings as high as the priority levels of transactions τ_H and τ_M, respectively.

For comparison, let us first schedule these transactions according to BAP. As shown in Fig. 1, transaction τ_L locks semaphore S_2 and runs at its assigned priority level at time 1. When τ_M arrives at time 2, τ_M preempts τ_L. At time 3, τ_M tries to lock semaphore S_2 and the lock request results in the aborting and restarting of transaction τ_L because τ_L holds a semaphore (S_2) with a priority ceiling no less than the priority of τ_M and τ_L is abortable. Since the lock request of semaphore S_2 is granted, τ_M proceeds with its execution, but it is later preempted by transaction τ_H at time 5. At time 6, the lock request of semaphore S_1 issued by τ_H is granted because the priority of τ_H is higher than the priority ceiling of semaphore S_2, which is owned by transaction τ_M at that time. At time 10, τ_H commits and releases its lock on semaphore S_1 and τ_M resumes its execution. At time 12,

1. It is one of the assumptions of PCP to handle the priority inversion problem.

2. BAP is composed of "compatible" real-time scheduling algorithms, conventional concurrency control algorithms, and simple aborting algorithms. The definition of compatibility is removed here because of the purpose of this paper. We refer interested readers to [17] for details.

3. We will show later that $\tau_{i,j}$ will not abort more than one transaction instance and the rechecking procedure can be done extremely efficiently.

4. We shall show in Theorem 3 that no transaction instance τ scheduled by BAP directly inherits a priority from a transaction instance which is aborted before τ commits or is aborted.

5. It is a 2PL scheme.

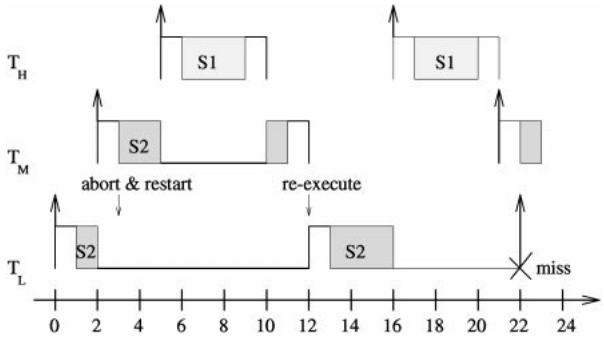

Fig. 1. A BAP schedule.

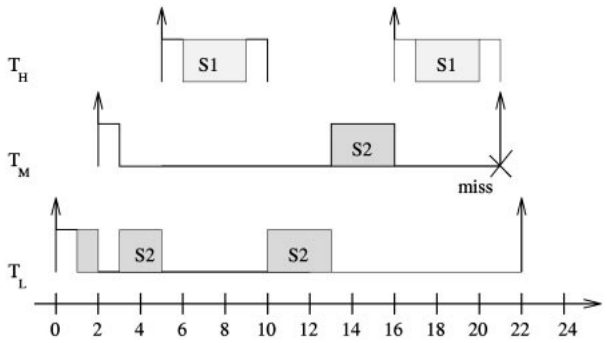

Fig. 2. A PCP + 2PL schedule.

τ_M commits and τ_L regains CPU and reexecutes as a new transaction. However, transactions τ_H and τ_M arrive again at times 16 and 21, respectively; τ_L receives only four units of computation time before its deadline at time 22, but τ_L needs seven units of computation time. τ_L misses its deadline at time 22.

For comparison, Fig. 2 shows a schedule of these transactions scheduled by the Priority Ceiling Protocol (PCP) [25] and transactions are required to access semaphores in a 2PL fashion. Clearly, the blocking of τ_M at time 3 (because the priority ceiling of semaphore S_2 is no less than the priority of τ_M) results in the deadline violation of τ_M at time 21.

This example demonstrates one of the goals in designing BAP, that a higher priority transaction may abort lower priority transactions to meet its timing constraints, and the schedulability improvement of a higher priority transaction is at the cost of the possible schedulability degradation of lower priority transactions.

We show in the next section that the Basic Aborting Protocol (BAP), which consists of 2PL, PCP, and a simple aborting algorithm, does preserve many important properties of 2PL and PCP.

3.3 Properties

The purpose of this section is to show important properties of BAP. We shall first show that BAP schedules are deadlock-free and serializable. We will then show that BAP prevents transitive blocking, which is important for system implementation because priority inheritance can be done with low run-time overheads [25]. In order to guarantee the schedulability of important firm real-time transactions, we will show that the worst-case number of priority inversions for each transaction is one and that a higher priority transaction instance can abort at most one lower priority transaction instance. Theorems of PCP in [25] are used to simplify the proofs of certain properties of BAP and to help in explaining the relationship between BAP and PCP. Note that BAP is a simple extension of PCP in allowing transaction aborting.

Lemma 1. *BAP prevents deadlocks.*

Proof. Without the aborting mechanism, BAP is simply a restricted PCP which requires transaction instances to lock semaphores in a 2PL fashion. Since the aborting mechanism of BAP does not introduce any new semaphore locks and PCP is deadlock-free, BAP is also deadlock-free. □

Theorem 1. *Schedules generated by BAP are logically correct (based on serializability).*

Proof. Because of the delayed write procedure, aborted transaction instances have no impact on the database. The data operations of aborted transaction instances such as reads and writes can be ignored when the logical correctness of BAP schedules are considered. Furthermore, transaction instances scheduled by BAP, regardless of whether they commit or are aborted, acquire locks in a 2PL fashion (and only access data objects after they lock their corresponding semaphores). All BAP schedules must be serializable. □

Note that BAP adopts a delayed write procedure in which the actual write of the data object is delayed until the commit time of the transaction instance. An aborting process of a transaction instance merely discards the results done by the aborted transaction instance and no cascading aborting is needed. The atomicity of transaction executions and the consistency of the database can be maintained at a very low cost. The delayed write procedure and the adopted 2PL scheme also help in maintaining the isolation property of the system. Since a memory-resident database is assumed and no failure is considered in this paper, the durability property of the system is not an issue here.

Lemma 2 [25]. *Suppose that the critical section $z_{j,n}$ of transaction instance J_j is preempted by transaction instance J_i which enters its critical section $z_{i,m}$. Under PCP, J_j cannot inherit a priority level which is higher than or equal to that of J_i until J_i completes its execution.*

Lemma 3. *Suppose that the critical section of transaction instance τ is preempted by transaction instance τ' which enters its critical section. Under BAP, transaction instance τ cannot inherit a priority level which is higher than or equal to that of τ' until τ' commits or is aborted.*

Proof. The correctness of the lemma follows from two things: Lemma 2 and the fact that no aborting of transaction instances will raise the priority level of any other transaction instances through priority inheritance and, without the aborting mechanism, BAP is simply a restricted PCP which requires transaction instances to lock semaphores in a 2PL fashion. □

Definition 1 [25]. *Transitive blocking is said to occur if a transaction instance is blocked by another transaction instance which, in turn, is blocked by the other transaction instance.*

Theorem 2. *BAP prevents transitive blocking.*

Proof. Suppose that there exists a transitive blocking in which transaction instance τ_1 blocks transaction instance τ_2 and transaction instance τ_2 blocks transaction instance τ_3. Due to the transitivity of priority inheritance, τ_1 inherits the priority of τ_3, which is assumed to be higher than that of τ_2. It contradicts Lemma 3, which shows that τ_1 cannot inherit a priority level higher than or equal to that of τ_2. □

Lemma 4 [25]. *A transaction instance J_H can be blocked by a lower priority transaction instance J_L under PCP only if J_L is executing within a critical section, when J_H is ready.*

Lemma 5. *If a lower priority transaction instance blocks or is aborted by a higher priority transaction instance under BAP, the lower priority transaction instance must receive at least one lock on a semaphore before the higher priority transaction instance is ready.*

Proof. A similar proof of Lemma 4 [25] is shown as follows: According to the definitions of BAP, a lower priority transaction instance τ_L can block or be aborted by a higher priority transaction instance τ_H if τ_L may directly or indirectly block τ_H. In either case, τ_L must own a semaphore lock. If τ_L does not own a semaphore lock when τ_H is ready, then τ_L can be preempted by τ_H and τ_L cannot block or be aborted by τ_H. □

Theorem 3. *No transaction instance τ scheduled by BAP directly or indirectly inherits a priority level from a transaction instance which is aborted before τ commits or is aborted.*

Proof. Suppose that a transaction instance τ directly inherits a higher priority from a transaction instance τ' which is later aborted. Because of Lemma 5, τ must have a lock on a semaphore S_i before τ' is ready and the lock later causes the direct priority inheritance of τ from τ'. Since τ' is later aborted, it must also have a lock on a semaphore S_j that later results in its aborting. Apparently, the priority ceiling of semaphore S_i must be less than the priority of τ'; otherwise, τ' cannot obtain a lock on semaphore S_j later. It contradicts the observation that the lock on semaphore S_i causes the priority inheritance of τ from τ'. Note that the priority ceiling of semaphore S_i must be no less than the priority of τ' to cause the priority inheritance. □

Theorem 4 [25]. *A transaction instance can be blocked for at most one critical section of at most one lower priority transaction instance under PCP.*

Theorem 5. *A transaction instance can experience at most one time of priority inversion under BAP.*

Proof. Because of Theorem 4 and the fact that PCP allows transaction instances to acquire semaphore locks in a 2PL fashion, transaction instances scheduled by PCP and acquiring semaphore locks in a 2PL fashion can be blocked for at most one critical section of at most one lower priority transaction instance. In the following, we shall show that the aborting mechanism of BAP does not falsify this proof.

The aborting mechanism of BAP simply offers a higher priority transaction instance a chance to abort a lower priority transaction instance. Lower priority transaction instances can never block a higher priority transaction instance after they leave their critical sections and no new blocking of any higher priority transaction instance is introduced because of abortings. Thus, the aborting mechanism should not increase the number of priority inversions of higher priority transaction instances. Based on these observations, a transaction instance can be blocked for at most one critical section of at most one lower priority transaction instance under BAP. □

Corollary 1. *Let a lower priority transaction instance τ_L block or be aborted by a higher priority transaction instance τ_H under BAP. τ_L must receive at least one lock on a semaphore that has a priority ceiling no less than the priority of τ_H before τ_H is ready.*

Proof. If τ_L can block or be aborted by τ_H, then τ_L must directly or indirectly block τ_H. Since Theorem 2 shows that BAP prevents transitive blocking, τ_L must own a semaphore lock with a priority ceiling no less than the priority of τ_H and directly block τ_H. □

Theorem 6. *A higher priority transaction instance can abort at most one lower priority transaction instance under BAP.*

Proof. Under BAP, a higher priority transaction instance aborts a lower priority transaction instance only when it tries to acquire a lock. Suppose transaction instance τ^H does abort more than one transaction instance and $\{\tau_1^L, \tau_2^L, \cdots, \tau_n^L\}$ is the set of lower priority transaction instances aborted by τ^H for $n > 1$. According to Corollary 1, each transaction instance τ_i^L must have a lock on a semaphore S_i before τ_H is ready, and the priority ceiling of semaphore S_i must be no less than the priority of τ_H. Suppose that semaphore S_i is the first semaphore locked by τ_i^L, which has a priority ceiling no less than the priority of τ_H. (Note that transaction instances under BAP access semaphores in a 2PL fashion, semaphore S_i shall not be released until τ_i^L is aborted by τ_H.) Apparently, transaction instance τ_i^L is ready before τ_H is ready and does not terminate until τ_H aborts it. Suppose that τ_1^L locks semaphore S_1 before τ_2^L locks semaphore S_2. Since the original priority of τ_2^L is lower than the priority of τ_H, the lock request of semaphore S_2 by τ_2^L must be rejected unless the priority of τ_2^L is raised to be higher than the priority ceiling of S_1 because of priority inheritance. However, the priority of τ_2^L cannot be raised so high unless it already locks a semaphore which blocks a transaction instance with a priority higher than the priority ceiling of S_1. It contradicts the assumption that semaphore S_i is the first semaphore locked by τ_i^L, which has a priority ceiling no less than the priority of τ_H. A contradiction can also be found in the same way if τ_1^L locks semaphore S_1 after τ_2^L locks semaphore S_2. The same arguments can be applied to any two transactions in $\{\tau_1^L, \tau_2^L, \cdots, \tau_n^L\}$ and a contradiction can always be found. Therefore, a higher priority

transaction instance can abort at most one lower priority transaction instance under BAP. □

4 Schedulability Analysis

4.1 Overview

Under BAP, transactions are classified as abortable or nonabortable in an offline fashion. The underlying idea of BAP is that, when a lower priority transaction may introduce excessive blocking to a higher priority transaction such that the higher priority transaction may miss its deadline in the worst case, the lower priority transaction is abortable. The purpose of this section is to provide a schedulability analysis procedure for a transaction system.

The amount of tolerable blocking time of a transaction depends on three factors: 1) the worst-case amount of computation times consumed by higher-priority transactions, 2) the aborting cost experienced by the transaction, and 3) the worst-case number of priority inversions for a transaction request when transaction aborting is possible. The worst-case amount of computation time consumed by higher-priority transactions is known because the system consists of a fixed set of transactions. The only two missing parts are the aborting cost experienced by a transaction and the worst-case number of priority inversions for a transaction request when transaction aborting is possible.

In the following sections, we shall first derive a formula to quantify the aborting cost for a transaction and then show the worst-case number of priority inversions for a transaction request when transaction aborting is possible. Note that more than one transaction, i.e., transaction instances, may be executed for a transaction request because of repeated transaction abortings. We need to know the worst-case number of priority inversions for all transaction instances executed for a single request. Finally, we shall propose a schedulability analysis procedure based on the idea of the Rate Monotonic Analysis theories [18], [22] to determine which transaction is abortable in an offline fashion.

4.2 Aborting Cost of BAP

Definition 2. *The direct aborting cost charged to a lower priority transaction instance τ by a higher priority transaction instance τ' is the CPU time that has been consumed by τ when τ is aborted by τ'.*

Definition 3. *A-cost$_{i,j}$ denotes the maximum direct aborting cost possibly charged to an instance of transaction τ_j by an instance of transaction τ_i.*

Definition 4. *Given a set of transactions $\tau_1, \tau_2, \cdots, \tau_n$ listed in the nondecreasing order of their priorities, the maximum aborting cost charged to an instance of transaction τ_j by an instance of transaction τ_i is α-cost$_{i,j} = \max(A\text{-}cost_{i,k})$, where $i < k \leq j$.*

Theorem 7. *A request of a lower priority transaction can be aborted at most once by a higher priority transaction within a period of the higher priority transaction.*

Proof. Let $\Pi_H = \{\tau_1^H, \tau_2^H, \cdots, \tau_m^H\}$ and $\Pi_L = \{\tau_1^L, \tau_2^L, \cdots, \tau_n^L\}$ be the sets of transaction instances instantiated for a request of a higher priority transaction τ_H and a request of a lower priority transaction τ_L, respectively. (Note that transaction instances τ_1^H and τ_1^L are the first instances instantiated for the requests of τ_H and τ_L, respectively. τ_i^H and τ_j^L for $i, j > 1$ are transaction instances restarted for the aborted transaction instances τ_{i-1}^H and τ_{j-1}^L, respectively. There is only one request for a transaction per period.) Suppose that transaction instances τ_i^H and τ_j^L are listed in the increasing order of their ready/restarting times.

Let τ_i^H abort τ_j^L for some index $i > 1$. As shown in Corollary 1, τ_j^L must have a lock on a semaphore S^* before τ_i^H is ready and the priority ceiling of semaphore S^* must be no less than the priority of τ_i^H. Since BAP restarts any aborted transaction instance immediately, τ_j^L must be in a critical section with a semaphore lock (not necessary on S^*) which has a priority ceiling no less than the priority of τ_k^H before τ_k^H is ready for $1 \leq k \leq i$; otherwise, τ_j^L can be preempted (and/or has no chance to lock S^*). By the definitions of BAP, τ_j^L would have been aborted by τ_1^H instead of τ_i^H. It contradicts the assumption that τ_i^H aborts τ_j^L for some index $i > 1$. We conclude that if τ_j^L is aborted by any transaction instance in Π_H, τ_j^L is only aborted by τ_1^H.

Furthermore, once a lower priority transaction instance is aborted, any of its corresponding restarted transaction instances would have no chance to gain CPU and lock semaphores unless higher priority transaction instances cease their existence. As shown in Lemma 5, a lower priority transaction instance must receive at least one lock on a semaphore before a higher priority transaction instance is ready if the lower priority transaction instance is aborted by the higher priority transaction instance. As a result, τ_1^H can only abort one transaction instance in Π_L. □

Let $\Pi = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of periodic transactions listed in the nonincreasing order of their priorities and $HPC_i = \{\tau_1, \tau_2, \cdots, \tau_{i-1}\}$ be the set of transactions with a priority no less than that of τ_i.

Lemma 6. *The worst-case aborting cost for a request of transaction τ_j between time 0 and time $t \leq p_j$ is at most $\sum_{\tau_i \in HPC_j} (\lceil \frac{t}{p_i} \rceil \times \alpha\text{-}cost_{i,j})$.*

Proof. Follows directly from the definition of α-cost$_{i,j}$ and Theorem 7. □

4.3 Blocking Cost of BAP

Theorem 8. *A request of a transaction can be blocked for at most one critical section of at most one lower priority transaction instance under BAP. In other words, the maximum number of priority inversion per transaction request is one.*

Proof. Suppose that transaction instance $\tau_{i,j}$ is initiated for a request $R_{i,j}$ of a transaction τ_i. Let $\tau_{i,j}$ be aborted and restarted for $n \geq 0$ times. For convenience, the transaction instances are renamed. Let τ_1^a be the transaction instance first initiated for request $R_{i,j}$ and $\{\tau_2^a, \tau_3^a, \cdots, \tau_{n+1}^a\}$ be the set of transaction instances restarted for request $R_{i,j}$. τ_k^as are listed in the order of their restarting times.

We shall show as follows that the total duration of priority inversion experienced by $\tau_1^a, \tau_2^a, \cdots,$ and τ_{n+1}^a is

at most one critical section of at most one lower priority transaction instance. Note that τ_1^a is ready when request $R_{i,j}$ occurs. Since BAP restarts an aborted transaction instance immediately after the aborting of the instance, τ_{i+1}^a is ready immediately after τ_i^a is aborted for $i \leq n$.

Let τ_i^a be the transaction instance blocked by a lower priority transaction instance τ^* with the largest index i. (Note that Theorem 5 shows that τ_i^a can be blocked for at most one critical section of at most one lower priority transaction instance.) Because of Corollary 1, τ^* must receive at least one lock on a semaphore that has a priority ceiling no less than the priority of τ_i^a before τ_i^a is ready. If τ^* has such a lock before τ_{i-1}^a is ready, τ_{i-1}^a is also blocked by the same semaphore lock. If τ^* receives this semaphore lock after τ_{i-1}^a is ready but before τ_i^a is ready, then τ^* must be in another critical section that locks another semaphore with a priority ceiling no less than the priority of τ_{i-1}^a; otherwise, τ_{i-1}^a can preempt τ^*. Thus, τ_{i-1}^a and τ_i^a can be considered as blocked by the larger critical section which properly contains the critical section blocking τ_i^a. (Note that BAP prevents transitive blocking and has all critical sections properly nested.) In either case, the total duration of priority inversion experienced by τ_{i-1}^a and τ_i^a is one critical section of one lower priority transaction instance. By repeating the same argument, we can show that the total duration of priority inversion experienced by $\tau_1^a, \tau_2^a, \cdots,$ and τ_i^a is one critical section of one lower priority transaction instance. Since τ_i^a is the transaction instance blocked by a lower priority transaction instance with the largest index i, a request of a transaction can be blocked for at most one critical section of at most one lower priority transaction instance. □

4.4 Schedulability Analysis Procedure

The purpose of this section is to provide a schedulability analysis procedure for a transaction system. The underlying idea is that if a lower priority transaction may introduce excessive blocking to a higher priority transaction such that the higher priority transaction may miss its deadline in the worst case, the lower priority transaction is made abortable by the higher priority transaction. Note that the proposed scheme depends on knowledge of the lengths of the execution time and critical sections of every transaction. If the lengths of the execution time and critical sections of any transaction is unknown, as in many traditional database systems, then the schedulability analysis in this paper cannot be done. However, we must emphasize that the schedulability guarantee of firm real-time transactions must rely on the above knowledge.

Let $\Pi = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of periodic transactions listed in the nondecreasing order of their priorities. $HPC_i = \{\tau_1, \tau_2, \cdots, \tau_{i-1}\}$ is a set of transactions with a priority no less than that of τ_i. Let b_i, d_i, p_i, and c_i be the worst case blocking time (from lower priority transactions), deadline, period, and worst case computation requirement of a transaction τ_i, respectively.

Lemma 7 [18], [22]. *Process τ_i in a set of periodic processes scheduled by a fixed priority-driven preemptive scheduling algorithm will always meet its deadline for all process phases if there exists a pair $(k, m) \in SP_i$ such that*

$$\sum_{j \in HPC_i} \left(c_j \left\lceil \frac{mp_k}{p_j} \right\rceil \right) + c_i + b_i \leq mp_k,$$

where b_i is the blocking time of τ_i and

$$SP_i = \left\{ (k, m) | 1 \leq k \leq i, m = 1, 2, \cdots, \left\lfloor \frac{p_i}{p_k} \right\rfloor \right\}.$$

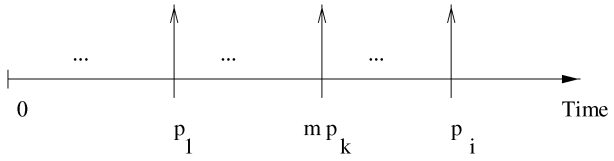

Fig. 3. Schedulability points for τ_i between 0 and p_i.

The phase of a process is the ready time of the first request of the process. SP_i is named the collection of schedulability points mp_k for τ_i, where mp_k is a time point when a new request of process execution, i.e., the $(m+1)$th request of process τ_k, occurs [18], [22] (Please see Fig. 3). Lemma 7 explores all time points no later than the deadline of process τ_i to see if all the computation requirements of existing process requests are satisfied. In other words, if there exists a time point no later than the deadline of τ_i , i.e., a schedulability point mp_k, such that all the computation requirements of existing process requests (including the request of τ_i) are satisfied, then process τ_i is schedulable.

Lemma 8. *A transaction τ_i scheduled by BAP will always meet its deadline for all process phases if there exists a pair $(k, m) \in SP_i$ such that*

$$\sum_{j \in HPC_i} \left(c_j \left\lceil \frac{mp_k}{p_j} \right\rceil \right) + c_i + b_i + ab_i \leq mp_k,$$

where b_i and ab_i are the worst case blocking cost and aborting cost of transaction τ_i, respectively, and

$$SP_i = \left\{ (k, m) | 1 \leq k \leq i, m = 1, 2, \cdots, \left\lfloor \frac{p_i}{p_k} \right\rfloor \right\}.$$

Each pair (k, m) represents a scheduling time point mp_k to test the schedulability of process τ_i.

Proof. The worst case aborting cost of transaction τ_i is modeled as an extra computation time of τ_i. (Note that the Rate Monotonic Analysis theories [18], [22] also model the worst case blocking time of transaction τ_i as an extra computation time of τ_i.) The correctness of the proof follows Lemma 7. □

Note that, when transactions are assigned priorities out of the rate monotonic order, the Rate Monotonic Analysis theories [22] model the execution times of higher priority but larger period transactions as the blocking times of lower priority but smaller period transactions. We refer interested readers to [18], [22] for details.

Schedulability Analysis Procedure: Let $\Pi = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of periodic transactions listed in the nondecreasing

order of their priorities. In the following procedure, we shall determine the maximum blocking time that transaction τ_i can tolerate (in an offline fashion).

Let $SP_i = \{(k,m)|1 \leq k \leq i, m = 1,2,\cdots,\lfloor \frac{p_i}{p_k} \rfloor\}$ and $MPK_i = \{mp_k|(k,m) \in SP_i\}$.

Lemma 8 shows that the maximum blocking time that transaction τ_i can tolerate is

$$MB_i = max_{t \in MPK_i}\left[t - \sum_{j \in HPC_i}\left(c_j\left\lceil\frac{t}{p_j}\right\rceil\right) - c_i - ab_i\right].$$

Based on Lemma 6, the maximum tolerable blocking time of transaction τ_i is

$$MB_i = max_{t \in MPK_i}\left[t - \sum_{j \in HPC_i}\left(c_j\left\lceil\frac{t}{p_j}\right\rceil\right) - c_i - \sum_{j \in HPC_i}\left(\left\lceil\frac{t}{P_j}\right\rceil \alpha\text{-}cost_{j,i}\right)\right].$$

Initially, all transactions are nonabortable. The maximum tolerable blocking time MB_i of transaction τ_i is calculated as defined.

1. i = 1.
2. If i > n, then stop.
3. If a transaction τ_j $(j > i)$ has a critical section that may lock a semaphore with a priority ceiling no less than transaction τ_i and the length of the critical section is larger than MB_i, then transaction τ_j becomes abortable. (See Theorem 8.) Repeat Step 3 until no such τ_j exists.
4. i = i + 1; go to Step 2.

When a lower priority transaction is abortable by a higher priority transaction, the worst-case blocking cost of the higher priority transaction may decrease, but at the cost of increasing the worst-case aborting cost of the lower priority transaction. Lower priority transactions may become unschedulable because of excessive aborting costs. On the other hand, higher priority transactions become schedulable because of lower blocking costs. Lemma 8 provides the best interpretation of the idea to trade the aborting costs of some transactions with the blocking costs of some other transactions.

5 Extensions of BAP

5.1 Motivation

Under the Basic Aborting Protocol (BAP), transactions are classified as abortable or nonabortable. A transaction is abortable if it potentially introduces excessive blocking to some higher priority transactions. Nevertheless, the excessive blocking that a lower priority transaction τ_L may introduce to a higher priority transaction τ_H may be tolerable to another higher priority transaction τ'_H. Under BAP, τ_L, which is abortable because of its potential excessive blocking to τ_H, may be unnecessarily aborted by τ'_H.

In this section, we will generalize the Basic Aborting Protocol in two ways: 1) Transaction abortings are managed in a more fine-grained fashion (see the Table-Driven Aborting Protocol). 2) Transaction abortings are determined in a totally online fashion (see the Dynamic Aborting Protocol).

5.2 Table-Driven Aborting Protocol (TAP)

The Table-Driven Aborting Protocol (TAP) is a generalization of the Basic Aborting Protocol (BAP) in the following way: Transactions are not simply abortable or nonabortable. Instead, every transaction τ_i has a set of transactions which it can abort. An instance of a lower priority transaction τ_j can be aborted by an instance of a higher priority transaction τ_i only if τ_j is in the set of transactions abortable by τ_i.

The aborting relationship can be realized as a table $AB[i,j]$: If $AB[i,j] = Yes$, an instance of a higher priority transaction τ_i can abort an instance of a lower priority transaction τ_j. If $AB[i,j] = No$, no instance of a higher priority transaction τ_i can abort an instance of a lower priority transaction τ_j. Whether an instance of a lower priority transaction τ_j is abortable by an instance of a higher priority transaction τ_i is determined by the maximum tolerable blocking time MB_i of τ_i and the maximum length CS_j of the critical section of τ_j which locks a semaphore with a priority ceiling no less than transaction τ_i. If $MB_i < CS_j$, $AB[i,j] = Yes$; otherwise, $AB[i,j] = No$. (See the schedulability analysis procedure in Section 4.4.)

The rest of the Table-Driven Aborting Protocol (TAP) is exactly the same as the Basic Aborting Protocol (BAP). Note that a data structure other than table/array can be used to implement the aborting relation $AB[i,j]$. The same properties and schedulability analysis procedure of BAP in Sections 3.3 and 4 also hold for TAP.

5.3 Dynamic Aborting Protocol (DAP)

The schedulability analysis procedure of the Basic Aborting Protocol (BAP) in Section 4.4 partitions transactions into two exclusive sets, either abortable or nonabortable, in an offline fashion. A lower priority transaction is abortable if it potentially introduces excessive blocking to a higher priority transaction and the blocking causes the deadline violation of the higher priority transaction. However, the schedulability analysis procedure based on Lemma 8 considers the worst case blocking time and the aborting cost of a transaction is also analyzed in the worst case situation, although the cost can change dynamically per period. (Note that the aborting cost of each transaction is tightly bounded.)

The technical question is how to relax the restrictions imposed by Lemma 8 on the estimations of blocking times and aborting costs. If the (maximum) blocking time that an instance of a transaction can tolerate is estimated dynamically and more precisely based on the current time and the current workload of the system, many unnecessary abortings of lower priority transaction instances can be avoided. However, Lemma 8 only offers a pseudopolynomial algorithm to estimate the maximum blocking time that any instance of a transaction can tolerate. We will use the following theorem to approximate Lemma 8 efficiently and closely. Based on our intensive simulation experiments, we observed that the Theorem 9 can approximate Lemma 8 quite efficiently and closely.

Note that every transaction in the system has a fixed priority and there is no constraint in determining the priority of a transaction for DAP and Theorem 9. However, if transactions are assigned priorities out of the rate monotonic order, Theorem 9 must model the execution times of higher priority but larger period transactions as the blocking times of lower priority but smaller period transactions, as done by the Rate Monotonic Analysis theories [22]. We refer interested readers to [18], [22] for details.

Theorem 9. *A transaction τ_i scheduled by BAP will always meet its deadline for all process phases if*

$$\sum_{j \in HPC_i} \left(\left\lceil \frac{d_i}{p_j} \right\rceil \times c_j \right) + c_i + b_i + ab_i \leq d_i,$$

where d_i, b_i, and ab_i are the deadline, worst case blocking cost, and worst case aborting cost of transaction τ_i, respectively.

Proof. The worst case aborting cost of transaction τ_i is modeled as an extra computation time of τ_i. (Note that the Rate Monotonic Analysis theories [18], [22] also models the worst case blocking time of transaction τ_i as an extra computation time of τ_i.) The correctness of the proof follows Lemma 7. □

The Dynamic Aborting Protocol (DAP) is an extension of the Basic Aborting Protocol (BAP) in the following way: At any time t, whether an instance of a higher priority transaction can abort an instance of a lower priority transaction depends on three conditions checked in an online fashion: 1) The lower priority transaction instance blocks the higher priority transaction instance on a semaphore lock at time t, 2) the lower priority transaction instance is abortable by the higher priority transaction, and 3) the maximum tolerable blocking time (see Theorem 9) of the higher priority transaction instance is less than the possible blocking time of the lower priority transaction instance at time t. (As in BAP, transaction abortings only happen when semaphore locks are requested.) The possible blocking time of the lower priority transaction instance can be estimated efficiently as the remaining computation time of the lower priority transaction instance or, more precisely, the remaining length of the critical section of the lower priority transaction instance that blocks the higher priority transaction instance at time t.

The rest of the Dynamic Aborting Protocol (DAP) is exactly the same as the Basic Aborting Protocol (BAP). The same properties of BAP in Sections 3.3 and 4 also hold for DAP. Because every transaction except the one with the highest priority is abortable in DAP, the maximum direct aborting cost $A\text{-}cost_{i,j}$ charged to an instance of transaction τ_j by an instance of transaction τ_i is usually not zero and the offline (worst case) schedulability analysis of a transaction system scheduled by DAP might be worse than that of the transaction system scheduled by BAP. However, we expect that DAP performs better than BAP does, on average.

5.4 Remark

The Basic Aborting Protocol (BAP) is a very simple protocol extended from PCP. Transactions are classified as abortable or nonabortable in an offline fashion, as described in Section 4.4. A transaction is abortable if it potentially introduces excessive blocking to some higher priority transactions. TAP is a generalization of BAP in maintaining the aborting relationship between any two transactions. DAP is another generalization of BAP in determining the aborting relationship between any two transactions in an online fashion. This paper considers different levels of the aborting relationship among transactions and evaluates the impacts of the aborting relationship when the relationship is built in an online or offline fashion. A more precise mechanism in trading blocking cost and aborting cost of transactions is also proposed.

BAP, TAP, and DAP all adopt a delayed write procedure in which the actual write of the data object is delayed until the commit time of the transaction. The delay write procedure eases the aborting process and avoids cascading aborting. DAP has a higher complexity than BAP and TAP because DAP must determine the aborting relationship of two conflicting transactions in an online fashion. However, we surmise that the complexity of DAP is very limited because the aborting relationship of conflicting transactions depends on the maximum tolerable blocking time of transactions and the maximum tolerable blocking time can be derived in a liner time (see Theorem 9).

Compared to the Abort Ceiling Protocol (ACP) [24], one of the best abort-oriented protocols, BAP and its variations are simpler in their protocol complexity and implementation. ACP and BAP both adopt the same delayed write procedure to ease the aborting process. The aborting overheads are the same. In addition to the priority ceilings defined for data objects in PCP and its variation [25], [27], ACP defines an abort ceiling for each transaction to allow a higher-priority transaction to abort a lower-priority transaction which blocks the higher-priority transactions if the abort ceiling of the lower-priority transaction is smaller than the priority of the higher-priority transaction. The abort ceiling of a transaction may be dynamically raised up with respective to the amount of time which the transaction has consumed. The abort ceilings of transactions are precalculated and stored in a table according to a nontrivial mechanism similar to the calculation of tolerable blocking time for BAP. BAP and its variation are simpler in the protocol design and implementation. Only a priority ceiling is needed for each data object (no abort ceiling and any other ceiling definition). When an access conflict occurs, only the aborting relationship of two conflicting transactions are checked, where the aborting relationship for BAP and TAP is even constructed in an offline fashion. No online determination of when to raise up the abort ceiling of a transaction is needed for BAP and its variations.

In summary, BAP represents a much simpler implementation of abort-oriented protocols, compared to ACP and other related protocols. We shall demonstrate the strengths of the work by improving the worst-case schedulability of an avionics example [20], a satellite control system [7], and randomly generated transaction sets in the next section.

6 Performance Evaluation

The strengths of the work are demonstrated by improving the worst-case schedulability of an avionics example [20], a satellite control system [7], and randomly generated transaction sets. We also provide the measurement of aborting overheads on a system running the LynxOS real time operating system.

6.1 Case Study

Two transaction systems based on an avionics example [20] and a satellite control system [7] were studied to demonstrate the strengths of the work by improving the worst-case schedulability of the transaction systems. The purpose of this section is to not only demonstrate the idea of transaction aborting in managing priority inversion for realistic workloads, but also to provide examples to illustrate the calculation of tolerable blocking time for each transaction and the idea of the schedulability analysis procedure proposed in Section 4.4. However, we shall emphasize that the realistic systems adopt various different mechanisms, such as the shrinking of critical sections and the adjustment of priority order to guarantee the schedulability of the systems. The adopted mechanisms are not related to the concurrency control issues addressed in this paper.

The avionics example has 18 periodic transactions, which define the software requirements for a Generic Avionics Platform similar to existing U.S. Navy/Marine Corps aircraft [20]. They belong to eight major subsystems: display, contact management, radar, navigation, tracking, weapon, built-in-test, and communication. Weapon_Release and Weapon_Aim in Table 2 are for the weapon system. Poll_Bus_Device and Radar_Tracking_Filter belong to the communication and radar systems, respectively. RWR_Tracking_Filter is for the contact management system. Table 2 shows the analysis results of the transaction system scheduled by the Basic Aborting Protocol (BAP). The schedulability analysis procedure (in Section 4.4) demonstrates in Table 2 that BAP can successfully schedule the first six most critical transactions, compared with the observation that only the most critical transaction, i.e., Timer_Interrupt, can be scheduled by the above pure locking PCP. Transactions in Table 2 are listed in the decreasing order of their priorities. Transaction Weapon_Release is listed second (not in a pure rate-monotonic order) to meet a 5ms jitter requirement. The jitter requirement is for ballistics computation to ensure target accuracy, based on vehicle trajectory, altitude, and attitude. Let the blocking time caused by any transaction be the execution time of the whole transaction. Unless stated, the deadline of a request is at the end of its period. Note that the cumulative processor utilization of transactions with priorities no less than that of transaction Weapon_Release is 6.605 percent. The improvement of the schedulability of critical transactions is achieved at the cost of aborting less critical transactions. Less critical transactions are more likely to miss their deadlines because of higher aborting costs. The tolerable blocking time and the aborting status of transactions were determined based on the procedure proposed in Section 4.4.

The second schedulability analysis is based on an abstraction of a sanitized version of the Olympus Attitude and Orbital Control Systems (AOCS) example, which is a satellite control system from [7]. The Olympus AOCS example [7] has 10 periodic transactions and four sporadic transactions, where Olympus is a three-axis-stabilized satellite launched in 1989 for TV-broadcasting and distance learning. The AOCS software operates/communicates with a telemetry & telecommand subsystem, an infrared earth sensor (IRES), a digital sun sensor, a rate gyro sensor, four reaction wheels, and thrusters during the normal operation mode. Transactions are assigned priorities according to the deadline monotonic priority assignment that assigns transaction priorities inversely proportional to their deadlines. In Table 3, transactions Bus_Interrupt and Telemetry_Response are modeled as periodic transactions. Even if transactions Bus_Interrupt and Read_Bus_IP (which share item Bus_IP_FIFO) can, indeed, execute concurrently without any interference from each other, all of the transactions except Bus_Interrupt remain unschedulable by the PCP protocol due to excessive blocking from transactions Process_IRES_data and Control_Law, where Control_Law implements the basic control laws and is responsible for maintaining the satellite Earth point, and Process_IRES_data is related to the processing of IRES data. Note that data received from the bus are encapsulated in a protected object. BAP makes transactions Process_IRES_data and Control_Law abortable and the schedulability of the Olympus AOCS is substantially improved. Table 3 shows that the first eight most critical transactions are schedulable by BAP.

The maximum blocking times of transactions in the avionics example [20] and the satellite control system [7] were recalculated based on Theorem 9 (for DAP), and they were almost the same as their correspondences in Tables 2 and 3.

6.2 Simulation Experiments

6.2.1 Measures and Methodologies

The experiments described in this section are meant to assess the capabilities of BAP, TAP, and DAP in reducing the number of deadline violations of critical transactions. We compared BAP, TAP, and DAP with the well-known PCP, Rate Monotonic Scheduling algorithm (RMS) [16], and Abort Ceiling Protocol (ACP) [24], where ACP is an abort-oriented protocol derived from PCP. Note that RMS represents an ideal situation where no data conflicting and priority inversion occurs.

The average number of data objects accessed by a transaction was five and the average number of transactions per transaction set ranged from four to 50. The periods of transactions were randomly chosen from 1 to $1,000$ time units and the CPU granularity was 0.01 time units. The same experiments were repeated on a database with 50, 100, 150, 200, and 1,000 data objects, respectively. Note that the proposed protocols aim at critical real-time systems with a well-defined workload, such as an avionics example [20] and a satellite control system [7], in which less than hundreds of data objects exist.

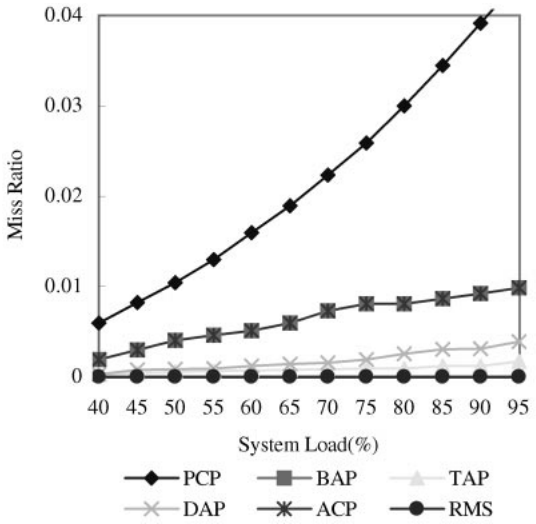

Fig. 4. Top 1/4 transactions, DB Size = 50.

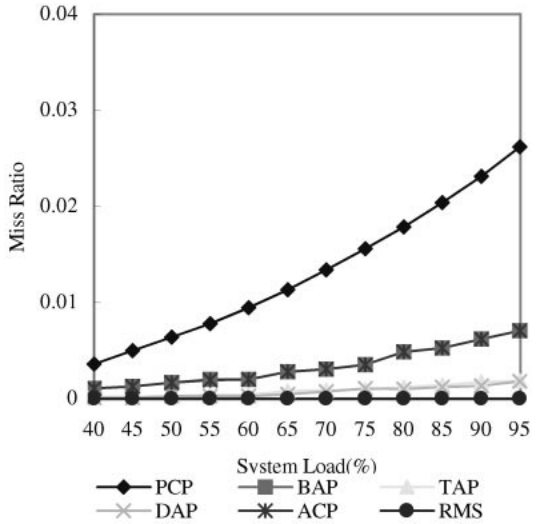

Fig. 6. Top 1/4 transactions, DB Size = 150.

The primary performance metric used is the ratio of requests that miss deadlines, referred to as *Miss Ratio*. Let num_i and $miss_i$ be the total number of transaction requests and deadline violations during an experiment, respectively. *Miss Ratio* is calculated as $\frac{miss_i}{num_i}$.

6.2.2 Experimental Results

Figs. 4, 5, 6, 7, and 8 summarize the miss ratio of the transactions which have the top $1/4$ highest priorities in their respective transaction sets. In general, abort-oriented protocols, especially TAP and DAP, resulted in a much lower number of deadline violations of critical or higher priority transactions than PCP did. As the size of the database decreased, more data conflicts were likely to occur. Figs. 4, 5, 6, 7, and 8 show the favoring of abort-oriented protocols when more data conflicts were possible. Fig. 9 summarizes the miss ratio of all of the transactions when the database had 150 data objects and the experimental results on a database of different sizes were very similar. The experimental results demonstrate the capability of abort-oriented protocols in reducing the number of deadline violations of critical transactions, though at the cost of increasing the number of deadline violations of less critical transactions. Note that the performance of BAP was about the same as that of the more complicated ACP in the experiments, and DAP and TAP greatly outperformed BAP.

6.3 Measurement of Aborting Overheads

The purpose of this section is to measure the major overheads in restarting a transaction. The measurement was done on a Pentium 133MHZ personal computer with 16MB of memory and LynxOS, where LynxOS is a Unix-like real-time operating system that provides deterministic, high performance real-time responsiveness for real-time applications [1]. The worst-case interrupt response time of LynxOS is $35\mu s$ on our experimental platform [2].

LynxOS library functions mutex_enter and mutex_exit were used to perform lock and unlock operations. The Unix library functions setjmp and longjmp [15] were used to

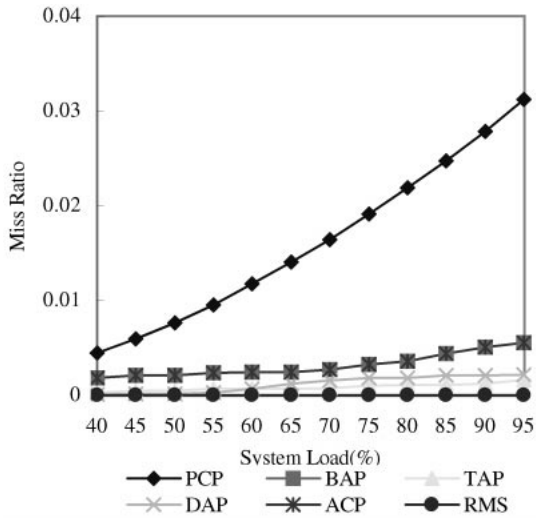

Fig. 5. Top 1/4 transactions, DB Size = 100.

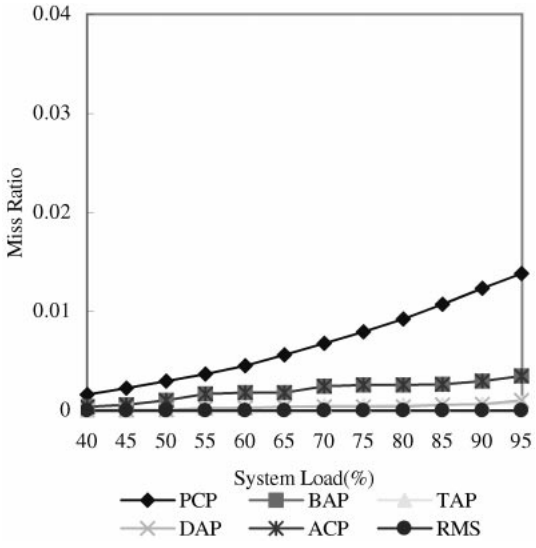

Fig. 7. Top 1/4 transactions, DB Size = 200.

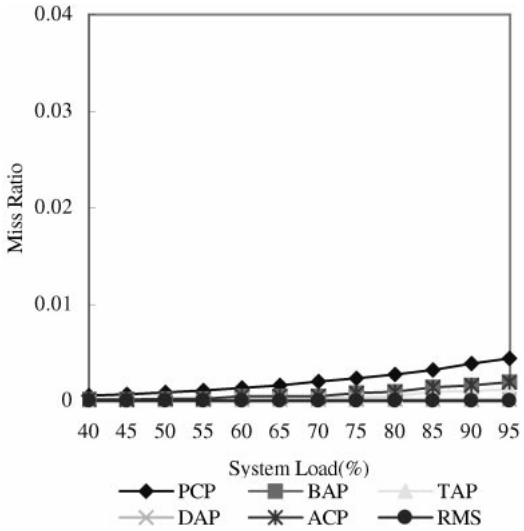

Fig. 8. Top 1/4 transactions, DB Size = 1,000.

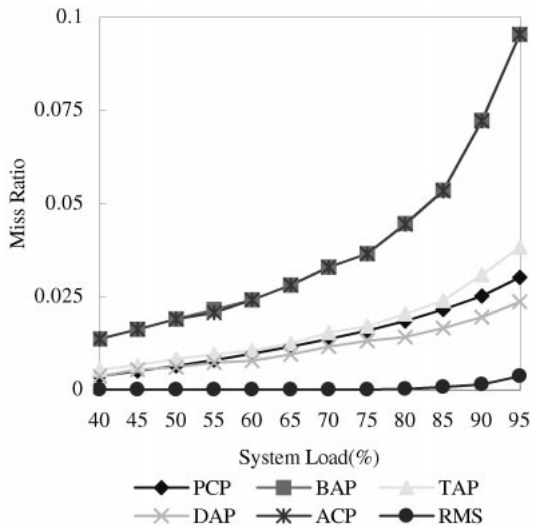

Fig. 9. The whole transaction set, DB Size = 150.

implement restart operations after transaction aborting. The setjmp operation, when invoked, saved the status of the execution stack and the return address in an environment variable. The longjmp operation restored the environment saved by setjmp and resumed the execution of a transaction at the place following the setjmp operation.

In the experiments, the setjmp and longjmp operations each took only $20\mu s$ and $13\mu s$, respectively. Since aborted transactions must release existing locks, the aborting cost should include the releasing of existing locks. As shown in Table 1, the time in releasing a lock is no more than $1\mu s$. The time needed to release locks was linearly proportional to the number of locks released. Note that this assumption was based on the duration of simple memory accesses for most operations where no context switch or operating system service was involved. Since this paper focuses on critical real-time systems with a well-defined workload, such as an avionics example [20] and a satellite control system [7], in which a transaction often accesses few data objects, we surmise that the restarting cost is very limited. The results further support the idea of having transaction aborting in a critical real-time database system.

7 Conclusion

This paper proposes a class of abort-oriented concurrency control protocols in which the schedulability of a transaction system is improved by aborting transactions that introduce excessive blockings. We consider different levels of the aborting relationship among transactions and

TABLE 1
Lock Release Times in LynxOS

locked resources	10	100	200	300	400	500	600	700	800	900	1000
release time (μs)	12	33	61	93	136	172	211	259	295	328	365

TABLE 2
Schedulability Analysis of BAP for the Generic Avionics Example (Unit: ms)

Transaction #	Period	Exec	Abortable ?	Aborting Cost (ab_i)	Schedulability Test (Use best mp_k for MB_i)	Tolerable Blocking
Timer_Interrupt	1	0.051	No	0	0.051	0.949
Weapon_Release	200	3.01	No	0	$\lceil 5/1 \rceil \times 0.051 + 3.01$	1.735
Radar_Tracking_Filter	25	2.03	Yes	2.03	$\lceil 25/1 \rceil \times 0.051 + \lceil 25/200 \rceil \times 3.01 + \lceil 25/25 \rceil \times 2.03 + abort$	16.655
RWR_Contact_Mgmt	25	5.03	Yes	10.06	$\lceil 25/1 \rceil \times 0.051 + \lceil 25/200 \rceil \times 3.01 + \lceil 25/25 \rceil \times 2.03 + \lceil 25/25 \rceil \times 5.03 + abort$	3.595
Poll_Bus_Device	40	1	No	15.09	$\lceil 40/1 \rceil \times 0.051 + \lceil 40/200 \rceil \times 3.01 + \lceil 40/25 \rceil \times 2.03 + \lceil 40/25 \rceil \times 5.03 + \lceil 40/40 \rceil \times 1 + abort$	4.74
Weapon_Aim	50	3.02	No	15.09	$\lceil 50/1 \rceil \times 0.051 + \lceil 50/200 \rceil \times 3.01 + \lceil 50/25 \rceil \times 2.03 + \lceil 50/25 \rceil \times 5.03 + \lceil 50/40 \rceil \times 1 + \lceil 50/50 \rceil \times 3.02 + abort$	10.21

TABLE 3
Schedulability Analysis of BAP for the Olympus AOCS Example (Unit: ms)

Transaction	Period	Exec	Deadline	Aborting ?	Aborting Cost (ab_i)	Schedulability Test (Use best mp_k for MB_i)	Tolerable Blocking
Bus_Interrupt	0.96	0.19	0.63	No	0	0.19	0.44
RTC	50	0.29	9	No	0	$\lceil 9/0.96 \rceil \times 0.19 + 0.29$	6.81
Read_Bus_IP	10	1.82	10	No	0	$\lceil 10/0.96 \rceil \times 0.19 + \lceil 10/50 \rceil \times 0.29 + 1.82$	5.8
Comand_Actuators	200	2.18	14	No	0	$\lceil 14/0.96 \rceil \times 0.19 + \lceil 14/50 \rceil \times 0.29 + \lceil 14/10 \rceil \times 1.82 + 2.18$	5.04
Request_DSS_Data	200	1.46	17	No	0	$\lceil 17/0.96 \rceil \times 0.19 + \lceil 17/50 \rceil \times 0.29 + \lceil 17/10 \rceil \times 1.82 + \lceil 17/200 \rceil \times 2.18 + 1.46$	6.01
Request_Wheel_Speeds	200	1.46	22	No	0	$\lceil 20/0.96 \rceil \times 0.19 + \lceil 20/50 \rceil \times 0.29 + \lceil 20/10 \rceil \times 1.82 + \lceil 20/200 \rceil \times 2.18 + \lceil 20/200 \rceil \times 1.46 + 1.46$	6.78
Request_IRES_data	100	1.46	24	No	0	$\lceil 24/0.96 \rceil \times 0.19 + \lceil 24/50 \rceil \times 0.29 + \lceil 24/10 \rceil \times 1.82 + \lceil 24/200 \rceil \times 2.18 + \lceil 24/200 \rceil \times 1.46 + \lceil 24/200 \rceil \times 1.46 + 1.46$	6.94
Telemetry_Response	62.5	3.24	30	No	0	$\lceil 30/0.96 \rceil \times 0.19 + \lceil 30/50 \rceil \times 0.29 + \lceil 30/10 \rceil \times 1.82 + \lceil 30/200 \rceil \times 2.18 + \lceil 30/200 \rceil \times 1.46 + \lceil 30/200 \rceil \times 1.46 + \lceil 30/100 \rceil \times 1.46 + 3.24$	8.37
Process_IRES_data	100	8.26	50	Yes	90.86	$\lceil 50/0.96 \rceil \times 0.19 + \lceil 50/50 \rceil \times 0.29 + \lceil 50/10 \rceil \times 1.82 + \lceil 50/200 \rceil \times 2.18 + \lceil 50/200 \rceil \times 1.46 + \lceil 50/200 \rceil \times 1.46 + \lceil 50/100 \rceil \times 1.46 + \lceil 50/62.5 \rceil \times 3.24 + 8.26 + abort$	miss

evaluate the impacts of the aborting relationship when the relationship is built in an online or offline fashion. A more precise mechanism in trading blocking cost and aborting cost of transactions is proposed. We demonstrate the strengths of the work by improving the worst-case schedulability of an avionics example [20], a satellite control system [7], and randomly generated transaction sets. We also measure aborting overheads on a system running the LynxOS real time operating system. We found that the schedulability of critical transactions can be greatly improved at the cost of slightly increasing the number of deadline violations of less critical transactions. When the database size is less than $1,000$ data objects, abort-oriented protocols have a great advantage over other concurrency control protocols, such as PCP. Abort-oriented protocols are thus very suitable to target system with a small or medium size of databases and a mixed collection of critical and noncritical transactions. When the maximum tolerable blocking time of a transaction can be estimated more precisely during run time (as the policy adopted by DAP), the schedulability of critical and less critical transactions can both be increased with only minor overheads in the determining of the aborting relationship among transactions.

We surmise that the implementation and run-time overhead of the Basic Aborting Protocol (BAP) is close to that of its component PCP. The priority inheritance of transactions scheduled by BAP can be done as efficiently as that of transactions scheduled by PCP because transaction abortings only occur when semaphores are requested and no transaction τ directly or indirectly inherits a priority level from a higher priority transaction that is aborted before τ commits or is aborted (See Theorem 3). Note that the schedulability analysis procedure shown in Section 4.4 is done before run time. The Table-Driven Aborting Protocol (TAP) and the Dynamic Aborting Protocol (DAP), though slightly more expensive than BAP, do show their superiority in managing the schedulability and aborting cost of transactions in the case studies and simulation experiments.

Trading the priority inversion times of critical transactions with the aborting costs of less critical transactions is a promising approach to boost the performance and stability of a transaction system. One future research direction of this work is to investigate the application semantics of a transaction system and its implications on transaction abortings and restartings. We also want to compare the performance of locked-based abort-oriented protocols against that of optimistic concurrency control protocols [6]. We believe that more research in classifying and analyzing real-time concurrency control algorithms and transaction systems to derive proper benchmarks may prove to be very rewarding.

ACKNOWLEDGMENTS

This research was supported in part by research grants from the National Science Council under Grants NSC85-2213-E-194-008 and NSC85-2213-E-309-001. This paper is an extended version of the paper that appeared in the *Proceedings of the Third International Workshop on Real-Time Computing Systems and Applications*. Ming-Chung Liang graduated from the Department of Computer Science and Information Engineering, National Chung Cheng University, under the supervision of Tei-Wei Kuo in 1996.

REFERENCES

[1] "LynxOS Frequently Asked Questions,"http://www.lynx.com, year?
[2] "LynxOS Real-Time Performance Analysis,"http://www.lynx.com/products/perfanal.html, year?
[3] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. 14th Very Large Data Bases Conf.,* pp. 1-12, 1988.
[4] A. Bestavros, "Timeliness via Speculation for Real-Time Databases," *Proc. IEEE 15th Real-Time Systems Symp.,* 1994.
[5] A. Bestavros, "Advances in Real-Time Database Systems Research," *Special Section on RTDB of ACM SIGMOD Record,* vol. 25, no. 1, Mar. 1996.
[6] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Reading, Mass.: Addison-Wesley, 1987.
[7] A. Burns, A.J. Wellings, C.M. Bailey, and E. Fyfe, "A Case Study in Hard Real-Time System Design and Implementation," YCS 190, Dept. of Computer Science, Univ. of York, 1993.
[8] L.B.C. Dipippo and V.F. Wolfe, "Object-Based Semantic Real-Time Concurrency Control," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1993.
[9] J.R. Haritsa, M.J. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proc. Ninth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems,* pp. 331-343, Apr. 1990.
[10] M.U. Kamath and K. Ramamritham, "Performance Characteristics of Epsilon Serializability with Hierarchical Inconsistency Bounds," *Proc. Int'l Conf. Data Eng.,* pp. 587-594, Apr. 1993.
[11] Y.-K. Kim and S.H. Son, "Supporting Predictability in Real-Time Database Systems," *Proc. IEEE 1996 Real-Time Technology and Applications Symp.,* 1996.
[12] T.-W. Kuo and A.K. Mok, "SSP: A Semantics-Based Protocol for Real-Time Data Access," *Proc. IEEE 14th Real-Time Systems Symp.,* Dec. 1993.
[13] K.W. Lam and S.L. Hung, "A Preemptive Transaction Scheduling Protocol for Controlling Priority Inversion," *Proc. Third Int'l Workshop Real-Time Computing Systems and Applications,* pp. 144-151, 1996.
[14] K.W. Lam, S.H. Son, V.C.S. Lee, and S.L. Hung, "Using Separate Algorithms to Process Read-Only Transactions in Real-Time Systems," *Proc. IEEE 19th Real-Time Systems Symp.,* Dec. 1998.
[15] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System.* Reading, Mass.: Addison-Wesley, 1989.
[16] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 46-61, Jan. 1973.
[17] M.-C. Liang, T.-W. Kuo, and L. Shu, "A Quantification of Aborting Effect for Real-Time Data Accesses," *IEEE Trans. Computers,* to appear.
[18] J.P. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proc. IEEE 10th Real-Time Systems Symp.,* Dec. 1989.
[19] Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proc. IEEE 11th Real-Time Systems Symp.,* Dec. 1990.
[20] C.D. Locke, D.R. Vogel, and T.J. Mesler, "Building a Predictable Avionics Platform in Ada: A Case Study," *Proc. IEEE 12th Real-Time Systems Symp.,* Dec. 1991.
[21] C.-S. Peng and K.-J. Lin, "A Semantic-Based Concurrency Control Protocol for Real-Time Transactions," *Proc. IEEE 1996 Real-Time Technology and Applications Symp.,* 1996.
[22] L. Sha, "Distributed Real-Time System Design Using Generalized Rate Monotonic Theory," lecture note, Software Eng. Inst., Carnegie-Mellon Univ., 1992.
[23] L. Shu and M. Young, "A Mixed Locking/Abort Protocol for Hard Real-Time Systems," *Proc. IEEE 11th Workshop Real-Time Operating Systems and Software,* pp. 102-106, May 1994.
[24] L. Shu, M. Young, and R. Rajkumar, "An Abort Ceiling Protocol for Controling Priority Inversion," *Proc. First Int'l Workshop Real-Time Computing Systems and Applications,* pp. 202-206, Dec. 1994.
[25] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Technical Report CMU-CS-87-181, Dept. of Computer Science, Carnegie-Mellon Univ., Nov. 1987, *IEEE Trans. Computers,* vol. 39, no. 9, Sept. 1990.
[26] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record,* vol. 17, no. 1, pp. 82-98, Mar. 1988.
[27] L. Sha, R. Rajkumer, S.H. Son, and C.-H. Chang, "A Real-Time Locking Protocol," *IEEE Trans. Computers,* vol. 40, no. 7, pp. 793-800, July 1991.
[28] H. Takada and K. Sakamura, "Real-Time Synchronization Protocols for Controlling Priority Inversion," *Proc. First Int'l Workshop Real-Time Computing Systems and Applications,* pp. 202-206, Dec. 1994.
[29] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley, "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *Proc. IEEE 17th Real-Time Systems Symp.,* pp. 240-249, 1996.

Tei-Wei Kuo received the BSE degree in computer science and information engineering from National Taiwan University in Taipei, Taiwan, in 1986. He received the MS and PhD degrees in computer sciences from the University of Texas at Austin in 1990 and 1994, respectively. He is currently an associate professor in the Department of Computer Science and Information Engineering of the National Taiwan University, Taiwan, Republic of China (ROC). He was an associate professor in the Department of Computer Science and Information Engineering of the National Chung Cheng University, Taiwan, ROC, from August 1994 to July 2000. His research interests include real-time databases, real-time process scheduling, real-time operating systems, and control systems. He is the program chair of IEEE Seventh Real-Time Technology and Applications Symposium, 2001, and has consulted for government and industry on problems in various real-time systems design. Dr. Kuo is a member of the IEEE and the IEEE Computer Society.

Ming-Chung Liang received the BSE degree in computer science and information engineering from Tatung Insititute of Technology, Taiwan, in 1994. He received the MS degree in computer science and information engineering from National Chung Cheng University in Taiwan in 1996. He is currently an engineer in the Pou Chen Group in Taiwan. His research interests include real-time database and real-time operating system.

LihChyun Shu received the PhD degree in computer sciences from Purdue University, West Lafayette, Indiana. He is an associate professor in the Department of Information Management at Chang Jung University in Tainan County, Taiwan, Republic of China. His research interests are real-time systems, databases, and data warehousing. He is a member of the IEEE and the IEEE Computer Society.