Schedulability and Performance Analysis of the Similarity Stack Protocol

Tei-Wei Kuo, *Senior Member*, *IEEE*, and Aloysius K. Mok, *Member*, *IEEE*

Abstract—We propose a class of real-time data access protocols called SSP (Similarity Stack Protocol). The correctness of SSP schedules is justified by the concept of similarity which allows different but sufficiently timely data to be used in a computation without adversely affecting the outcome. SSP schedules are deadlock-free, subject to limited blocking, and do not use locks. We give a schedulability bound for SSP and also report simulation results which show that SSP is especially useful for scheduling real-time data access on multiprocessor systems. Finally, we present a variation of SSP which can be implemented in an autonomous fashion in the sense that scheduling decisions can be made with local information only.

Index Terms—Real-time databases, concurrency control, real-time data access, data similarity, Δ-serializability, distributed data management, process scheduling.

1 Introduction

A number of researchers have studied the problem of meeting the timing requirements of applications while preserving data consistency. Not only must real-time transactions preserve properties such as data consistency, they must also meet the timing constraints that specify the recency of data. Conventional database systems are usually not used in real-time applications, owing to their poor performance and the lack of predictability in the timeliness of the transactions [5].

Theories and practice that apply to conventional databases do not directly apply to databases that deal with real-time data constantly on the change. This is mainly caused by the existence of an external environment that imposes new relations between the data objects in the database and the real-world objects they model [17], [20]. The validity of data in a real-time database usually degrades with time.

It has been recognized by a number of researchers that the notion of serializability is too strict a correctness criterion for concurrency control in accessing real-time data. In avionics software, for example, the precision of an answer to a query involving sensor data is often acceptable as long as the data is sufficiently timely, even though updates are sometimes performed in violation of the usual serializability criterion. Instead of read/write locks, a "cyclic executive" is routinely used in these applications to enforce a set of timing constraints on data access which in turn guarantees data consistency, the usual database locking protocols being too inefficient for the purpose.[1] Obviously, violation of serializability must be justified in the context of the semantics of the application domain. In [14], [15], we explored a weaker correctness criterion for concurrency control in real-time transactions, namely, the notion of "similarity."

Similarity is a binary relation on the domain of a data object. Every similarity relation is reflexive and symmetric, but not necessarily transitive. In a schedule, we say that two event instances are similar if they are of the same type (read/write) and access similar values of the same data object. The concept of similarity can be used to extend the usual correctness criteria for transaction scheduling. The main difficulty in these extensions is that similarity is in general not a transitive relation. This poses a limit on how events can be swapped in a schedule and complicates the usual notion of equivalence among schedules. By making use of the depth of data dependency in transaction schedules, we introduced in [14], [15] the notion of strong similarity and showed how it can be used to establish equivalence among schedules that permits a restricted form of event swapping. Our results justify the weaker notion of correctness that has been employed on an ad hoc basis in many real-time applications where the state information is "volatile" and the value of data depends on its timeliness. Similarity is also closely related to the important idea of imprecise computation in real-time systems [19] and to the idea of partial computation for databases [7]. An example of using the idea of similarity is to drop similar frames in video streams to reduce system workload while maintaining the quality of service in video playing.

Having thus provided a semantic foundation for accessing real-time data, an obvious question is how the notion of similarity relates to real-time transaction scheduling. As one might expect, the flexibility in scheduling read/write events introduced by similarity relaxes the scheduling problem since more write events can be deferred and some read events can be scheduled ahead of the latest write events. In this paper, we examine a class of scheduling policies which we call Similarity Stack Protocol (SSP) whose

1. Private communication with engineers at the Naval Weapons Center, China Lake, CA.

- *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.*
- *A.K. Mok is with the Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712. E-mail: mok@cs.utexas.edu.*

Manuscript received 1 June 1996; revised 1 Nov. 1997; accepted 12 July 1998. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 116545.

correctness is justified by similarity. The SSP protocol can significantly improve the schedulability of real-time transaction workloads on multiprocessor systems. Our performance studies are fundamentally different from past reports on analytic and simulation results in meeting transaction deadlines (e.g., [1], [10], [11], [24], [30]) in that unlike previous studies, we do not assume serializability as the correctness criterion for database consistency.

The rest of the paper is organized as follows: In Section 2, we summarize the similarity concept and the correctness criterion in [14], [15]. Section 3 describes the SSP scheduling policy based on the similarity concept and discusses some of its properties. Section 4 describes some simulation experiments and presents performance results which compare SSP with the well-known PCP and SRP protocols. Section 5 presents a modification of SSP which allows independent scheduling of transactions across processors under reasonable assumptions. Section 6 is the conclusion.

2 System Model, Similarity, and Correctness Criterion

In this section, we give a summary of the essential concepts in concurrency control of real-time transactions and the semantics of data similarity. For a more detailed discussion, we refer the reader to [14], [15].

2.1 Data Objects, Events, Transactions, and Schedules

The state of a real-time system is represented by the values of a collection of data objects. Each data object takes its value from its *domain*. *Events* are primitive data operations (read or write) and the same event may occur many times in a computation. Each occurrence of an event is associated with a time stamp whose value is the (wall-clock) time at which the instance of the event occurs. A transaction is a partial order of events and the same transaction may occur many times in a computation. Each occurrence of a transaction is called an instance of the transaction. To distinguish between a transaction and an instance of it, we shall use the notation $\tau_{i,j}$ to denote the jth instance of transaction τ_i. The view of a transaction instance is a vector of data object values such that the ith component is the value read by the ith read event instance of the transaction instance [22].

A *schedule* over a set of transactions is a partial order of event instances such that each event instance is issued by one transaction instance. The ordering of event instances in a schedule must be consistent with the event ordering as specified by the transaction set. In a real-time computation, the partial ordering of event instances in a schedule is induced by the time stamps of event instances at different sites. A *serial schedule* is a sequence of transaction instances (i.e., a schedule in which the transaction instances are totally ordered).

2.2 Similarity, Strong Similarity, and Correctness of Schedules

2.2.1 Similarity

The value of a data object that models an entity in the real world cannot in general be updated continuously to perfectly track the dynamics of the real-world entity. The time needed to perform an update alone necessarily introduces a time delay which means that the value of a data object cannot be instantaneously the same as the corresponding real-world entity. Fortunately, it is often unnecessary for data values to be perfectly up-to-date or precise to be useful. In particular, data values of a data object that are slightly different are often interchangeable as read data for transactions. This observation underlies the concept of similarity among data values.

Similarity is a binary relation on the domain of a data object. Every similarity relation is reflexive and symmetric, but not necessarily transitive. Different transactions can have different similarity relations on the same data object domain. *Two views of a transaction are similar* iff every read event in both views uses similar values with respect to the transaction. We say that *two values of a data object are similar* if all transactions which may read them consider them as similar. In a schedule, we say that two *event instances are similar* if they are of the same type and access similar values of the same data object. We say that *two database states are similar* if the corresponding values of every data object in the two states are similar.

A minimal restriction on the similarity relation that makes it interesting for concurrency control is the requirement that it is preserved by every transaction, i.e., if a transaction T maps database state s to state t and state s' to t', then t and t' are similar if s and s' are similar. We say that a similarity relation is *regular* if it is preserved by all transactions. We are interested in regular similarity relations only.

2.2.2 Strong Similarity

The definition of regular similarity only requires a similarity relation to be preserved by every transaction, so that the input value of a transaction can be swapped with another in a schedule if the two values are related by a regular similarity relation. Unless a similarity relation is also transitive, in which case it is an equivalence relation, it is, in general, incorrect to swap events an arbitrary number of times in a schedule. For example, let v_1, v_2, and v_3 be three values of a data object such that v_1 and v_2 are similar, as are v_2 and v_3. A transaction instance reading v_1 as input will produce similar output as one that reads v_2 as input. Likewise, the same transaction reading v_2 as input will produce similar output as one that reads v_3 as input. However, there is no guarantee that the output of the transaction reading v_1 as input will be similar to one reading v_3 as input since v_1 and v_3 may not be related under the regular similarity relation. Swapping events two or more times in a schedule may result in a transaction reading a value that is not similar to the input value before event swapping and, hence, the output of the transaction may not be similar to that before swapping.

The notion of strong similarity was introduced in [14], [15] which has the property that swapping similar events in a schedule will always preserve similarity in the output. This notion is motivated by the observation that the state information of many real-time systems is "volatile," i.e., they are designed in such a way that system state is determined completely by the history of the recent past,

e.g., the velocity and acceleration of a vehicle are computed from the last several values of the vehicle's position from the position sensor. Unless events in a schedule may be swapped in such a way that a transaction reads a value that is derived from the composition of a long chain of transactions that extends way into the past, a suitable similarity relation may be chosen such that output similarity is preserved by limiting the "distance" between inputs that may be read by a transaction before and after swapping similar events in a schedule. For the purpose of this paper, it suffices to note that, if two events in a schedule are strongly similar (i.e., they are either both writes or both reads, and the two data values involved are strongly similar), then they can always be swapped in a schedule without violating data consistency requirements.

2.2.3 Correctness of Schedules

Corresponding to the notions of final-state serializability, view serializability, and conflict serializability in conventional database theory, we introduced in [14], [15] the notions of final-state, view, and conflict Δ-serializability. In this paper, we adopt view Δ-serializability as the correctness criterion for consistency management since, without specifying input/output transactions, the notion of a final state is incompatible with a real-time system. Other correctness criteria have been proposed for different purposes and application areas [6], [8], [9], [17], [26]. For a comparison, interested readers are referred to [14], [15]. Below, we give the definition of view Δ-serializability.

Definition 1 [15], [16] View Similar. *Two schedules are* view-similar *iff*

1. *They are over the same set of transaction instances.*
2. *They transform (strongly) similar initial database states into similar database states under any interpretation.*
3. *Every transaction instance has similar views in both schedules for any initial state and under any interpretation.*

It is clear that, if a schedule is view-equivalent to another schedule, then it is view-similar to that schedule, but the converse may not hold. Note that the view-similarity relation between schedules is reflexive and symmetric but not necessarily transitive. A schedule is view Δ-serializable iff it is view-similar to a serial schedule.

Notice that similarity bounds (relations) are used to justify the logical correctness of schedules, while "real-time" consistency constraints such as *external consistency*, *temporal consistency*, and *dynamic consistency* explored in the literature, e.g., [14], [15], [17], [20], [30], are used to measure the timeliness of real-time data. In other words, similarity relations consider values of data objects, whereas the consistency constraints mentioned above consider the time tags (ages) of data values, which are often enforced by assigning proper timing requirements to processes. For more detailed discussion, we refer readers to [14], [15], [17], [20], [30].

Suppose two schedules π and π' have the same event set E, Δ and $\Delta^{\#}$ are a strong similarity relation and a regular similarity relation for both π and π'. We say that π' is a *derived* schedule of π if for any read event that appears in π and π', the two corresponding write events in π and π' read by the read event are strongly similar in π and the last write events which update the same data object in π and π' are strongly similar in π.

We need the following theorem to justify the correctness of the scheduling policies to be introduced in the next section.

Theorem 1 [15], [16]. *Suppose two schedules π and π' have the same event set E, Δ and $\Delta^{\#}$ are, respectively, a strong similarity relation and a regular similarity relation for both π and π'. If π' is a derived schedule of π, then π and π' are view-similar under $\Delta^{\#}$, i.e., π and π' transform similar states (under Δ) into similar states (under $\Delta^{\#}$).*

3 SSP (Similarity Stack Protocol)

Similarity is an inherently application-dependent concept and we expect the application engineer to define it for specific applications. In many real-time applications, it is often acceptable to use an older value of a sensor as input to a calculation, instead of waiting for a more up-to-date value. This is possible because the physics of the application may be such that changes in sensor reading over a short interval of time are so small as to be insignificant to the calculation. This observation provides us with the needed connection between similarity and timing constraints governing data access.

Specifically, we assume that the application semantics allows us to derive a *similarity bound* for each data object such that two write events on the data object must be strongly similar if their time-stamps differ by an amount no greater than the similarity bound, i.e., all instances of write events on the same object that occur in any interval shorter than the similarity bound can be swapped in the (untimed) schedule without violating consistency requirements. Notice that the existence of a similarity bound does not imply that the similarity relation is transitive since event swapping is based on (wall-clock) time values and not on the relative positions of events in a schedule. The existence of a similarity bound provides more freedom in event scheduling, whereas two conflicting transactions must be totally ordered to preserve serializability; they need not be if their conflicting events occur in an interval shorter than the similarity bound and can therefore be executed concurrently. To take advantage of the expanded concurrency, we introduce the Similarity Stack Protocol (SSP).

We shall not concern ourselves with the derivation of similarity bounds in this paper. We assume that sufficiently stringent timing constraints are imposed on transactions that update data objects, and the timing constraints are to be enforced by the real-time OS schedulers. There are two questions of interest to us: 1) How do we ensure that our real-time data access protocol does indeed produce correct schedules *vis-a-vis* view Δ-serializability? 2) How efficient is our protocol in meeting application timing constraints? Let us first describe the SSP protocol.

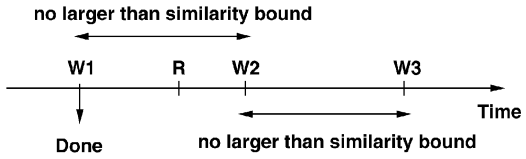


Fig. 1. Similarity of conflicting events.

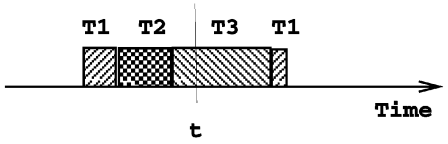


Fig. 2. A timing diagram for processor 1.

3.1 The Basic Strategy

The basic strategy of the SSP protocol can be summarized as follows: Transactions are normally scheduled according to their priorities which can be dynamic (e.g., earliest-deadline-first) or static (e.g., as determined by the Rate Monotonic Assignment algorithm), with the provision that transaction execution follows the stack discipline, i.e., if transaction B starts after transaction A, then A cannot resume until after B finishes. However, no transaction is allowed to start execution if it conflicts with another transaction that has already started but not committed such that the conflicting read/write events may not be arbitrarily swapped under the similarity relation in the following way.

Suppose two events e_1 and e_2 conflict with each other. Let e_1 and e_2 be the write events w_2 and w_3, respectively. If their write values are similar under the similarity bound as shown in Fig. 1, these two events are similar and it does not matter the result of which write is read by subsequent read events. Suppose e_1 and e_2 are, respectively, the write event w_2 and the read event r in Fig. 1. For their relative ordering to be unimportant, there must exist an earlier write event whose write value is similar to the write value of w_2 under the similarity bound. If this is the case, as is shown in Fig. 1, then it does not matter which write value the read event r reads. The same argument applies to the case where e_1 and e_2 are a read event and a write event, respectively.

Conflicting transactions cannot block one another as long as their event conflicts can be resolved by appealing to the similarity bound as shown above. To prevent deadlocks and unlimited blocking, however, the SSP protocol does not allow a transaction to start before all higher-priority transactions have completed.

3.2 Data Structure and Mechanism

To increase concurrency, we adopt the atomic data set concept of [24] and partition the transactions into disjoint subsets which we call *interactive sets* such that no two transactions from different interactive sets may conflict. However, we do not require transactions in an interactive set to run exclusively on one processor. Every *j*th interactive set on each *i*th processor is associated with two variables:

- $rn_{i,j}$ denotes the number of scheduled but uncommitted transaction instances in the *j*th interactive set on the *i*th processor.
- $accu_{i,j}$ denotes the sum of the computation time of transaction instances scheduled on the *i*th processor since the time at which the earliest scheduled but uncommitted transaction instance in the *j*th interactive set on the *i*th processor was first scheduled.

For example, suppose π is a schedule of three transactions τ_1, τ_2, and τ_3 on one processor as shown in Fig. 2. Let τ_1 and τ_2 be in the first interactive set and τ_3 be in the second interactive set. Suppose τ_1, τ_2, and τ_3 have computation requirements 1, 2, and 3, respectively. As shown in Fig. 2, τ_3 which starts execution after τ_2 committed is running at time t. Since τ_1 is preempted by τ_2 and has not committed, there is one scheduled but uncommitted transaction instance in the first interactive set on the first processor. Therefore, $rn_{1,1} = 1$ and $accu_{1,1} = 6$ at time t according to the above definitions.

Conceptually, we shall assume the existence of a stack for each processor. We call it the *preemption stack* of the processor. When a transaction instance is scheduled on a processor, it is pushed onto the top of the preemption stack. At any time, the transaction instance at the top of a preemption stack is the one being executed on the processor. A transaction instance preempts another transaction instance by being pushed on top of the latter on the preemption stack. When a transaction instance commits, it is popped off from the top of its preemption stack. Intuitively, $accu_{i,j}$ is a measure of the *temporal depth* of the *i*th stack for the *j*th interactive set.

The key idea of the SSP protocol is to use the preemption stacks to restrict the maximum time interval spanning two conflicting transaction instances that may overlap in their execution. Since the preemption stack contains all the transactions that have started execution but not committed, enforcing a bound on the temporal depth of the preemption stacks achieves the desired effect. For each preemption stack, we shall use a different bound for every interactive set, which we called the *recency bound* for the interactive set. We now determine the relationship between the recency bounds and the similarity bounds of the data objects.

Suppose sb_x is a similarity bound for data object x so that any two writes on x within an interval shorter than sb_x are interchangeable and strongly similar. We define a *W-recency bound* ω_x which is the upper bound to be enforced by the SSP protocol on the temporal distance between two write events on x which belong to overlapping transaction instances. Obviously, the choice of ω_x is constrained by $\omega_x \leq sb_x$. We also define a *R-recency bound* δ_x which is the upper bound to be enforced by the SSP protocol on the temporal distance between a read and a write event on x which belong to overlapping transaction instances. Suppose the read and write events in question are r_x and w_x, and r_x occurs before w_x. If we want to swap r_x and w_x without violating the correctness criterion, the write event, call it w'_x that r_x reads from must be sufficiently close to w_x so that these two write events are strongly similar (see Fig. 1). Specifically, if there is an update transaction on x which is scheduled at least once every p_x time units, then the two write events are strongly similar if the distance between the two write events w_x and w'_x is at most sb_x and $sb_x \geq 2p_x$. (Here, we assume that, in the absence of additional

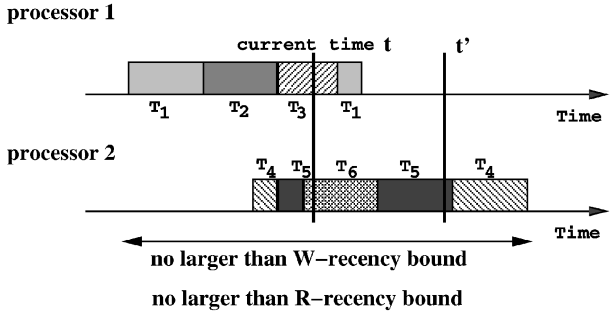


Fig. 3 Scheduling mechanism.

application information, the update transaction may actually perform the write operation on x anywhere inside a period of p_x time units and, hence, a read event may read from a write event almost $2p_x$ time units away.) Since we assume that the maximum distance between a read event r_x and the write event w'_x it reads from is $2p_x$, we have $sb_x \geq \delta_x + 2p_x$. Hence, δ_x is constrained by $\delta_x \leq sb_x - 2p_x$.

For the jth interactive set of transaction, we assign a W-recency bound ω_j and a R-recency bound δ_j which are the minimum of, respectively, the W-recency bounds and R-recency bounds of objects accessed by transactions in the jth interactive set. The basic scheduling problem is to enforce these bounds for every interactive set. This can be done simply by blocking any transaction whose execution may cause the sum $accu_{i,j} + accu_{k,j}$ to exceed ω_j or δ_j for some i and k. To see why this works, notice that $accu_{i,j} + accu_{k,j}$ is an upper bound on the length of an interval spanning two overlapping transaction instances.

If we cannot predict the arrival of transaction requests, the only safe way to maintain the above condition is to keep $accu_{i,j}$ to not exceed $\min\{\lfloor\frac{\delta_j}{2}\rfloor, \lfloor\frac{\omega_j}{2}\rfloor\}$ so that the stacks cannot grow unexpectedly in the future. As an example, consider a transaction instance τ_7 which preempts the transaction instance τ_5 unexpectedly at time t' as shown in Fig. 3 such that the sum of these two stack depths becomes larger than the W-recency bound at time t', although it is less than the W-recency bound at time t. Therefore, $\min\{\lfloor\frac{\delta_j}{2}\rfloor, \lfloor\frac{\omega_j}{2}\rfloor\}$ should be considered as a bound on the depth of the stack for the jth interactive set. We use $\alpha_j = \min\{\lfloor\frac{\delta_j}{2}\rfloor, \lfloor\frac{\omega_j}{2}\rfloor\}$ to denote the *recency bound* for the jth interactive set.

3.2.1 Calculation of Recency Bounds α_i

Given a similarity bound sb_x for a data object obj_x, let ω_x and δ_x be its W-recency bound and R-recency bound, respectively. Suppose p_x is the minimum period among the update transactions of obj_x. The recency bound $\alpha_x = \min\{\lfloor\frac{\delta_x}{2}\rfloor, \lfloor\frac{\omega_x}{2}\rfloor\}$ can be easily maximized by having $\omega_x = sb_x$ and $\delta_x = sb_x - 2p_x$. In this case, $\alpha_x = \lfloor\frac{sb_x - 2p_x}{2}\rfloor$.

The recency bound α_j of the jth interactive set is the minimum recency bound of data objects accessed in the jth interactive set. Notice that, if $sb_x \leq 2p_x$, then SSP guarantees that data object obj_x will be accessed exclusively because $\alpha_j \leq 0$.

3.3 The Scheduling Policy and Examples

Initially, $rn_{i,j} = accu_{i,j} = 0$ for all i, j. The recency bounds α_i of the interactive sets are calculated as defined above. Priorities of transactions are assigned as required. If a dynamic priority assignment such as earliest-deadline-first is used, then the priorities of transactions will change at run time according to the priority assignment policy.

3.3.1 Scheduling Rules

An unscheduled transaction instance $\tau_{x,y}$ with computation requirement c_x in the jth interactive set can be scheduled on the ith processor if the following conditions are all satisfied.

1. $\tau_{x,y}$ has the highest priority among all unscheduled transactions across all processors and has a higher priority than all scheduled transactions on its assigned processor.
2. $(accu_{i,k} + c_x) \leq \alpha_k$, for any k (including j) such that $rn_{i,k} \geq 1$.
3. $c_x \leq \alpha_j$ for any $k \neq i$ such that $rn_{k,j} \geq 1$.
4. $accu_{k,j} \leq \alpha_j$ for any $k \neq i$ such that $rn_{k,j} \geq 1$.

These rules can be interpreted as follows: The first rule ensures that the transaction instance under consideration has the highest priority among all pending transactions to preempt the scheduled transactions. The second rule enforces the strategy that conflicting events of overlapping transaction instances must be sufficiently close together in time, as discussed above. The third and fourth rules take care of the boundary condition of the rule 2 when there is a scheduled instance of a transaction whose computation time is greater than the recency bound. The third rule prevents the ith stack from growing over the appropriate recency bounds if there exists any active (scheduled but uncommitted) transaction instance in the jth interactive set running on other processors. The fourth rule ensures that any kth stack whose $accu_{k,j}$ is over the recency bound α_j should prevent the scheduling of any transaction from the jth interactive set on other processors, including the ith processor.

Notice that, if there is no transaction with computation time greater than the recency bound of its interactive set, then there is no need to check rules 3 and 4 because rule 2 ensures that no preemption stack can grow bigger than the recency bound of any interactive set.

3.3.2 The Maintenance of Scheduling Parameters

If an unscheduled transaction instance $\tau_{x,y}$ with computation requirement c_x in the jth interactive set is scheduled at time t on the ith processor, $rn_{i,j}$ and $accu_{i,j}$ are updated as follows:

- $accu_{i,k} = accu_{i,k} + c_x$, when $rn_{i,k} \geq 1$ for any $k \neq j$.
- If $rn_{i,j} = 0$, then $accu_{i,j} = c_x$; otherwise, $accu_{i,j} = accu_{i,j} + c_x$.
- $rn_{i,j} = rn_{i,j} + 1$.

When a scheduled transaction instance $\tau_{x,y}$ in the jth interactive set commits on the ith processor, $rn_{i,j}$ and $accu_{i,j}$ are updated as follows:

- If $rn_{i,j} = 1$, then $rn_{i,j} = accu_{i,j} = 0$; otherwise, $rn_{i,j} = rn_{i,j} - 1$.

Example 1. Scheduling four transactions on two processors:

We illustrate the working of SSP by an example. Suppose

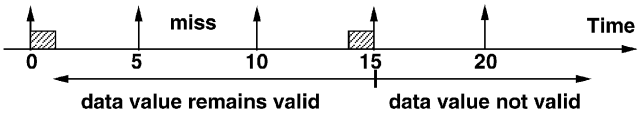


Fig. 4. Validity of data values.

there are four transactions τ_1, τ_2, τ_3, and τ_4 in a two-processor environment. Let τ_1, τ_2, τ_3, and τ_4 have computation requirements 1, 2, 1, and 2, respectively, and let them have the same period 5. Suppose τ_1 conflicts with τ_2, τ_2 conflicts with τ_3, and τ_3 conflicts with τ_4. According to our Similarity Stack Policy (SSP), these four transactions are in the same interactive set. For a typical similarity bound, we assume that the application permits data updates to be missed by one cycle, as is the case in many avionic software systems. As shown in Fig. 4, this can be taken to mean that data values shall remain valid for three periods. Thus, we can reasonably assume that a similarity bound of this interactive set is 15. Then, the recency bound of their interactive set is 2.5 from the formulas for W-recency and R-recency.

For comparison, let us first schedule these transactions according to the Priority Ceiling Protocol (PCP) [28] and the Stack Resource Policy (SRP) [2].[2] Without loss of generality, let τ_1, τ_2, τ_3, and τ_4 have fixed priorities 4, 3, 2, and 1, respectively, and have preemption levels 4, 3, 2, and 1, respectively. When τ_i is scheduled, the system priority ceiling and system preemption level are raised to i. As shown in Fig. 5, every execution of them prevents others from executing because of the high priority ceiling and preemption level. Therefore, it is unavoidable to have some transactions miss their timing constraints. However, SSP can successfully schedule them because more concurrency is allowed on account of similarity, and processor utilization is effectively raised. The execution of these transactions can complete at time 4, as shown in Fig. 6.

3.4 Properties of SSP

Theorem 2. *Every schedule allowed by the Similarity Stack Protocol (SSP) is view* Δ-serializable.

Proof. The proof follows directly from Theorem 1 if there exists a serial schedule π' which is a derived schedule of an arbitrary schedule π generated by SSP. According to the definition of *derived schedule*, π and π' must satisfy the following two requirements: 1) all write events read by every read event in π and π' must be strongly similar in π and 2) the last write events on every data object in π and π' must be strongly similar in π. In the following, we shall prove that there exists a sequence of event swaps from π to some serial schedule π' such that the requirements of a derived schedule are preserved at every step in the sequence.

Two transaction instances are executed concurrently in schedule π if they are on some preemption stacks simultaneously. By construction, SSP only allows transaction instances to be executed concurrently if they share no data objects or all of their conflicting events are strongly similar. Transaction instances which are not scheduled concurrently in π are serialized.

Since the conflicting write events of concurrently executing transaction instances in π are strongly similar, they can be swapped in any way without violating the second requirement of a derived schedule. Likewise, a conflicting read event and a conflicting write event of two concurrently executing transaction instances in π can be swapped in any way without violating the first requirement of a derived schedule because they are strongly similar. More importantly, instances of all write events on the same data object that occur in any interval shorter than the similarity bound can be swapped in a untimed schedule without violating consistency requirements. Thus, swapping such write events will not violate the first requirement of a derived schedule. Therefore, conflicting events of concurrently executing transaction instances can be swapped in any order. Since nonconflicting events can also be swapped in any order, events of concurrently executing transactions can be swapped in any order. In other words, concurrently executing transaction instances can be serialized in any order. As indicated above, transaction instances which are not scheduled concurrently in π are already serialized. Therefore, π can be serialized by swapping events of concurrently executing transaction instances in any order. □

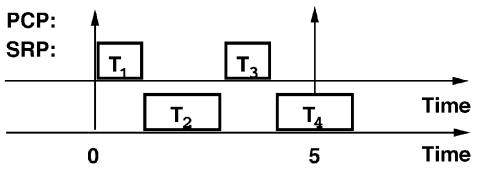


Fig. 5. Schedules of PCP and SRP.

Lemma 1. *No deadlock can occur under SSP.*

Proof. This follows directly from the fact that transaction instances will not be blocked by lower priority transaction instances once they are scheduled (stack discipline) and a transaction instance cannot be scheduled unless all higher-priority transactions are scheduled. □

Lemma 2. *No transaction instance can be blocked by lower priority transaction instances more than* $\max\{\alpha_i, c_j\}$ *for all* i, j, *where* α_i *is the recency bound of the* i*th interactive set and* c_j *is the computation requirement of transaction* τ_j.

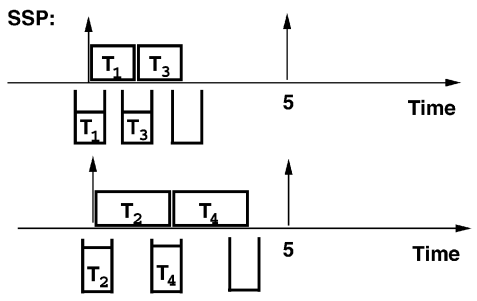


Fig. 6. A schedule of SSP.

2. Notice that PCP and SRP are originally designed for single processor systems. Hence, comparing them with SSP may not be germane.

Proof. Since no transaction instance is blocked by lower priority transaction instances once it is scheduled, its maximum priority inversion time is no larger than the maximum stack depth, i.e., $\max\{\alpha_i, c_j\}$ for all i, j. This is because the stack depth is the maximum computation time consumed by lower-priority transaction instances which are already scheduled and may prevent higher-priority transactions from being scheduled. □

Corollary 1. *Suppose there is only one interactive set and $\alpha \leq \min(c_j)$ for any computation requirement c_j of any transaction τ_j. Then, no transaction instance can be blocked by more than one lower-priority transaction instance.*

Proof. Follows directly from Lemma 2. □

To derive a schedulability bound for the SSP protocol, we introduce the concept of a *harmonic chain* and state a theorem by Kuo and Mok which is an extension of a well-known result of Liu and Layland [18] in scheduling.

Let S be the set of periods (positive numbers) of a set T of periodic processes. A subset R of S is said to be a *harmonic base* of the process set T if there is a partition, say Γ, of S into $|R|$ subsets such that: 1) each member of R is the smallest element in exactly one member of the partition Γ and 2) if x and y are two elements in the same member of the partition Γ, then either x divides y or y divides x. We call each subset in the partition Γ a *harmonic chain*.

Theorem 3 [13]. *Let T be a set of periodic independent processes and let K be the size of a harmonic base of T. If the utilization factor of T is smaller than $K(2^{\frac{1}{K}} - 1)$, then T is schedulable by a preemptive fixed priority scheduler.*

The following theorem establishes a schedulability bound for SSP.

Theorem 4. *Suppose a transaction set $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ is listed in increasing order of their periods p_is. A transaction τ_i is schedulable by SSP under the rate monotonic priority assignment if*

$$\left(\sum_{j=1}^{i} \frac{c_j}{p_j}\right) + \frac{b_i}{p_i} \leq K(2^{\frac{1}{K}} - 1),$$

where K is the size of the harmonic base of T and $b_i = \max\{\alpha_j, c_k\}$ for all j, k, where α_j is the recency bound of the jth interactive set and c_k is the computation requirement of transaction τ_k.

Proof. By accounting for the blocking time as an additional computation requirement of τ_i, the correctness of this proof follows directly from Theorem 3. □

4 Simulation Experiments

4.1 Measures and Methodologies

In the last section, we presented an analytic schedulability bound for the SSP protocol. This bound cannot be directly compared with the similar bounds that have been obtained for the well-studied Priority Ceiling Protocol (PCP) [28], [29] and Stack Resource Policy (SRP) [2] since the *blocking factors* in the formulas involve different types of parameters. Nevertheless, we would like to investigate quantitatively how the concept of similarity may be used to improve the schedulability of typical real-time transaction workloads. Our simulation experiments compared the performance of the SSP, PCP, and SRP protocols, in both single and multiple processor environments. While there is no significant difference in performance in the case of a single-processor environment, SSP performs substantially better than both PCP and SRP in a multiprocessor environment. Intuitively, this is to be expected since the use of locks in PCP and SRP tends to artificially serialize computations whereas SSP uses no locks at all!

Our experiments were conducted over hundreds of periodic transaction sets executed in a shared-memory multiprocessor environment. Exclusive locks were used in the implementations of the PCP and SRP protocols, since, as indicated in [29], the maximum priority inversion may quickly deteriorate if some data locks, such as read-locks, are not exclusive. This is because a high-priority update transaction may have to wait for more than one lower-priority query transaction. We also required every transaction to lock all of data objects it accesses before performing its computation and to release all of its locks after the transaction commits or aborts. This ensures that the serializability requirement is satisfied in the PCP protocol. It turned out that SRP behaves in the same way as PCP with rate monotonic priority assignment (RMS) since SRP requires a transaction to obtain all of its data objects (resources) before performing its computation. In the experiments, the similarity bounds of data objects (interactive sets) were chosen as multiples of the minimum period of their update transactions, instead of the maximum, so as not to unduly bias the results towards SSP since a large similarity bound usually implies better schedulability.

There were two parts in our experiments. The first part consisted of more realistic workloads and included transaction sets based on [3], [4], [22]; the second part included transaction sets generated by a normal distribution generator and by a random number generator and provided a more general and unbiased comparison of SSP and PCP/SRP. For each workload, the following protocols were exercised: SSP(RMS), SSP(EDF), PCP(RMS), SRP(RMS), and SRP(EDF), where X(EDF), X(RMS) means protocol X with, respectively, dynamic priority assignment by earliest-deadline-first and fixed priority assignment by the rate monotonic algorithm. For each protocol, every transaction set of various utilization factors was simulated from time 0 to time equal to the least common multiple of the periods of the transactions in the set. The simulation of each transaction set started with a utilization factor equal to 1 percent and increased by 1 percent at a time until the transaction set becomes unschedulable by the chosen protocol. The highest utilization factor schedulable by the chosen protocol for the transaction set is recorded and is referred to as the *schedulable utilization factor* of the transaction set by the protocol. The utilization factor of a transaction set was increased by adjusting the computation requirements of transactions according to the chosen utilization factor.

The improvement ratio of SSP(RMS/EDF) against PCP(RMS) and SRP(RMS/EDF) was computed as follows:

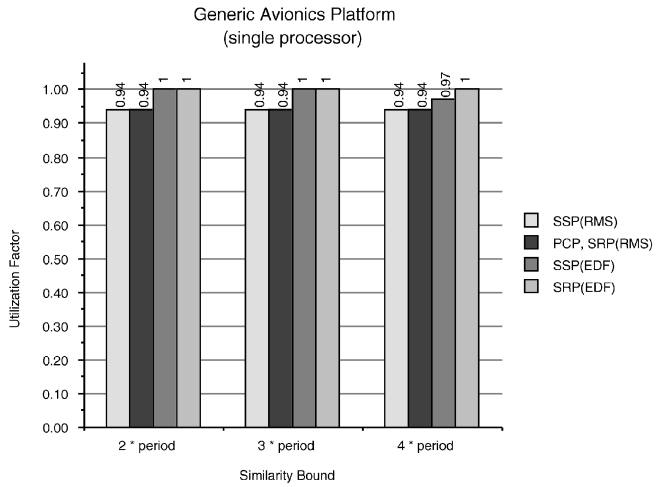


Fig. 7. The schedulable utilization factor of Generic Avionics Platform over various similarity bounds.

If the schedulable utilization factor of SSP(RMS/EDF) and PCP(RMS) or SRP(RMS/EDF) are x percent and y percent for a transaction set, respectively, then the improvement ratio of SSP(RMS/EDF) is $\frac{x-y}{y}$ for the transaction set. Note that minor factors such as the order of equal-priority transaction instances in a priority queue may slightly affect the schedulability of transaction sets. The results reported in the bar graphs were averages over hundreds of transaction sets.

4.2 Generic Avionics Platform

In the first part, the transaction sets were based on [3], [4], [22]. The first transaction set was based on Tables 1 and 2 in [22] which illustrates the timing constraints and data flow of processes in a *generic avionics platform* [22], respectively. We made some minor changes which included removing jitter requirements and transforming sporadic processes into periodic processes. The first test set had 18 periodic transactions and nine data objects, where some data objects had multiple update transactions. As shown in Fig. 7, SSP, PCP, and SRP performed equally well over various similarity bounds. As expected, SSP had no advantage in transaction scheduling in a single-processor environment. Note that with similarity bound ="2 * period," SSP is reduced into a nearly serial scheduler. Depending on the access pattern of data objects and the periods of update transactions in a transaction set, SSP may behave exactly like a serial scheduler which is what happens in this case. We think that the reasons why the schedulable utilization factor of SSP(EDF) with similarity bound equal to "4 *period" was 0.97 percent instead of 100 percent was caused by the order of equal-priority transaction instances in a priority queue or by the extra preemptions allowed by the large similarity bound.

4.3 HARM Low Cost Seeker

The second transaction set was an abstraction of a sanitized version of the HARM Low Cost Seeker system (HARM LCS) [3], [4]. It consisted of 32 transactions running on three processors. The transactions shared 48 data objects and were presumed to be periodic. Transactions in HARM LCS were tightly coupled and many data objects were updated by more than one transaction. As shown in Figs. 8 and 9, SSP(RMS) with similarity bounds equal to "2 * period" (which is a serial scheduler in this case) performed equally badly as PCP(RMS) and SRP(RMS). When EDF priority assignment was used, SSP(EDF) with similarity bounds equal to "2 * period" performed slightly worse than SRP(EDF) because SSP(EDF) in this case serialized the computation, but SRP(EDF) did make use of some degree of concurrency in the 3-processor environment. However, SSP(RMS/EDF) started improving the schedulability of the transaction sets substantially when the similarity bounds were over "2 *periods;" the improvement ratio of SSP(RMS/EDF) soared to 36.36 percent (42.5 percent) when the similarity bounds were "3 * period" or "4 * period." This is because the transactions in HARM LCS interacted with

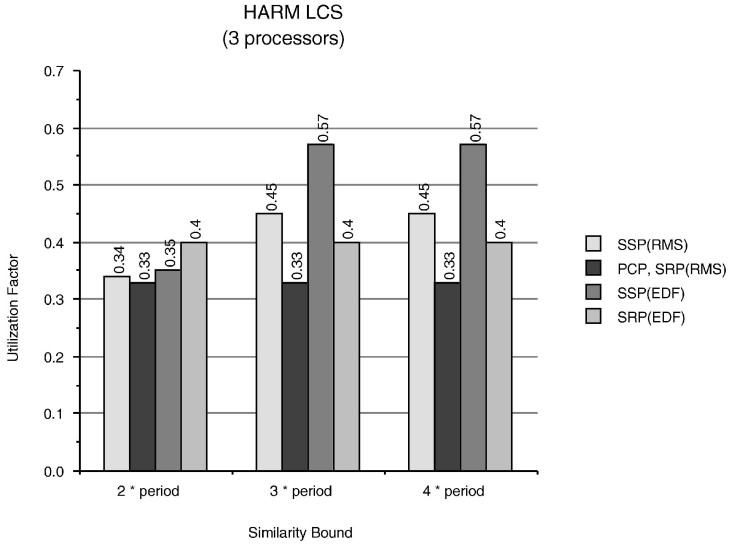


Fig. 8. The schedulable utilization factor of HARM LSC over various similarity bounds.

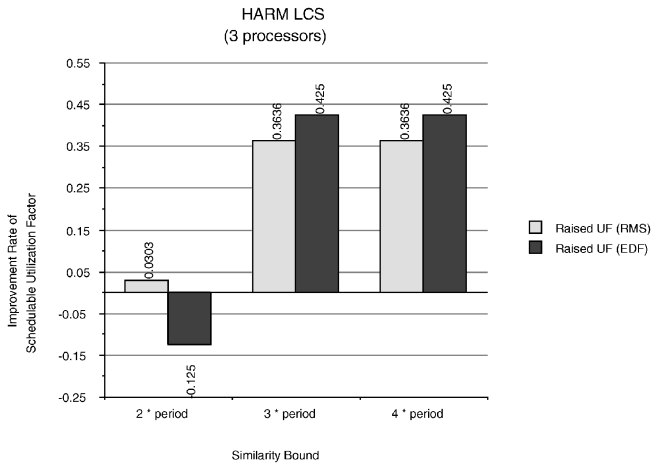


Fig. 9. The improvement ratio of HARM LSC under SSP over various similarity Bounds.

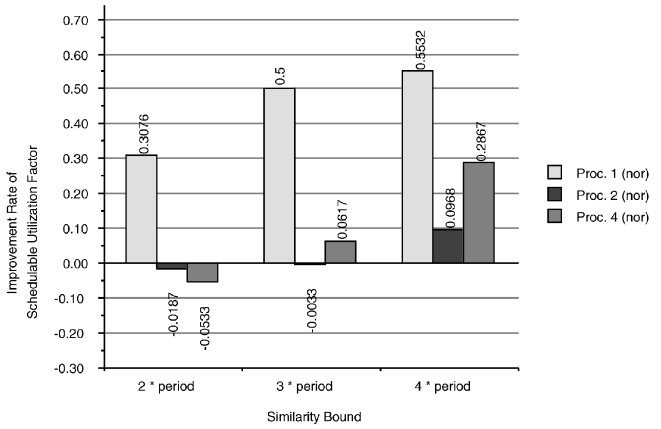


Fig. 10. The improvement ratio of transaction sets generated by a normal distribution function over various of similarity bounds.

one another heavily so that PCP and SRP both could take very little advantage of the multiprocessor. A transaction executing on one processor frequently forced the other two processors into idle status under PCP and SRP. On the contrary, SSP allowed conflicting transactions to run on different processors simultaneously and, therefore, improved the schedulability of the transaction set.

4.4 Randomly Generated Transaction Sets

The second part of our experiments simulated the execution of transaction sets executing on one, two, or four processors where the parameters were generated by a normal distribution function and a random number generator. The average number of transactions per processor was five. When a normal distribution function was used, the variance was 2. Transactions were first distributed to the processors by using the chosen distributions. Periods of transactions were then chosen in a range from 2 to $50,000$. In order to scale down the least common multiple of the periods, we exploited the idea of harmonic chains [13], and used a smaller average number of transactions. This way, we reduced the least common multiple of the periods (which can easily be very large) and made it possible to complete a simulation that would otherwise be impossible. The number of harmonic chains in a transaction set was selected by a normal distribution function or a random number generator, where the ratio of the number of harmonic chains and the number of transactions in a transaction set was selected from a range of ratios from 0.1 to 0.25 [3], [4], [12], [22]. When a transaction was assigned a harmonic chain, its period was selected from a "base" range from 1 to $1,000$ if the harmonic chain had no transaction; otherwise, it was assigned the product of the largest period in the harmonic chain so far and a number selected from a "factor" range of 2 and 50. If the product was over $50,000$, its period was selected from the periods of transactions in the same harmonic chain. Thus, transactions were more likely to have larger periods under a normal distribution function because the generator was more likely to hit periods in the middle of a harmonic chain in our experimental data. On the contrary, a random number generator scattered choices equally over periods in a harmonic chain. Note that the above period selection process was further randomized such that a transaction had no privilege to choose periods over others. Finally, the computation requirement and deadlines of transactions were selected by the chosen generator.

The average number of data objects per processor ranged from four to 12. The average number of data objects read and written by a transaction were four and two, respectively. The data objects accessed by a transaction were chosen by a normal distribution function or a random number generator. In general, transactions in a transactions set generated by a normal distribution function generator were more likely to share common data objects, i.e., they had more resource competition and a smaller set of accessed data sets than that generated by a random number generator.

Because the results are very similar, only the improvement ratios of SSP(RMS) against PCP(RMS) and SRP(RMS) are reported here.

Fig. 10 summarizes the experimental results of transaction sets generated by a normal distribution function. Transactions were executed from time 0 to time equal to their least common multiple of the periods. Fig. 10 indicates that SSP started improving the schedulability of transaction sets when the similarity bounds are larger than "2 *period." It is obvious that the improvement ratio of SSP soared rapidly as similarity bounds or the number of processors increased. However, there is an abnormal phenomenon in Fig. 10 where SSP substantially outperformed PCP and SRP in scheduling transaction sets on one processor, especially when similarity bounds were equal to "2 * period." Several factors contributed to this. As mentioned above, a normal distribution function tends to assign transactions larger periods and, therefore, larger similarity bounds. With the average number of transactions per processor set to a small number (five), this effect became dominant in single-processor cases. This is why the improvement ratio of SSP in scheduling transaction sets on one processor was so good. However, the high improvement ratio of SSP with similarity bounds equal to "2 * period" is a vexing question because SSP may be reduced to a nearly serial scheduler. By carefully studying the experimental data, we noticed that there was a transaction set without write events, and it alone contributed to the high improvement ratio of SSP

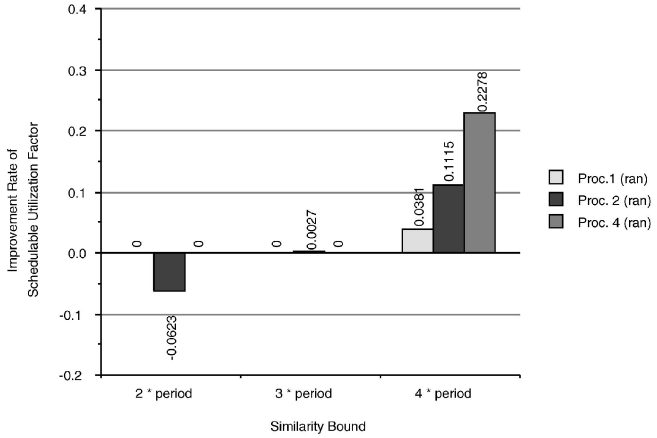


Fig. 11. The improvement ratio of transaction sets generated by a random number generator over various similarity bounds.

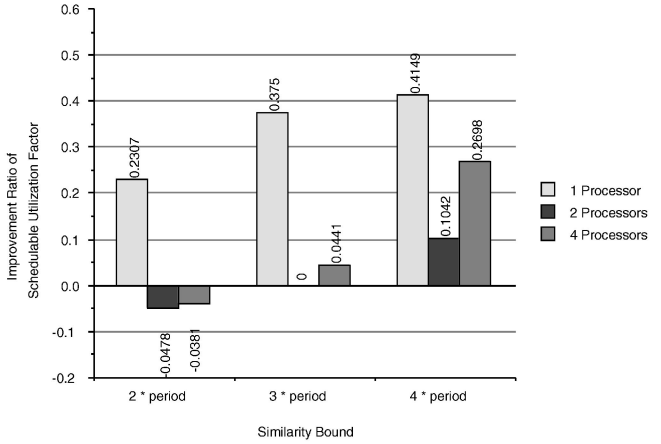


Fig. 12. The improvement ratio of randomly generated transaction sets over various similarity bounds.

with similarity bounds equal to "2 * period," where a read-only transaction set has infinitely large similarity bounds. This phenomenon suggests that the improvement ratio of SSP is more likely to increase substantially when data objects have fewer update transactions.

Fig. 11 summarizes the experimental results of the transaction sets generated by a random number generator. As expected, the improvement ratio of SSP also soared as the similarity bounds or the number of processors increased. The reason why the improvement ratio of SSP with similarity bounds equal to "2 * period" on two processors was -0.0623 percent is that some transactions in two transaction sets of the experiments had huge computation requirements. Once they were scheduled by SSP, their computation requirements simply caused the depths of the preemption stacks to exceed the recency bound of any interactive set, preventing the transactions with smaller periods from being scheduled. Thus, lower schedulable utilization factors by SSP resulted and, hence, lower averaged improvement ratios of SSP were observed in the 2-processor cases. Although we did not generate the same type of transaction sets in the 4-processor cases, we did expect the existence of such sets because a random number generator should yield a uniform distribution over a wide range of periods/computation requirements for transactions.

In general, we believe that the improvement ratios of SSP over PCP and SRP should be higher in scheduling transaction sets generated by a normal distribution function because a normal distribution function is more likely to let transactions share common data objects, i.e., transactions tend to have heavy resource competition and have larger similarity bounds. PCP and SRP are more prone to serialize transaction executions when the transactions are more likely to share common data objects. This is not the case in SSP because other factors such as the values of similarity bounds are equally important in deciding the behavior. Fig. 12 shows the average improvement ratios of Figs. 10 and 11, and that, in general, the improvement ratios of SSP soar as the similarity bounds or the number of processors increases. From the trend of the improvement ratios, it is likely that SSP will perform substantially better than PCP and SRP in multiprocessor environments.

5 Autonomous Multiprocessor Scheduling

Although the Similarity Stack Protocol (SSP) we have discussed so far shows promising results, the scheduling decision on a processor still depends on the execution state (the preemption stack) of other processors, just as it is with the Priority Ceiling Protocol or Stack Resource Policy. In this section, we present a modified SSP which allows independent scheduling of transactions across processors under some reasonable assumptions.

5.1 Assumption, Policies, and Properties

Given a transaction set $T = \{\tau_1, \cdots, \tau_n\}$, the independent scheduling of transactions across processors can be achieved if the computation requirement c_i of every transaction τ_i is no larger than the recency bound α_j of its interactive set. This assumption is likely to be true in many real-time applications, in view of the fact that the computation requirement of a real-time transaction is often relatively small compared to its period.

We call the modified SSP the *Multiprocessor Similarity Stack Protocol* (MSSP). MSSP is almost the same as SSP except for minor changes in its scheduling rules.

5.1.1 Scheduling Rules

An unscheduled transaction instance $\tau_{x,y}$ with computation requirement c_x in the *j*th interactive set can be scheduled on the *i*th processor if the following conditions are all satisfied:

1. $\tau_{x,y}$ has the highest priority among all unscheduled transaction instances on the *i*th processor and has a higher priority than all scheduled transaction instances on the *i*th processor.
2. $(accu_{i,k} + c_x) \leq \alpha_k$, for any k (including j) such that $rn_{i,k} \geq 1$.

The same properties of SSP that we listed in Section 3 also hold for MSSP, except that the least upper bound of the utilization factor is now dependent on the processor allocation policy. However, if transactions are preallocated

by an application and the assumption of MSSP is satisfied, then Corollary 1 should be removed and Theorem 4 should be changed as follows: Notice that the interactive sets of a given application and their recency bounds should be defined with respect to the whole transaction set in the application.

Theorem 5. *Given a transaction set $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$, suppose T_i is the set of transactions which are assigned the same processor as τ_i. Let T_i' be a subset of T_i whose transactions have higher or equal priority than τ_i and let IS_i be a collection of interactive sets to which transactions in T_i belongs. A transaction τ_i is schedulable by MSSP under the rate monotonic priority assignment if*

$$\left(\sum_{\tau_j \in T_i'} \frac{c_j}{p_j}\right) + \frac{b_i}{p_i} \leq K\left(2^{\frac{1}{K}} - 1\right),$$

where K is the size of the harmonic base of T_i and $b_i = \max_{j \in IS_i}\{\alpha_j\}$.

Finally, we mention that SSP is really a class of protocols since the recency bounds can be tuned.

6 Conclusion

In this paper, we have proposed the SSP (Similarity Stack Protocol), a class of data access protocols for real-time applications where serializability is too strict a criterion for correctness. Our protocols are based on the concept of similarity which is used routinely on an ad hoc basis by application engineers to provide more flexibility in concurrency control. We show that every schedule generated by our scheduling policies is view similar to some serial schedule [14], [15] and is thus logically correct. SSP schedules also enjoy limited priority inversion and at most two context switches per transaction.

With a proper implementation (not using the conceptual scheme described in this paper), the SSP scheduling policies provide lower overhead in scheduling than the Priority Ceiling Protocol or Stack Resource Policy. In our simulation experiments, our scheduling protocol improves the schedulability of real-time transaction workloads in many cases. Our results also suggest that SSP is also likely to be the preferred technique for real-time data access control in the multiprocessor environments, especially when transactions are tightly coupled with many shared data objects. It is also worth mentioning that, by exploiting the application semantics of real-time sensory data, a contribution of this paper is the formal justification for a class of timing constraints that are not characterized by simple deadlines in the usual real-time tasking models which have been a cause for concern to some researchers.

Finally, we should mention that the basic techniques in SSP and PCP (SRP) are really orthogonal and can be made to complement each other. How SSP can be combined with PCP and SRP to yield protocols that can be truly adaptive to the resource and data-sharing requirements of a workload is a subject for future investigation.

Acknowledgments

T.-W. Kuo's work was supported in part by a research grant from the ROC National Science Council under Grant NSC86-2213-E-194-017. A.K. Mok's work was supported in part by a research grant from the Office of Naval Research under ONR contract number N00014-89-J-1472. This paper is an extended version of a paper that appeared at the 14th IEEE Real-Time Systems Symposium.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. 14th Very Large Databases Conf.*, pp. 1-12, 1988.

[2] T.P. Baker, "A Stack-Based Resource Allocation Policy for Real-Time Processes," *Proc. IEEE 11th Real-Time Systems Symp.*, pp. 4-7, Dec. 1990.

[3] P.C. Clements, C.L. Heitmeyer, and A.K. Mok, "Applying Formal Methods to an Embedded Real-Time Avionics System," *Proc. First IEEE Workshop Real-Time Applications,* May 1993.

[4] P.C. Clements, B. Labaw, and A. Bull, "Modeling a Parallel Real-Time Avionics System with Modechart," technical report, US Naval Research Laboratory, Apr. 1993.

[5] R.P. Cook, S. H. Son, H.Y. Oh, and J. Lee, "New Paradigms for Real-Time Database Systems," *Proc. Real-Time Workshop,* 1991.

[6] W. Du and A.K. Elmagarmid, "Quasi Serializability: A Correctness Criterion for Global Concurrency Control in Interbase," *Proc. 15th Int'l Conf. Very Large Data Base,* pp. 347-355, 1989.

[7] S.B. Davidson and A. Watters, "Partial Computation in Real-Time Database Systems," *Proc. Fifth Workshop Real-Time Software and Operating Systems,* pp. 117-121, May 1988.

[8] H. Garcia-Molina and K. Salem, "Sagas," *Proc. 1987 ACM SIGMOD Conf. Management of Data,* pp. 249-259, 1987.

[9] H. Garcia-Molina and G. Wiederhold, "Read-Only Transactions in a Distributed Database," *ACM Trans. Database Systems,* vol. 7, no. 2, pp. 209-234, June 1982.

[10] 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.

[11] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley, "On Using Priority Inheritance in Real-Time Databases," *Proc. IEEE 12th Real-Time Systems Symp.,* pp. 210-221, Dec. 1991.

[12] K.L. Heninger, J.W. Kallander, and J.E. Shore, "Software Requirements for the A-7E Aircraft," Technical Report, NRL Memorandum Report 3876, Nov. 1978.

[13] T.-W. Kuo and A.K. Mok, "Load Adjustment in Adaptive Real-Time Systems," *Proc. IEEE 12th Real-Time Systems Symp.,* pp. 160-170, Dec. 1991.

[14] T.-W. Kuo and A.K. Mok, "Application Semantics and Concurrency Control of Real-Time Data-Intensive Applications," *Proc. IEEE 13th Real-Time Systems Symp.,* Dec. 1992.

[15] T.-W. Kuo and A.K. Mok, "Real-Time Data Semantics and Similarity-Based Concurrency Control," *IEEE Trans. Computers,* vol. 49, no. 11, pp. 1241-1254, Nov. 2000.

[16] 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.

[17] H.F. Korth, N. Soparkar, and A. Silberschatz, "Triggered Real Time Databases with Consistency Constraints," *Proc. 16th Very Large Databases Conf.,* Aug. 1990.

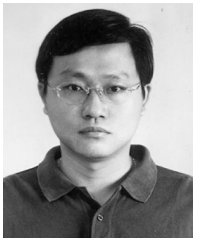
[18] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 46-61, Jan. 1973.

[19] K.-J. Lin, S. Natarajan, and J.W.-S. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," *Proc. IEEE Eighth Real-Time Systems Symp.,* pp. 210-217, Dec. 1987.

[20] K.-J. Lin and M.-J. Lin, "Enchancing Availability in Distributed Real-Time Databases," *ACM SIGMOD Record,* vol. 17, no. 1, pp. 34-43, Mar. 1988.

[21] 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.

[22] 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.
[23] A.M. Mok, "Fundamental Design Problems for the Hard Real-Time Environment," PhD dissertation, MIT, Cambridge, Mass., 1983.
[24] H. Pang, M. Livny, and M.J. Carey, "Transaction Scheduling in Multiclass Real-Time Database Systems," *Proc. IEEE 13th Real-Time Systems Symp.,* pp. 23-34, Dec. 1992.
[25] C. Papadimitriou, *The Theory of Database Concurrency Control.* Computer Science Press, 1986.
[26] C. Pu and A. Leff, "Epsilon-Serializability," Technical Report CUCS-054-90, Dept. of Computer Science, Columbia Univ., Jan. 1991.
[27] K. Ramamritham and C. Pu, "A Formal Characterization of Epsilon Serializability," Technical Report CUCS-044-91, Dept. of Computer Science, Columbia Univ., 1991.
[28] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Technical Report 9, CMU-CS-87-181, Dept. of Computer Science, Carnegie Mellon Univ., Nov. 1987, also in *IEEE Trans. Computers,* vol. 39, no. 9, Sept. 1990.
[29] 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.
[30] X. Song and J.W.-S. Liu, "Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency," technical report, Univ. of Illinois at Urbana-Champaign, 1990.



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 a professor in the Department of Computer Science and Information Engineering at the National Taiwan University, Taiwan, Republic of China. He was an associate professor in the Department of Computer Science and Information Engineering at the National Chung Cheng University, Taiwan, Republic of China, from August 1994 to July 2000. His research interests include real-time databases, real-time process scheduling, real-time operating systems, and embedded systems. He was the program cochair of the IEEE Seventh Real-Time Technology and Applications Symposium, 2001, and has been an associate editor of the *Journal of Real-Time Systems* since 1998. He has consulted for government and industry on problems in various real-time systems design. Dr. Kuo is a member of the IEEE Computer Society and a senior member of the IEEE.

Aloysius K. Mok received the BS degree in electrical engineering and the MS and PhD degrees in computer science from the Massachusetts Institute of Technology. He is a Quincy Lee Centennial Professor of computer science at the University of Texas at Austin. Since 1983, he has been on the faculty of the Department of Computer Sciences at the University of Texas at Austin. He has done extensive research on computer software systems and is internationally known for his work in real-time systems. He was the chairman of the technical committee on real-time systems at the Institute of Electrical and Electronics Engineers and has served on numerous national and international research and advisory panels. His current interests include real-time and embedded systems, robust and secure network-centric computing, and real-time knowledge-based systems. He is a member of the IEEE.