# Concurrency Control for Transactions with Priorities*

Keith Marzullo

TR 89-996
May 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N00140-87-C-8904.

# Concurrency Control for Transactions with Priorities*

Keith Marzullo†
Cornell University
Department of Computer Science

April 30, 1989

**Abstract**

*Priority inversion* occurs when a process is delayed by the actions of another process with less priority. With atomic transactions, the concurrency control mechanism can cause delays, and without taking priorities into account can be a source of priority inversion. In this paper, three traditional concurrency control algorithms are extended so that they are free from unbounded priority inversion.

**Keywords:** Priority inversion, concurrency control, real–time databases.

In a real–time system, the actions of some process may be more urgent than those of another. For example, the first process may need to synchronize with a physical process and sp must must a deadline. If both processes have access to common resources that cannot be shared, the less urgent process may delay the more urgent one by holding onto the resource. This situation

*Submitted to the *10th Real–Time Systems Symposium*, Los Angeles, December 1989.

†This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N00140-87-C-8904 The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

is commonly called a *priority inversion* [7,4]. There are several approaches to this problem, but the simplest is to simply force the less urgent process to relinquish the resource in favor of the more urgent process. *Priority schedulers* are an example of the implementation of this strategy[1].

In database management systems, the concurrency control mechanism is a scheduler through which a process may be delayed by the actions of another process. In this paper, some common concurrency control algorithms are extended so that priority inversions are detected and broken. Transactions will inherit their process's priority, and a transaction will be aborted or delayed if it could delay a more urgent transaction. A transaction is delayed while a less urgent transaction is aborted; we assume that aborts have a fixed overhead and can be taken into account when determining the running time of a transaction.

In this paper, we assume that the transactions submitted by a process are not known *a priori*. The schedulers presented here guarantee that the actions of a transaction cannot be delayed for more than a bounded time by the actions of transactions with less priority. A transaction, however, may be starved by the actions of transactions with more priority. In practice, these kinds of concurrency control algorithms are important for data base systems that support real–time transactions ([8], [1]). They are also important for real–time process control problems with concurrently accessed shared data.

We make the somewhat unusual assumption that priorities are assigned from a partial order rather than a total order. By doing so, we subsume the more typical priorities. We also allow more flexibility in specifying the inadmissible delays; with a total order, we may needlessly constrain the system. We also assume priorities are statically assigned.

This is not a practical paper, in that we have not implemented the algorithms presented here. Concurrency control algorithms are developed by making some decisions on what the equivalent serial order should be. Our goal in this paper is to re-examine these decisions when priorities are also considered. The amount of complexity some of these algorithms took on is surprising. There are some comments on the practical application of these

[1]In this paper, the more urgent process will be said to have *more priority* than the less urgent process.

algorithms in the conclusions of the paper.

In section 1, we describe the properties a concurrency control mechanism must have if it is to support transactions with priorities. In section 2 we develop a general concurrency control mechanism based on *serialization graph testing* algorithms that detects priority inversions. While easy to understand, such algorithms are complex to implement since a directed graph must be maintained and updated with each operation submitted to the scheduler.

There are two popular concurrency control mechanisms where the scheduler use a much simpler data structure at a cost of reduced concurrency. One (*two–phase locking*) delays operations to ensure serializability while the other (*timestamp order*) aborts operations to ensure serializability. In section 3 we show the typical extension of two–phase locking does prevent priority inversion when the priorities are drawn from a connected order. In section 4 we develop a timestamp order mechanism that detects priority inversion.

In this paper, we follow the notation and system model found in [2].

# 1 Concurrency Control

Suppose we have a set of processes submitting operations under transactions to a database scheduler. Each process can submit an unspecified number of transactions.

There exists a partial order $\succ$ of priorities over the transactions, where $p_1 \succ p_2$ means process 1 has priority over process 2. A transaction $T_i$ submitted by $p_i$ has the same priority as $p_i$, so we can also write expressions like $T_1 \succ T_2$. The database scheduler knows $\succ$ but has no other information about the transactions any process will submit. A transaction's priority is static; it cannot be changed by the scheduler or the process submitting the transaction.

Our goal is to devise a concurrency control algorithm that:

1. ensures the resulting execution is serializable, and

2. does not delay nor reject an operation of $T_i$ due to the action of $T_j$ when $T_i \succ T_j$.

In general, a scheduler can delay, reject or accept operations in order to guarantee the resulting execution is serializable. Typical schedulers abort a transaction by rejecting one of its operations. In our schedulers, a transaction will be aborted when an operation is submitted by another transaction with more priority.

The scheduler will also ensure that properties other than serializability are met by the resulting execution. For example, suppose a transaction $T_2$ reads the value of a variable $x$ written by transaction $T_1$. It is a bad idea to let $T_2$ commit before $T_1$ terminates. If $T_1$ decides to abort, $T_2$ will have committed using a value that was not produced by a committed transaction, possibly leaving the database in an inconsistent state. So, a scheduler should delay the commit from $T_2$ until $T_1$ decides to commit or abort. The property preserved by this delaying action is called *recoverability.*

A more dramatic delay is a *cascaded abort.* Using the above example, since $T_2$ has read $x$ written by $T_1$, if $T_1$ decides to abort, then $T_2$ must also abort. Again, the scheduler can prevent this condition by delaying some operations. For example, the read of $x$ by $T_2$ could have been delayed until it was after the termination of $T_1$.

In both cases, the delay of a transaction ($T_2$) was caused by a transaction ($T_2$) reading a value from an uncommitted transaction ($T_1$). This is a priority inversion when $T_2 \succ T_1$. The priority inversion can be represented graphically. A *reads from graph* (or *RFG*) is a directed graph with all currently active transactions as nodes. There are two kinds of edges in a *RFG.* A *priority edge* from $T_i$ to $T_j$ is drawn with a dashed arrow, and indicates $T_i \succ T_j$. A *reads from* edge from $T_i$ to $T_j$ is drawn with a solid arrow and indicates there is a value $x$ that was written by $T_i$ and later read by $T_j$. Figure 1 is a *RFG* showing $T_2 \succ T_3$, $T_2$ has read from $T_1$ and $T_1$ has read from $T_3$. A *cycle* in a *RFG* that that contains one priority edge represents a potential priority inversion. For example, in Figure 1 aborting transaction $T_3$ will force the abort of $T_2$ via $T_1$. We will call such cycles *priority inversion cycles.*

The following theorem argues this more formally.

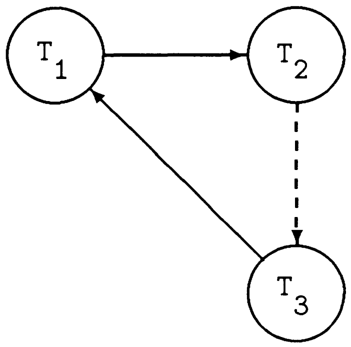


Figure 1: Reads–From Graph

**Theorem 1** *If the* RFG *of a set of transactions contains a priority inversion cycle, a priority inversion can occur.*

*Proof:* Suppose we have a *RFG* that contains such a a cycle. Let the two transactions with the priority edge between them be $T_i$ to $T_j$ such that $T_i \succ T_j$. By the definition of a *RFG*, $T_j$ is active. If $T_i$ wishes to commit, it must delay until $T - j$ commits; otherwise, the resulting execution would not be recoverable. Additionally, if $T_j$ aborts $T_i$ must (transitively) abort. Both cases represent a priority inversion. □

A *purely conservative scheduler* is a scheduler that never rejects an operation (thereby aborting the transaction submitting the rejected operation); it only delays operations until it is safe to execute them. Theorem 1 implies that there are no purely conservative schedulers that avoid priority inversion. Suppose such a scheduler existed, and it were submitted the operation $w_j x$ where $p_i \succ p_j$. By theorem 1, if $T_i$ were to submit the operation $r_i x$, it would introduce the possibility of a priority inversion. So, the scheduler must delay the write operation until it knows that $T_i$ will not submit a $r_i x$ before $T_j$ commits. Since the nature of the transactions submitted by $p_i$ are unknown to the scheduler, it must delay $w_j x$ forever.

Theorem 1 doesn't give a complete characterization of all priority inversions; it only deals with those due to cascaded aborts. For example, suppose we have the following history with $T_i \succ T_j$:

$$w_i x;\ w_j x;\ w_j y;\ c_j;\ w_i y$$

At this point, $T_i$ must abort due to the actions of $T_j$; otherwise, the execution will not be serializable. We will say $T_i$ is *ordered before* $T_j$ in a history $H$ if, in any serial history equivalent to $H$, $T_i$ occurs before $T_j$. Suppose $T_i$ is ordered before $T_j$ where $T_i \succ T_j$. If $T_j$ commits before $T_i$ terminates, $T_i$ could submit some operation that conflicts with $T_j$. This new operation violates serializability, and since $T_j$ has committed, $T_i$ must abort. To avoid this priority inversion, the schedulers developed here will generate histories with the following property.

**Definition 1** *A history $H$ is* priority committed *if for all pairs of transactions $T_i, T_j$ in $H$, if $T_i$ is ordered before $T_j$ and $T_i \succ T_j$, then $c_i < c_j$.*

A *purely aggressive scheduler* is a scheduler that never delays an operation; it rejects operations that violate its scheduling policy. A practical scheduler that generates priority committed histories will probably not be a purely aggressive scheduler. With a purely aggressive scheduler, if $T_i$ were ordered before $T_j$, $T_i \succ T_j$, $T_i$ were active, and $T_j$ submitted a commit, a purely aggressive scheduler would have to abort $T_j$. This abort could be unnecessary; if instead the scheduler delayed the commit until $T_i$ committed, the history would still be priority committed.

# 2 Priority Serialization Graph Testing

Serialization graph testing schedulers (or *SGT* schedulers) [6,2] guarantee serializable executions by maintaining a *serialization graph.* This graph contains nodes for all active and "relevant" committed transactions (described below). The scheduler ensures this graph contains no cycles, thus guaranteeing a serializable history.

$SGT$ schedulers are more of theoretical than practical interest. They are easy to understand and argue correct, but the overhead of maintaining a serialization graph may not justify any increase in concurrency over other schedulers. In this section, a $SGT$ scheduler will be extended to avoid priority inversions. This extension increases the complexity of the scheduler. In particular, much of the simplicity of $SGT$ schedulers comes from aborting a transaction only when it submits an operation. As noted in section 1, this policy cannot be used when avoiding priority inversion.

A $SGT$ scheduler operates as follows. When a transaction $T_i$ submits an operation $p_i x$, the scheduler tentatively adds *conflict* edges from all vertices $T_j$ to $T_i$ if there exists an operation $q_j x$ executed earlier that conflicts with $p_i x$. If $p_i x$ creates a cycle in the serialization graph, the scheduler aborts $T_i$, since the resulting execution would not be serializable. Once aborted, $T_i$ is removed from the graph along with all edges either into or out of $T_i$. If $p_i x$ does not create a cycle, the tentative edges can be made permanent and the operation executed.

To ensure the executed instructions are recoverable, the scheduler delays the commit from $T_i$ until all transactions from which $T_i$ read have also committed. Once $T_i$ has committed, $T_i$ can be removed from the serialization graph when it cannot be involved in any future cycles. Since all operations after $T_i$'s commit will be ordered after $T_i$, any new edges will be added leading out of $T_i$. This means $T_i$ can be removed when there are no edges in the graph leading into $T_i$. We will assume such transactions are automatically removed.

A priority serialization graph testing scheduler (or $PSGT$ scheduler) follows a similar strategy, with the caveats outlined in section 1. In particular, the rejection strategy of $SGT$ can cause a priority inversion. Instead of aborting the transaction that submitted the operation, we may have to abort a transaction with less priority. By generating priority commit histories, we will always be able to abort such transactions.

However, this strategy complicates the scheduler. If the submitted operation is a write, it could conflict with several unordered reads. Each new conflict can create a distinct cycle in the serialization graph. With $SGT$, all cycles are avoided by rejecting the new operation; with $PSGT$, we may have to abort a different transaction from each cycle.

Additionally, the *PSGT* scheduler will need to avoid priority inversions caused by cascaded aborts. The scheduler can do so by maintaining a *RFG* and checking for priority inversion cycles. Maintenance of a *RFG* is not as straightforward as a serialization graph. When a transaction is aborted, the *reads–from* relation changes which in turn may introduce new priority inversion cycles. For example, consider the following history where $T_0 \succ T_1, T_2, T_3$.

$$w_1x; w_2x; w_3x; r_0x$$

The only priority inversion cycle is ($T_0$, $T_3$). Once $T_3$ is aborted, the cycle ($T_0$, $T_2$) is created, and when $T_2$ is aborted the cycle ($T_0$, $T_1$) is created.

One way simplify detecting and removing priority inversion cycles is to augment the *RFG*. An *augmented RFG* will contain a vertex for each active transaction, and three kinds of edges:

1. Priority edges, as in a *RFG*.
2. Read–from edges, as in a *RFG*, except that the edge is labeled with the name of the variable that was read.
3. Write–after edges, also labeled with the name of a variable. When a transaction $T_i$ writes a variable $x$, a write–after edge labeled $x$ is drawn from the last transaction that wrote $x$ (if it is still active) to $T_i$.

When a read–from edge is added to the augmented *RFG*, the graph can be traversed to determine which transactions should be aborted. Let the function Abort($T$, $v$, $p$) be the set of transactions that must be aborted due to the read of variable $v$ written by $T$; $p$ is the priority of the transaction that submitted the original read operation. The functions read($T$, $x$) and write($T$, $x$) encode the reads–from and write–after edges; *i.e.* they are the transaction from which $T$ read $x$ and wrote $x$ after, respectively. Abort is recursively defined as follows.

$\text{Abort(T, v, p)} \stackrel{\text{def}}{=}$

$$\begin{array}{l} \textbf{if } \text{p} \succ \text{T} \rightarrow \{\text{T}\} \cup \text{Abort(write(T, v), v, p)} \\ \Box \; \text{p} \not\succ \text{T} \rightarrow \forall \text{ variables w read by T:} \\ \qquad\qquad \bigcup_w \text{Abort(read(T, w), w, p)} \\ \textbf{fi} \end{array}$$

Figure 2 shows an example, where write–from edges are drawn as doubled arrows. When $T_1$ submits $r_1x$, the function Abort$(T_2, x, T_1)$ is evaluated, yielding $\{T_3, T_4\}$. $T_2$ will also be aborted as a cascaded abort.

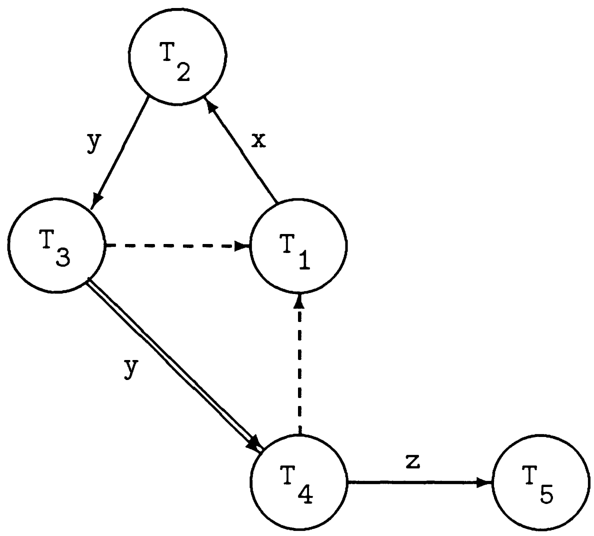


Figure 2: Abort$(T_2, x, T_1) = \{T_3, T_4\}$

A *PSGT* scheduler executes as follows. Let $T_i$ be a transaction that has submitted an operation $p_ix$ to the scheduler.

- If $p_i$ is a read or write operation:

  1. Add the operation to the serialization graph as described above. Let $\mathcal{C}$ be the set of cycles created by adding the new edges. If $|\mathcal{C}| = 0$, skip to step 3.

2. If $T_i$ can be aborted without introducing a priority inversion; *i.e.*

   $$\exists c \in \mathcal{C} : \forall T_j \in c : T_i \not\succ T_j$$

   then reject the submitted operation, abort transaction $T_i$ and await the next submitted operation. Otherwise, choose a set of transactions from the cycles in $\mathcal{C}$ that, when aborted, will remove all cycles (the selection process will be described shortly); abort these transactions, and proceed with step 3.

3. Add the appropriate edge to the augmented *RFG*. If the operation is a *read*, determine the set of transactions to abort, and abort them. The transaction that must be aborted are those in Abort(read($T_i$, $x$, $T_i$).

- If $p_i$ is a commit operation, the scheduler must ensure the history is priority committed. The commit operation is delayed until all transactions ordered earlier than $T_i$ in the serialization graph are either committed or of less or incomparable priority.

*PGST* maintains serializability in the same way *SGT* does; by maintaining an acyclic serialization graph. *PGST* avoids priority inversion by the (as yet unspecified) method used to select transactions to abort, described next.

Not all of the cycles in $\mathcal{C}$ need to be distinct; there can be cycles $c_1, c_2$ such that $c_1 \cap c_2 \supset \{T_i\}$. Note that if $c_1 \subset c_2$, $c_2$ is broken when $c_1$ is broken, and $c_1$ must be broken. In order to reduce the number of aborted transactions, the scheduler should examine the cycles in order of ascending length. The scheduler accumulates a list of transactions $\mathcal{A}$ to abort; if, when examining a cycle $c$, it is found that $\mathcal{A} \cap c \neq \emptyset$, the scheduler need not select a transaction from $c$ to abort. Otherwise, the scheduler can choose any active transaction from $c$; by the property of priority committed histories, any one with less priority relative to $T_i$ is still active.

Some issues have been glossed over for brevity. For example, the augmented *RFG* must be updated when transactions from $\mathcal{C}$ are aborted, and a transaction must be able to find the value of a variable after a cascaded abort.

# 3 Preemptive Two–Phase Locking

If we assume $\succ$ is connected (*i.e.* all processes have comparable priorities), two–phase locking ([3], [2]) can be easily extended to detect and eliminate priority inversion. Basic strict two–phase locking uses the following rules:

1. A transaction $T_i$ acquires a *lock* on a data item before referencing the item. These locks are typically *read* or *write* locks (also called *share* and *exclusive* locks) depending on the submitted operation. $T_i$ delays until the required lock is available.
2. All locks held by $T_i$ are released after $T_i$ commits.

In order to avoid priority inversion, a preemptive version of two–phase locking (*P2PL*) cn be used. When $T_i$ tries to acquire a lock, it waits until either the lock is free or all processes holding the lock with conflicting access have less priority. In the latter case, the scheduler then aborts the transactions holding the lock and gives it to $T_i$. Since all committed transactions follow the original two–phase rules, *P2PL* generates serializable histories. Additionally, while *2PL* is susceptible to deadlock, *P2PL* limits deadlock to occur only among transaction with the same priority. If a set of deadlocked processes have different priorities, there must exist a priority inversion, and *P2PL* will detect it and remove it.

*P2PL* does not have cascaded aborts, so it cannot generate priority inversion cycles in the *RFG*. A transaction $T_i$ reads from another transaction $T_j$ only after $T_j$ commits, and only active transactions are in the *RFG*, so the *RFG* will contain no reads–from edges.

*P2PL* generates priority committed histories without additional delays at commit. If $T_i$ is ordered before $T_j$, either there exists two conflicting operations $p_i x < q_j x$ or there exists a transaction $T_k$ such that $T_i$ is ordered before $T_k$ and $T_k$ is ordered before $T_j$. For strict two–phase locking, $(p_i x < q_j x) \Rightarrow (c_i < c_j)$, and since the commits form a total order, if ($T_i$ ordered before $T_j$) $\Rightarrow (c_i < c_j)$. This simplicity comes at a cost, however. For example, consider the submitted history $w_2x; w_1x; c_2; c_1$ where $T_1 \succ T_2$. Under *PSGT*, the commit from $T_2$ is delayed until after the commit of $T_1$; under *P2PL*, $T_2$ is aborted by the write from $T_1$.

As it currently stands, *P2PL* does not detect priority inversions with non-connected orders. Let $T_1, T_2, T_3$ have priorities $T_2 \succ T_3$, and let $T_3$ acquire an exclusive lock on $x$ and $T_1$ acquire an exclusive lock on $y$. If $T_1$ attempts to acquire the lock on $x$ it will block since $T_1 \not\succ T_3$. If $T_2$ then attempts to acquire the lock on $y$ it too will block since $T_2 \not\succ T_1$. We now have $T_2$ transitively blocked on $T_3$, which is a priority inversion. Extending *P2PL* to work with partial priority orders complicates the algorithm; it must examine the owner of all locks held by processes transitively blocking the request.

# 4 Priority Timestamp Order

Timestamp order (*TO*) schedulers ([9], [2]) operate by assigning transactions a *timestamp* when they start. The timestamp, typically an integer, places the transaction in a total order with respect to all other transactions. The scheduler ensures operations occur in an order consistent with the total timestamp order. Since the transactions are totally ordered, the history is serializable. The scheduler typically assigns timestamps in the order the transactions start, but this is not necessary; the scheduler guarantees the operations respect any order assigned by the timestamp allocation rule.

Associated with each variable $x$ in the data base is a *read stamp* $x.r$ and a *write stamp* $x.w$. These stamps are the timestamps of the last transaction to read and write $x$ respectively. When $T_i$ with timestamp $s_i$ submits an operation to a *TO* scheduler:

1. If it is a read operation: if $s_i < x.w$ then the read is *too late* and $T_i$ is aborted; otherwise, $x.r$ is set to $s_i$ and the read is executed.
2. If it is a write operation: if $s_i < x.r$ then this write is *too late* and $T_i$ is aborted; otherwise, the write is executed if $s_i > x.w$, and $x.w$ is set to $s_i$.
3. If it is a commit operation, it is delayed until all transactions that $T_i$ has read from have committed. There are several ways to achieve this property ([2]).

A timestamp concurrency control algorithm that detects priority inversion (*PTO*) allocates timestamps such that priority inversion cycles in the *RFG* cannot occur. A timestamp $s_i$ for $T_i$ is uniquely allocated from a total order such that it meets the following two conditions:

1. For all committed transactions $T_k$, $s_i > s_k$.
2. For all active transactions $T_j$: if $T_j \succ T_i$ then $s_i > s_j$ and if $T_i \succ T_j$ then $s_j > s_i$.

The first condition is the same as for typical *TO* schedulers: to do otherwise implies the later transaction must appear to have run *before* a committed transaction. The second condition guarantees that the *RFG* will contain no priority inversion cycles: a *reads from* edge cannot go from a transaction with less priority to one with more priority. Since the timestamps have a total order, there can be no *reads from* path from a transaction with less priority to one with more priority.

It is not difficult to generate timestamps that obey the above two conditions. If a timestamp is represented as a number, the number space must be dense. Consider transactions $T_j$ with timestamp $s_j$ and $T_i \succ T_j$ with timestamp $s_i < s_j$. For any $n$, if $n$ new transactions start with priority between $T_i$ and $T_j$, $n$ timestamps with values $s_i < s < s_j$ must be assigned. In practice this shouldn't be a real problem, and in extreme cases the scheduler can abort $T_j$.

*PTO* must use a different comparison rule than *TO*. With *TO*, a transaction is aborted if it submits its operation too late: that is, it has too low a timestamp. Under *PTO* the transaction with more priority could be the one that is late, so the transaction that acted *too early* should be aborted. Like *PSGT*, there can be several such transactions that acted too early. For example, consider the history $w_2x; r_3x; w_1x$ where $T_1 \succ T_2 \succ T_3$. The first two operations happened too soon, and $T_2$ and $T_3$ are aborted. Instead of associating a single read and write timestamp with a variable, a list of read timestamps and write timestamps must be kept. For recoverability, each list must contain at least one timestamp from a committed transaction. This lists can grow arbitrarily long, but in practice this shouldn't be a real problem. A timestamp can be removed from a list if the list contains a larger timestamp of a committed transaction. In extreme cases, the scheduler can abort the active transaction with the largest timestamp; *e.g.* transaction $T_j$ in the example above.

Since it is necessary to store lists of timestamps, the value of a write can also be stored with its timestamp. By doing so, fewer aborts will occur since a write can never be done too early. A database that stores histories of variables is called a *multiverison database* ([2,5]).

When $T_i$ with timestamp $s_i$ submits an operations, *PTO* uses the following rules:

1. If it is a read operation: $s_i$ is entered into $x.r$. The largest entry $s$ in $x.w$ such that $s < s_i$ is found, and the value written at that time is returned.

2. If it is a write operation: $s_i$ is entered into $x.w$ along with the value being written. Let $s$ be the smallest timestamp in $x.w$ greater than $s_i$, or $\infty$ if $s_i$ is the largest timestamp. All transactions $T_k$ in $x.r$ that have timestamps in the range $s_i < s_k < s$ are aborted (there may be no such transactions), as they read $x$ too early.

3. If it is a commit operation, it is delayed until all transactions with timestamps less than $s_i$ have committed. Once the transaction successfully commits, for each variable $x$ in $T_i$'s read (*cf.* write) set, the timestamp lists $x.r$ (*cf.* $x.w$) can be truncated: all timestamps less than $s_i$ can be removed.

When a transaction $T_j$ is aborted, its timestamps are removed from all variable timestamp lists. Additionally, transactions that read from $T_j$ must also be aborted. For each variable $x$ in $T_j$'s write set, let $s$ be the smallest time stamp in $x.w$ that is larger than $s_j$, or $\infty$ if no such timestamp exists. All transactions $T_k$ in $x.r$ such that $s_j < s_k < s$ read from $T_j$, so they are aborted.

*PTO* ensures serializability by using timestamps from a total order, and ensures there are no priority inversions for recoverability by its timestamp generation rule. The main weakness with *PTO* is the delay in the commit rule. Suppose a transaction only wishes to update $x$ but is started at the same time a long-running transaction with more priority is active. Even if the two transactions never reference the same variables, the shorter transaction must wait for the longer running transaction to complete. With both *P2PL* and *PSGT*, the shorter transaction will be able to complete without delay. For $PTO$ to do similarly, it must either maintain the actual *reads from* relation as $PSGT$ does, or know more information about the transactions (such as a transaction's read set and write set).

# 5 Discussion

This paper examined three common concurrency control algorithms and showed how each could be extended to avoid priority inversion. The results are mixed:

- Without some knowledge of the transactions that will be submitted, there are no purely conservative concurrency control schedulers nor any practical purely aggressive concurrency control schedulers that avoid priority inversion.

- Traditional aggressive schedulers, like serialization graph testing and timestamp order schedulers abort a transaction by rejecting an operation when submitted. This method cannot be used when priority inversion must be avoided. Instead, a transaction that submitted its operation earlier must be aborted, so the more urgent transaction can continue. This policy increases the complexity of aggressive schedulers. In the case of serialization graph testing, it isn't clear that the increased concurrency would ever compensate for the increased complexity, given a reasonable workload.

- The traditional conservative scheduler, two phase locking, can be easily extended to avoid priority inversion when the priority relation is connected. The extension for nonconnected priorities is somewhat more complex.

- Timestamp order schedulers, when extended to avoid priority inversion, suggest using a multiversion concurrency control algorithm. The extended algorithm is not much more complex than a traditional multiversion timestamp order algorithm. However, transactions with less priority can be needlessly delayed unless read sets and write sets are declared when a transaction starts.

The algorithms presented here have not been implemented, and their relative performance has not been examined in any detail. Additionally, only the priorities of transactions has been to schedule or abort operations. Other information could be used, such as the remaining running time of a transaction ([1]). It isn't clear what kind of information would be useful for the more aggressive schedulers.

These algorithms were developed as part of the Cornell *RR Project*, where which we are developing both theory and tools for building real–time reliable systems. Part of this project is the development of a process control system, which will eventually contain a database–like component. Our next step with the algorithms in this paper will be to evaluate them in the context of the *RR* project.

**Acknowledgements** This work profited from many discussions the author had with Özalp Babaoğlu and Fred Schneider, as well as with Jacob Aizikowitz, Ken Birman, Robert Cooper and Pat Stephenson.

# References

[1] Robert Abbot and Hector Garcia-Molina. Scheduling real–time transactions. *SIGMOD Record*, 17(1):71–81, March 1988.

[2] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison–Wesley, 1987.

[3] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[4] Özalp Babao glu, Keith Marzullo, and Fred Schneider. Priority inversion. In preparation.

[5] David Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1), February 1983.

[6] G. Schlageter. Process synchronization in database systems. *ACM Transactions on Database Systems*, 3(3):248–271, September 1978.

[7] Lui Sha, Ragunathan Rajikumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real–time synchronization. Technical report, Carnegie Mellon University Departments of CS, ECE and Statistics, May 1988.

[8] John A. Stankovic and Wei Zhao. On real–time transactions. *SIGMOD Record*, 17(1):4–10, March 1988.

[9] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(12):180–209, June 1979.