Applying Update Streams in a Soft Real-Time Database System [*]

B. Adelberg H. Garcia-Molina B. Kao
Deptartment of Computer Science, Stanford University
{adelberg,hector,kao}@cs.stanford.edu

Abstract

Many papers have examined how to efficiently export a materialized view but to our knowledge none have studied how to efficiently import one. To import a view, i.e., to install a stream of updates, a real-time database system must process new updates in a timely fashion to keep the database "fresh," but at the same time must process transactions and ensure they meet their time constraints. In this paper, we discuss the various properties of updates and views (including staleness) that affect this tradeoff. We also examine, through simulation, four algorithms for scheduling transactions *and* installing updates in a soft real-time database.

Keywords: soft real-time, temporal databases, materialized views, updates.

1 Introduction

The problem we study in this paper arose during the ongoing implementation of the STRIP real-time database system.[1] This system [2] provides traditional database services (e.g., SQL, indexing, recovery) with real-time facilities (e.g., transaction deadlines, triggers). It is a *soft* real-time system where the exact resource requirements are not known in advance; the system minimizes the number of timing constraints violated (as opposed to a hard system that guarantees all are always met). An application we are targeting for is program trading, where financial instruments and currencies are examined to discover trading opportunities.

The heart of such a real-time database system (RTDB) is the scheduler, which has to allocate scarce resources (mainly CPU cycles) to meet the time constrained service requests. All RTDB scheduling algorithms we are aware of (see [7, 11] for RTDB overviews) assume that service requests exclusively come from transactions. Hence, the scheduling problem is simply to decide what transaction to run next.

However, we have discovered that in almost all RTDB applications there is another very significant component: managing the data input streams and applying the corresponding database updates. In our program trading application, consider that there are currently over three-hundred thousand financial instruments world-wide that could be tracked. The update stream can be up to 500 updates/second during peek time [5]. In other applications, input streams report data from sensors (e.g., in an industrial control system) or service requests (e.g., call requests in a telecommunications system), or reflect the state of other databases.

The key issue is that these database updates should not be handled either as separate transactions with individual deadlines (overhead would be too great), or as a single transaction (it is a continuous process without a single deadline). Instead we will argue that they must be handled by a single process (as is done in CICS), and that the scheduler must *very carefully* balance the requirements of transactions and their deadlines, against the need to keep the part of the database that reflects the outside world up-to-date. If updates are executed with higher priority so as to maintain "external consistency," the system may be left with no time to meet transaction deadlines. On the other hand, if transactions are given preference, they may read stale data. Neither of these outcomes is desirable. In our example, missing deadlines represents missed opportunities, and operating on stale data means we may be making the wrong decisions.

Notice that our input stream management problem can be viewed as an instance of the *materialized view* problem. In other words, the portion of the real-time database that is being externally updated can be thought of as a materialized views of some external databases (in our example, the databases at the New York Stock Exchange, the Tokyo Exchange, and so on). Although the question of how to efficiently *export* a materialized view has been studied in great detail [6, 9, 12, 14, 3, 4, 13], the question of how to efficiently *import* a view has not been studied at all. It is usually assumed, often implicitly, that the problem of incorporating view updates is straightforward and that the resource demand it represents is trivial. We believe this is definitely not true, especially in real-time applications where the view importation must be done in a timely fashion.

Incidentally, one way to "solve" the problem we are

[*]This work was supported by the Telecommunications Center at Stanford University and by Hewlett Packard Company.

[1]STRIP stands for STanford Real-time Information Processor.

addressing is to limit the size of the materialized view so that the number of updates per second is so low that the resource demands for importing them are indeed trivial. In our sample application, this would correspond to selecting in advance a small subset of financial instruments to track. While this may be acceptable in some cases, it also severely restricts the types of application processing that can be done (e.g., the trades one may consider). We believe it is much better to develop efficient mechanisms for installing large volumes of updates, without interfering with the deadlines of transactions.

2 Properties of Updates and Views

Since a major concern of this paper is how to maintain external data consistency, we start by investigating properties of updates and views, including the notion of view staleness. In general, updates can be characterized by the following two properties:

- *Periodic versus aperiodic updates.* With periodic updates, the current value of a data object is provided at periodic intervals regardless of whether it has changed or not. Aperiodic updates do not occur at times predictable by the database system, and usually only occur when the value of the data object changes. In this paper we focus on aperiodic updates.
- *Complete versus partial updates.* In a complete update, the value of every attribute of an object is provided with each update, even if the value has not changed since a previous update. Partial updates provide only values for attributes that have changed since the previous update. In this paper we mainly focus on complete updates.

The database views (i.e., the database portion updated from external sources) may also have a variety of properties, including:

- *Snapshot versus historical views.* Snapshot views only maintain the value of an object at one point in time (typically the current time). Updating an object causes the previous attribute values to be lost forever. Historical views provide support for maintaining not only the current attribute values of an object, but its past values as well. In this paper we only consider snapshot views.
- *View complexity.* In some cases, the updates are applied to the view by simply storing the new values. In other cases, the update values must be transformed or combined with other values before being stored. For example, company names may have to be changed to match local conventions, and running averages may have to be computed.
- *View staleness.* The application semantics typically determine when view data is considered up-to-date or stale. There are actually several options for defining staleness.

  One option is to define staleness based on the time when update values were generated. In this case, updates arrive at the RTDB with a timestamp giving the generation time at their external source. A database value is then considered stale if the difference between the current time and its generation timestamp is larger than some predefined maximum age Δ.[2] We call this definition *Maximum Age* (*MA*). Notice that with *MA*, even if a view object does not change value, it must still be periodically updated, or else it will become stale. Thus, *MA* makes more sense in applications where all view data is periodically updated and/or where data that has not been recently updated is "suspect." For example, *MA* may be used in a plant control system where sensors generate updates on a regular basis, or in a military scenario where an aircraft position report is not very useful unless it is quite recent.

  Another option is to be optimistic and assume that a data object is always fresh unless an update has been received by the system (put into the update queue) but not yet applied to the data. We will refer to this definition as *Unapplied Update* (*UU*). A problem with *UU* is that it ignores delays that may occur before an update is handed to the RTDB system. Hence, *UU* is more useful in applications with fast and reliable delivery of updates. For example, *UU* may be used in a telecommunications RTDB server because delays outside the system may be irrelevant and because we do not want the extra traffic associated with *MA*. (Example: if a call is on-going, we do not want to be periodically notified that it is still going on.)

  There are several variations on these two basic definitions. For example, we could combine *MA* and *UU*. Here, an object would be considered stale if it were stale under either definition. Due to space limitations, in this paper we only consider *MA* (generation time) and *UU* (separate from *MA*) as the possible staleness criteria. Readers are referred to [15, 8] for some other interesting discussion on data timeliness.

Once staleness is defined, we need to discuss what transactions do when they encounter stale data. One option is for them to abort. For example, in the program trading application it may be dangerous to commit a transaction that made its decisions based on out-of-date information. Another option is to complete such transactions, but to raise some warning indicator. For instance, in a plant control system, it may be better to operate with stale data than to do nothing at all, as long as a "red light" goes on in the control room. A third option is to complete transactions that read stale data, without any special action. In this last case, the notion of staleness is only used to compare scheduling algorithms, but is not implemented in the system.

Notice that if transactions need to know at run time if their data is stale, detection mechanisms must be implemented, and they will have a cost. In particular, for *MA* all view objects must have a timestamp associated with them. For *UU*, for every object read we must check for

[2]For simplicity we assume synchronized physical clocks.

relevant updates in the unapplied update queue (see Section 3). Also notice that for *UU*, aborting transactions that read stale data is probably not a good strategy. If the data is stale, we have already identified some updates that need to be applied, so it seems best to apply the updates and resume the transaction, instead of aborting it. Under *MA* however, stale data will only become fresh if an external update arrives, so aborting the transaction is a reasonable option. In Section 6 we return to this issue and study the impact of aborting transactions.

3 Conceptual Model

In order to present concrete scheduling algorithms, we first need to define a conceptual model that specifies in more detail the structure of the RTDB system and the database, and how updates and transactions are processed.

3.1 System Architecture

As discussed in Section 1, updates can be applied as individual transactions or by a single update process. It is tempting to treat each update as a separate transaction because this leads to a simpler scheduler that only deals with transactions. Unfortunately, this approach creates many difficulties. First, it vastly increases the number of transactions in the system which creates additional overhead. Updates rates can be orders of magnitude higher than transaction rates, so treating updates as separate transactions can overload the scheduler. Second, while real-time transactions have explicit deadlines and values[3], updates do not. One may assess a value to an update based on the data it is updating and the value of the transactions that use the data. However, this assessment is dynamic as transactions are created and committed, and as newer updates on the same data arrive, changing the value of the older updates. Assigning a fixed value to an update when it arrives fails to capture these dynamics.

Therefore, in the architecture for our conceptual model there are three types of processes:

Controller: This process controls all of the other processes. It performs scheduling of the update process and transaction processes. Any communication with processes outside the RTDB system are handled by this process as well.

Update Process: This process installs all updates. Only one update process runs in the system no matter how many updates are queued. The install process does not have a traditional value or deadline since it will never terminate.

Transaction Process: There are many transaction processes in the system, one per transaction.

We model the cost of switching between processes as x_{switch} CPU instructions. (For our experiments, we need to assign values to parameters such as x_{switch}; this is done in Section 5. A summary of all model parameters is given in tables in that section.)

[3]The value of a transaction represents the benefit that the system gains by completing the transaction. In the program trading example, the value of a buy-sell transaction could be the profit gained by the trade.

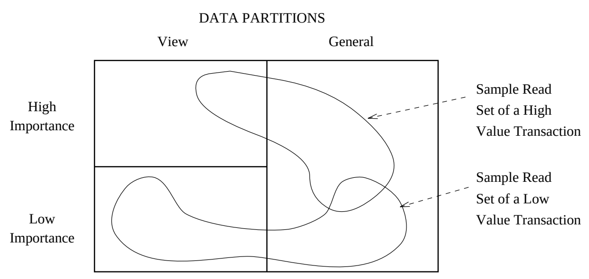
DATA PARTITIONS
View
General
High Importance
Low Importance
Sample Read Set of a High Value Transaction
Sample Read Set of a Low Value Transaction

Figure 1: Data partitions

3.2 Data Model

The data organization is shown in Figure 1. The database consists of a collection of *objects* partitioned into two sets: *view* and *general*. View objects represent materialized views from remote databases or real world variables (e.g. stock prices, sensor readings). In either case, view data is not updated locally but is refreshed with updates from an external source. We only consider snapshot views (no historical views). Transactions can read, but not write, the values of view data. We assume that each update has a timestamp recording the time at which it was generated by the external source, and that each database object has a timestamp giving the generation time of the current value. (In some cases timestamps are not required to check staleness, but we still assume them to simplify our discussion. For example if updates are applied in order of arrival and the staleness definition is *UU*, timestamps are not needed.)

In many cases, certain objects in the database are more "valuable" than others. For example, the Dollar-Yen exchange rate may be more important to investors than the stock price of the corner grocery store. Updates to the "hotter," more "valuable" objects would carry a higher potential value. We want to study whether performance can be improved by treating updates differently depending on how many valuable transactions access their data. Although data could have many different values, we believe we can study the essential issues by considering only two types of data. Thus, we subdivide the view objects into two sets: low importance and high importance. Low importance data is used by low value transactions and high importance data is used by high value transactions (described in Section 3.4). In our model, variables N_l and N_h determine the the number of low and high importance objects.

General data, in contrast to view data, is read and written only by transactions. It represents data derived from the view data as well as data created and maintained locally. Examples of general data from the program trading application would be composite indices or tables representing current stock holdings. Although some general data, i.e., derived data, may have timeliness constraints in practice, we have ignored them in this paper for the sake of simplicity; thus general data does

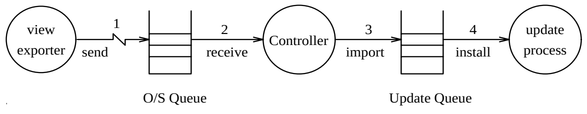

Figure 2: Update processing steps

not become stale.

3.3 Update Model

Each update changes only one object, either in the low or high importance view. The life of an update is illustrated in Figure 2. We assume that all updates arrive at the system over a network: they are not generated locally (step 1). This means that when an update arrives at the database, it will already have aged due to transmission time. Upon arrival, an update is queued by the OS until the controller can actively receive it (step 2) and enqueue it internally (step 3). The timing of these steps depends on the scheduling algorithm being used: the controller may want to immediately receive new updates when they arrive at the system or it may want the system to queue them until it is ready to receive them. If transactions have higher priority than updates do, the controller should not interrupt transactions to receive new updates: the context switch time alone under high update rates would seriously delay transactions. Instead, the controller will let the system queue the updates until there are no transactions to run. At that point, all of the updates will be received at once.

After an update is received (step 2), it must be enqueued within the RTDB system (step 3). The update queue is maintained in order of update generation, not arrival. This allows the system to apply updates in order even if the network has variable delay and to quickly discard updates that have become too old under MA. Updates are removed from the queue by the update process (step 4). The updater can choose to apply the update to a view or to discard it. The method for choosing the next update to remove depends on the scheduling algorithm used (Section 4).

The time required for each of the four steps is modeled individually. We do not model the time required for step 1 since it will have the same effect on all of the algorithms investigated. Step 2 is also considered instantaneous unless a transaction is running which needs to be preempted. In this case, the number of instructions required is $2x_{switch}$. Queueing time dominates step 3. We model the number of instructions needed to add/remove an update to/from the queue by $x_{queue} ln(n)$ where n is the number of updates in the update queue.

Finally, step 4 is composed of two steps. First, the database object to be updated must be found. This typically involves using an index, and we assume it takes x_{lookup} instructions. Before applying the update, some algorithms may check if the update is worthy. In particular, if updates are not applied in order the database may have a more current value for the object, in which case, the update can be skipped. If the update is applied, it takes x_{update} instructions. Notice that here we are only modeling CPU costs of database access. We could also incorporate I/O costs, but will not do so in this paper. This means we are focusing on a database that resides in memory (common for real-time applications), although the scheduling algorithms of Section 4 can be used with disk resident data.

3.4 Transaction Model

Transactions in our model have firm deadlines with associated values (see Section 5.2). After a transaction has missed its deadline it is worthless and so it will be aborted. Transactions in the system are divided into two classes: low value and high value. Transactions are scheduled by the database system prioritized on value density, which is the ratio of value to remaining processing time. This means execution time estimates are required by the system to calculate value density. Because transactions are valueless past their deadlines, when the system is overloaded it may be desirable to abort hopeless transactions earlier, before the system wastes time on them. This policy, called feasible deadline, also requires execution time estimates for the transactions. For this study, we assume perfect estimation which is impossible in practice but reasonable for studying first order effects. Also, run time estimates are not as difficult to compute for main memory databases as they are for disk based ones since there are no unpredictable I/O delays.

In our model, transactions execution follows the pattern: (1) Do work; (2) Read view data and perform staleness check; and (3) Do more work. Steps (1) and (3) are composed of accesses to general data and of CPU computations. The staleness check in step 2 will depend on the definition being used. A parameter p_{view} gives the fraction of the total work done in step 1 (with $1 - pview$ done in step 3). In practice, transactions do not read all the view data at once, but we believe this simple model is sufficient to study the effects of checking for staleness at different stages.

3.5 Metrics

A key component of our model is the set of metrics used for evaluation. As stated in Section 1, we need to go beyond the traditional "missed deadlines" metrics, to ones that include data staleness. The metrics we propose are:

- The average fraction of objects that are stale, $\overline{f_{old_l}}$ and $\overline{f_{old_h}}$. Define $f_{old_l}(t)$ to be the fraction of low importance data objects which are stale at time t and define $f_{old_h}(t)$ to be the equivalent fraction for high importance data objects at time t. We can now define $\overline{f_{old_l}}$ as

$$\overline{f_{old_l}} = \frac{\int_0^{t_{end}} f_{old_l}(t)dt}{t_{end}}$$

where t_{end} is the time the simulation ends. The metric $\overline{f_{old_h}}$ can be defined similarly.

- The fraction of all transactions that do not complete by their deadlines, p_{MD}.

- The fraction of all transactions that complete by their deadline without having read stale data, $p_{success}$.
- Out of the transactions that complete by their deadline, the fraction that also read fresh data, $p_{suc|nontardy}$. If $p_{suc|nontardy} = 0.33$, for instance, it means that out of the transactions that a traditional metric would consider successful (non-tardy), 66% are reading stale data.
- The average value per second (AV) returned by completed transactions. This gives us a measure of how much useful work was done.

4 Algorithms

The scheduling problem consists of two closely related components: (1) With what priority should the update process run, versus the transaction processes? (2) How should the install process decide what update to install next? In this section we present four scheduling algorithms addressing both components. There are many other possible variations, but we believe that these four represent the major choices available.

4.1 Do Updates First (*UF*)

Apply an update whenever it arrives at the system. This approach gives all updates higher priority than all transactions. If an update arrives while a transaction is being run, the database system must make a decision. The system can preempt the running transaction to receive and apply the update or it can wait until the transaction is finished and then apply all of the updates that arrived during its execution. In this study, transactions are preempted immediately when a new update arrives at the system. Since updates are run immediately upon arrival (except for a possible context switch delay) there is no need for the update queue discussed in the last section. If two updates arrive very close to each other, the second will be stored in the OS queue until the first update is installed.

4.2 Do Transactions First (*TF*)

This is the opposite of the previous approach. Updates are only applied when no transactions are waiting. Since updates should usually be quick, if a transaction arrives when an update is being processed the transaction will wait (no update preemption). Unlike *UF*, this algorithm requires an update queue to place updates in while transactions are running. It would also be possible to split the update queue into two queues, and to partition updates by their importance. When no transactions were waiting, updates could first be installed out of the high importance queue. This enhancement is a subject for future study.

When the update process finally runs, it can install updates in FIFO or LIFO order. With LIFO order, updates that arrived earlier may have to wait a significant amount of time, and the objects they cover could become stale. Furthermore, LIFO is not appropriate if the application requires that updates be applied in order. This seems to imply that FIFO could be better. However, with the *MA* staleness definition, the oldest updates in the queue might be close to expiring (reaching the maximum age) anyway, and FIFO would install them first, wasting resources. If the criteria is *UU*, there may be a more recent update to the same data which makes the older update worthless. Thus, applying updates in LIFO order will maximize the lifetime of the updated data, and could be a better strategy. The choice of queuing discipline, LIFO versus FIFO, is a parameter in our model.

Under heavy transaction loads, there may not be enough time left for the update process to keep up with update arrivals. In such a case, the length of the queues will grow unbounded. After the queue reaches a predefined maximum, the system could delete the oldest update when a new update arrived. Unfortunately, since updates are lost, transactions will have to be aborted or complete with stale data as a result. If the staleness criterion is *MA*, however, the maximum age can be safely utilized to bound the queues: updates can be removed from the queue when they exceed the maximum age. Since the update queue is kept in update generation order, this check can be performed in constant time. In our model, expired updates are identified and discarded at every scheduling point.

For *UU*, which has no maximum age constraint, the system must find another way to bound queue size. For systems with complete updates to snapshot views, such as the one we study in this paper, it is not necessary to store more than one update per view object since all updates but the newest are worthless. A hash table can be built on the update queue to help eliminate old updates and keep the queue size bounded. This approach is not evaluated in our experiments but does indicate an interesting direction for future work. Instead, we discard the oldest updates when the maximum queue size, UQ_{max}, has been exceeded. (This, of course, can cause transactions to accidentally read stale data.)

4.3 Split Updates (*SU*)

This algorithm is a compromise between the first two. Updates to the high importance data will be applied on arrival and updates to low priority data will be applied when no transactions are waiting. High importance updates will never be queued by the controller but low importance updates will when transactions are waiting. This means that the same questions regarding FIFO versus LIFO and bounding queue size discussed for *TF* apply to *SU* as well.

4.4 Apply Updates On Demand (*OD*)

In the three previous algorithms, if a transaction encounters stale data, it either reads it or aborts (depending on whether staleness causes abort). With the On Demand (*OD*) strategy, an attempt is first made to obtain fresh data from the update queue. The *OD* algorithm is an extension to *TF*: transactions will still always be given precedence over updates. However, when a transaction encounters a stale object, with *OD* the update queue will first be searched for an applicable update. If an update is found that will make the data fresh, it is applied. *OD* minimizes the number of transactions that read stale data and, if stale reads are not allowed, the number of transactions that are aborted. Notice that *OD* can only be used if it is possible to identify queued updates that may impact the desired object.

The queueing issues discussed for *TF* and *SU* apply to *OD* as well but with some variations. First, the choice between FIFO and LIFO servicing is less important since *OD* will search through the update queue anyways and find the most recent update. Unbounded queue size, however, is still a concern. The idea of building an index on the updates is even more useful for *OD* than for *TF* or *SU*. In addition to bounding the queue size, a hash table would reduce the time required to find applicable updates to stale data from a linear function to a constant one. Our evaluation did not consider extra indices, so *OD* must scan the queue for updates which takes time proportional to the number of updates in the queue. The number of instructions taken is $x_{scan}N_q$ where N_q is the number of updates in the queue.

5 Simulation Model

To evaluate the algorithms of Section 4, we developed an event-driven simulation. For the simulation, we need to assign values to the parameters of the conceptual model. At the same time, we need to simulate a transaction and update load. This section describes our choices.

To compare the relative performance of the algorithms, we selected a set of representative base parameter settings. We are not attempting to model any particular system; rather we selected a base setting that illustrates the main tradeoffs involved. For the base setting we then vary one or two parameters at a time, showing their impact on performance. We also performed sensitivity analysis on simulation parameters. Due to space limitation we do not present all of our results, only the ones we consider the most illustrative. (See [1] for full results.) The simulator is written in the simulation language **DeNet** [10]. Each simulation experiment (generating one data point) ran for 1000 simulated seconds. With an update arrival rate of 400/sec, 400,000 updates are processed for each data point generated.

5.1 Update Model

Update arrival is modeled as a Poisson process with arrival rate λ_u. The probability that an arriving update is to an object in the low importance group is defined to be p_{ul}. Conversely, the probability that an arriving update is to a high importance datum is p_{uh}. Clearly, $p_{ul} + p_{uh} = 1$. The actual datum changed by a particular update is picked randomly (and uniformly) from the set $\{1, 2, \ldots, N_l\}$ or $\{1, 2, \ldots, N_h\}$ depending on the class. When an update arrives at the database, it will already have aged due to the transmission time of the network. Ages are assumed to be exponentially distributed with mean $\overline{a_{update}}$. A summary of the parameters affecting updates is provided in Table 1.

Description	Param	Value
update arrival rate	λ_u	400
prob of low priority update	p_{ul}	0.5
mean age of new updates (s)	$\overline{a_{update}}$	0.1
# of low priority view objs	N_l	500
# of high priority view objs	N_h	500

Table 1: Scheduler baseline settings for data and updates

Description	Param	Value
transaction arrival rate	λ_t	10
prob of low value xaction	p_{tl}	0.5
min slack of xactions (s)	S_{min}	0.1
max slack of xactions (s)	S_{max}	1.0
mean value of low value xaction	μ_{v_l}	1.0
mean value of high value xaction	μ_{v_h}	2.0
S.D. of value of low value xaction	σ_{v_l}	0.5
S.D. of value of high value xaction	σ_{v_h}	0.5
mean # of view objs read by xactions	μ_r	2.0
S.D. of # of view objs read by xactions	σ_r	1.0
max age of data used by xactions (s)	Δ	7.0
mean computation time of xactions (s)	μ_x	0.12
S.D. of computation time of xactions	σ_x	0.01
frac of computation before view reads	p_{view}	0.0

Table 2: Scheduler baseline settings for transactions

5.2 Transaction Model

Transactions in the system are divided into two classes: low value and high value. Low (high) value transactions have normally distributed values with mean μ_{v_l} (μ_{v_h}) and standard deviation σ_{v_l} (σ_{v_h}). Transaction arrival is modeled as a Poisson process with arrival rate λ_t. The probability that an arriving transaction has a low value is defined to be p_{tl}. Conversely, the probability that an arriving transaction has a high value is p_{th}. Clearly, $p_{tl} + p_{th} = 1$.

The time to execute a transactions has two components: view access time and computation time. The number of view objects read by a transaction is normally distributed with mean μ_r and with standard deviation σ_r. The actual objects read by a particular transaction are picked randomly (and uniformly) from the set $\{1, 2, \ldots, N_l\}$ or $\{1, 2, \ldots, N_h\}$ depending on the class of the transaction. Each view read requires x_{lookup} instructions.

The cost to access general data is not modeled directly, but is included in the computation time of a transaction. We ignore concurrency control issues for general data since the database is in main memory and therefore has very little transaction concurrency. Since only one transaction will usually be running concurrently, the likelihood of conflicts, and hence the effect of concurrency control, is small. The computation time of a transactions is distributed normally, with mean μ_x and standard deviation σ_x. The slack of transactions is uniformly distributed in the range $[S_{min}..S_{max}]$. A summary of all of the transaction parameters is provided in Table 2.

Description	Param	Value
# instrs executed per second	*ips*	50×10^6
# instrs to find a data obj	x_{lookup}	4000
# instrs to update a data obj	x_{update}	20000
# instrs for context switch	x_{switch}	0
# instrs to add update to queue	x_{queue}	0
# instrs to read update in queue	x_{scan}	0
max size of OS queue (updts)	OS_{max}	4000
max size of update queue (updts)	UQ_{max}	5600
use feasible scheduling	feasible_dl	TRUE
allow transactions preemption	preemption	FALSE
update retrieval policy	queue_policy	FIFO

Table 3: Scheduler baseline settings for system

5.3 Performance Model

We assume a uniprocessor system with a CPU capable of *ips* instructions per second. The parameters that control the execution times of transactions and updates are included in Table 3 (together with other system parameters). Using these parameters, the number of instructions to perform an update is $x_{lookup} + x_{update}$. The time required to execute a transaction which reads exactly v view objects is the computation time of the transaction, which includes the time to read and write general data, plus $v \times x_{lookup}$. If the algorithm used is *OD*, a transaction could take even longer since if it encounters stale data it will incur scan costs. If it then finds an appropriate update, it will have to apply it which requires x_{update} instructions. The execution times in all cases can be obtained by dividing the number of instructions by the processor speed, *ips*.

6 Results

In this section we present the results of our simulations. Our goals are to compare the four algorithms with respect to the various performance metrics (Section 3.5), and to study the effect various system parameters have on the timeliness of both data and transactions. By evaluating the system under different settings, we are able to address the following questions:

- Which algorithm is best under which conditions?
- Overloading is usually undesirable in a real-time system because it causes a high percentage of missed deadlines. However, if our goal is a high *total* value returned, shall we submit more transactions to get more value? Or shall we keep the load low to avoid tardy transactions?
- Does distinguishing objects by their potential value improve performance? Will split updates (*SU*), which treats updates to objects with high/low values differently, be superior to other algorithms because of this?

Our discussion will first focus on using *Maximum Age* (*MA*) as the staleness criteria. We consider two cases (Sections 6.1 and 6.2), corresponding to whether transactions are allowed to read stale data or not. In both cases, we evaluate performance based on (1) how likely it is for a transaction to meet its deadline, (2) how much value the system can deliver, and (3) how fresh/stale the data is. In Section 6.3 the results of using *Unapplied Update* (*UU*) as the staleness definition will be discussed.

The experiments described in this section were all run with the parameter values described in Tables 1, 2 and 3 unless otherwise stated.

6.1 MA With No Transaction Abortion Due To Stale Data

In this experiment we consider data staleness as undesirable but non-fatal. Transactions that read stale data are no less valuable than those that read fresh data and hence are not aborted. However, it is still desirable to have data as up-to-date as possible.

To compare the scheduling algorithms under different loads we vary the transaction arrival rate (λ_t). Let us first relate λ_t quantitatively to the system load, and see how the system spends its time on updates and transactions.

Figure 3 shows the fraction of CPU time spent on transactions (ρ_t) and the fraction spent on updates (ρ_u), as λ_t varies. (The time spent on context switching between activities is charged to the activity that is being started or restarted.) The graphs show clearly the execution priority of updates and transactions under different algorithms. For example, algorithm *TF* shows a decrease in ρ_u as λ_t increases. This is because *TF* favors transactions and spends more time on them, to the detriment of updates. *UF*, on the other hand, services an update as soon as it can at the expense of transactions. Its ρ_u is therefore unaffected by λ_t. Also from Figure 3(b), we see that ρ_u starts out at about 0.19 for light loads. This means that installing the update stream (400 updates/second) fully takes up about one-fifth of the system time, a significant portion of the CPU cycles.

If we add ρ_t and ρ_u we obtain the total CPU utilization. This is not shown in a graph, but it grows steadily to 1 at about $\lambda_t = 10$ and stays at that value for larger λ_t. There is almost no difference in total utilization by algorithm, meaning that all algorithms do the same total amount of work, but expend it differently. We remark that although under normal conditions, a soft real-time system should be operating with low load and few deadlines missed, occasionally the system will be overloaded. It is precisely at those times when we need a good scheduler, and for this reason we study performance beyond the full load limit (i.e., beyond 10 transactions/second).

Next, we look at how well the different algorithms can keep up with the timing constraints of transactions. In our model, a transaction is associated with a deadline and a value. If the transaction is committed before its deadline, the system gains its value. Figure 4(a) shows the fraction of tardy transactions (p_{MD}) with respect to the transaction arrival rate. As expected, we see that more transactions miss their deadlines as λ_t increases. Among the four algorithms, *TF* and *OD* spend less time on updates (Figure 3(b)) and thus have more time on transactions, resulting in a smaller miss rate.

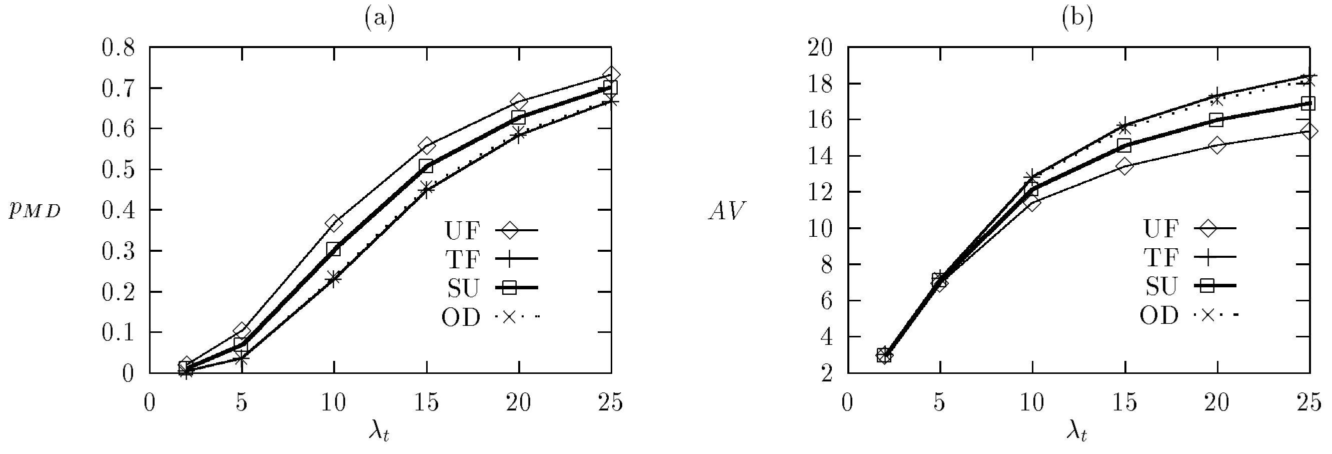

Figure 4: Effects of λ_t on fraction of missed deadlines and AV.

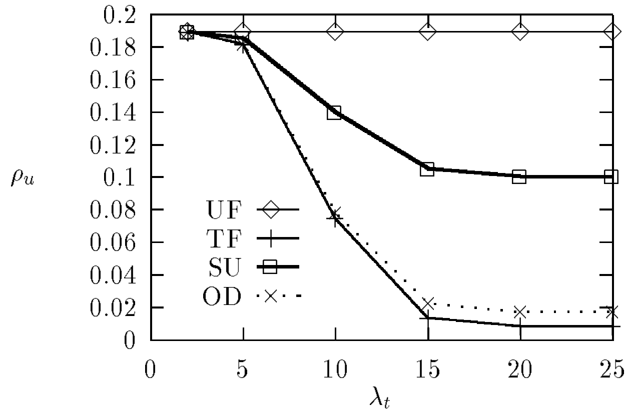

Figure 3: Effects of λ_t on update processing.

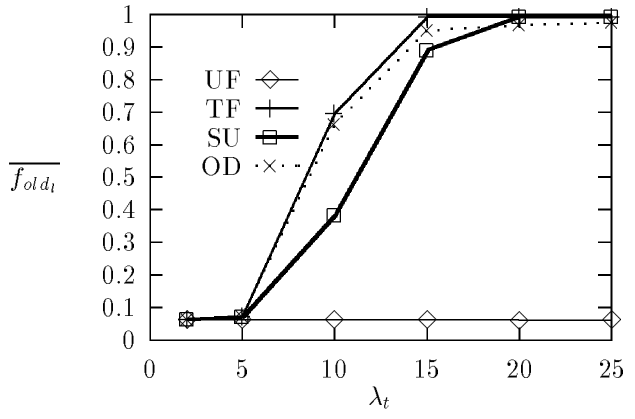

Figure 5: Effects of λ_t on f_{old_l}.

If the objective of the RTDB system is to minimize missed deadlines, one should not run the system with high transaction loads. However, in some cases the goal is to maximize value (e.g., in a program trading application), and this can lead to the opposite conclusion. Figure 4(b) shows the average value per second (AV) obtained by completing transactions (Section 3.5). We see that as the load increases, the value increases, even though more deadlines are missed. The reason is that as the load increases, the system gets more "opportunities" to chose from. It executes the higher values ones, ignoring the low value ones or the ones that are likely to miss their deadlines. Notice that *TF* and *OD* are also superior under the AV metric. Since *TF* and *OD* miss fewer deadlines than *UF* and *SU*, they enable more transactions to complete, returning a higher value.

We now look at data timeliness. Recall that under *MA* an object is stale if its age is larger than the maximum age allowed (Section 3.5). Figures 5(a) and (b) show the fraction of view objects that violated the age constraints in the low importance and high importance groups respectively.

From the figures we see that *UF*, which always gives updates first priority, is not affected by transaction load, and maintains the stale percentage of the database under 10%.[4] *TF* and *OD*, on the other hand, do poorly, showing high values of f_{old}. As we know, when the transaction arrival rate is high, *TF* and *OD* spend very little time on installing updates. When $\lambda_t > 15$, Figure 5 shows that most of the data objects are stale. Actually, *OD* is slightly better than *TF* because it does spend some time installing updates, in particular when transactions find stale data that can be refreshed from the update queue. As expected, the performance of Split Updates (*SU*) falls between that of *TF* and *UF*: *SU* keeps high importance objects fresh but not the low importance ones.

We have shown that *TF* and *OD* are good at meeting transaction deadlines and obtaining value, and that *SU* and *UF* are good at keeping the database fresh. If

[4]This *small* but non-zero percentage is due to the random nature of the update stream, which does not always refresh an object before its last update expires (Section 5.1). Of course, for other parameter settings this minimum staleness could be lower or higher.

it is clear to the system designer which factor (transaction timeliness or data timeliness) is more important, the choice is clear. However, if both factors are important, we have a dilemma: give preference to updates or to transactions? One way out of the dilemma is to look at $p_{success}$. As defined in Section 3.5, $p_{success}$ is the probability that a transaction meets its deadline and reads only fresh data. Figure 6(a) shows $p_{success}$ versus λ_t. As λ_t increases, $p_{success}$ decreases for all algorithms, meaning that the system is less successful in meeting both the data and transaction timeliness requirements. Interestingly, *OD* is the winner in our baseline setting with the highest success rate over the entire λ_t range, even when most of the database is stale. The reason *OD* does well is that it concentrates on refreshing on-demand the data that is actually needed by transactions, while not wasting time updating data that will not be read. *TF*, on the other hand, performs the worst because it lets its data get stale and then lets transactions read it. *UF* and *SU* show intermediate performance. Even though they miss more transaction deadlines, since their data is fresher, they manage to provide a reasonable value of $p_{success}$.

Another way to study the impact of stale data on the "success" of transactions is to look at $p_{suc|nontardy}$, the probability that a transaction only reads fresh data given that the transaction meets its deadline, Figure 6(b). A high $p_{suc|nontardy}$ value, such as the one observed for *OD* and *UF*, means that staleness is not an issue in determining the success of transactions. That is, if they meet their deadlines, it is very likely they read fresh data. On the other hand, the lower $p_{suc|nontardy}$ values of *SU* and *TF* mean that staleness is an issue: many transactions are meeting their deadlines but are not successful because they read stale data. Notice that for *SU* there is a counterintuitive dip in the $p_{suc|nontardy}$ curve. The reason is that under high λ_t, only high importance transactions can finish and *SU* behaves more like *UF* for high importance data, with its $p_{suc|nontardy}$ approaching that of *UF*. Under small λ_t, both high and low importance transactions can complete, and *SU* behaves like a hybrid between *TF* and *UF*. This accounts for the initial drop of $p_{suc|nontardy}$.

In conclusion, *OD*, which fetches updates from the update queue *on demand* seems to outperform the other simpler algorithms. It enables the system to return high value while at the same time avoids the use of stale data. The disadvantage of *OD* is that extra code is needed to maintain and to search the update queue efficiently. We will look at the impact of the queueing overhead in the following subsection.

6.1.1 Various Costs Of Updates

As discussed in Section 3.3, there are various costs associated with updates. First, there is the cost of installing the updates. Second, for *TF*, *OD*, and *SU*, there is the cost of maintaining an update queue for the unapplied updates. Third, for *OD*, the update queue is searched every time a transaction encounters a stale data object. To study the impact of these costs, we vary three cost factors: (1) x_{update}, the number of instructions required to update an object; (2) x_{queue}, the number of instructions required to add an update to the update queue; and (3) x_{scan}, the number of instructions required to examine *one* update in the update queue (see Section 5.3). Figures 7 and 8 show the impact of these parameters on the average value returned.

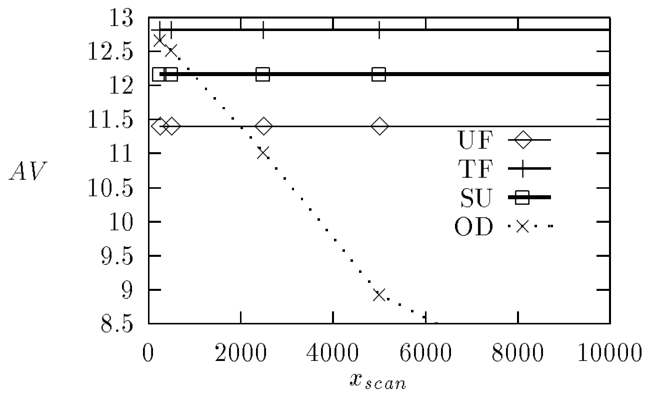

Figure 8: Effects of x_{scan} on AV.

From Figure 7(a) we see that *UF* and *SU* performance drops sharply as x_{update} increases, due to the higher number of updates these schemes process. Thus, *UF* and *SU* are not be advisable when updates are heavyweight, e.g., more than say 10,000 CPU instructions.

Algorithms *TF* and *OD* (and to a certain extent *SU*), however, rely on an update queue where they can store unapplied updates. Figure 7(b) shows the impact of the queue management cost. Keep in mind that x_{queue} is the proportionality constant for a queue insert, not the total cost. Thus, we believe it will be less than 1,000 instructions (probably less than 100). In this range, the overall queueing cost is not significant, and having the queue pays off in better *TF*, *OD* and *SU* performance.

Finally, *OD* incurs the additional cost of scanning the queue frequently, and the impact of this is studied in Figure 8. (Incidentally, under *UU*, the other algorithms also pay this cost.) Again, we believe that x_{scan} should be well less than 1,000 instructions, making the scan cost tolerable. The reason why *OD* does not degrade much, when say $x_{queue} = 100$, is that the update queue stays small. In a light load situation, the update process can keep up and the updates are installed. In a high load situation, unapplied updates on the queue time out and are removed. Also, notice that with an index on the update queue, the *amortized* cost of finding an update in the queue would be much less. Therefore, *OD* is still a promising strategy, even when queue management costs are taken into account.

6.2 MA With Transaction Abortion Due To Stale Data

In this section, we consider data timeliness as essential for commitment. That is, transactions are aborted *as*

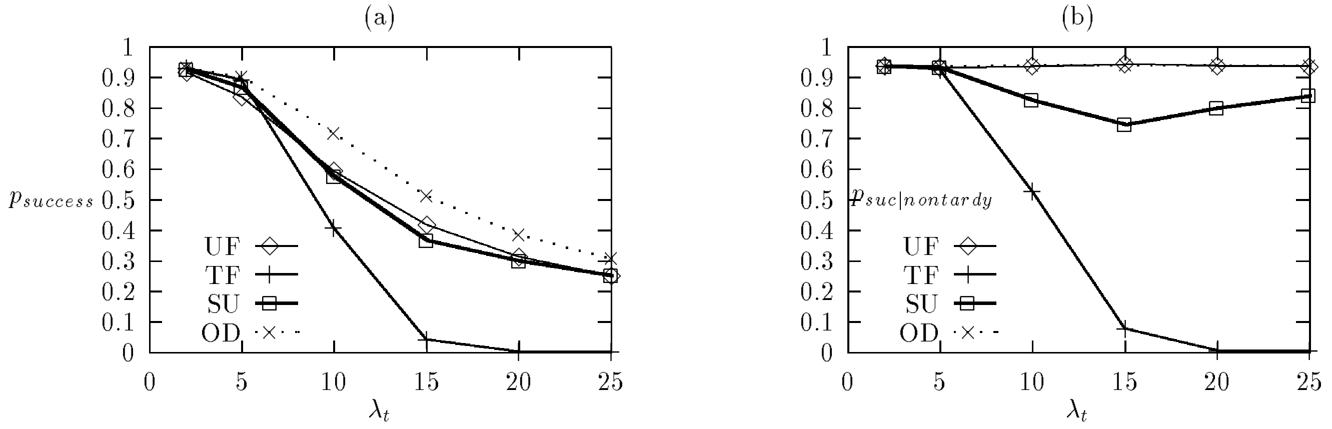

Figure 6: Effects of λ_t on $p_{success}$ and $p_{suc|nontardy}$.

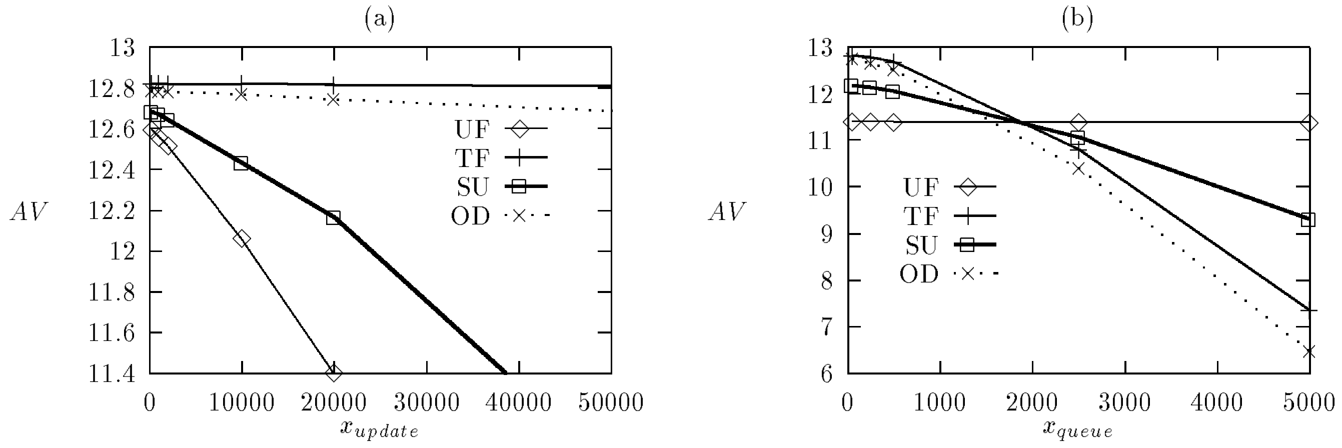

Figure 7: Effects of x_{update} and x_{queue} on AV.

soon as a stale data object is read.[5] Otherwise, this scenario is identical to that in Section 6.1. Due to space constraints, we will only present results in this section which are significantly different from those in Section 6.1.

Let us first note that aborting transactions because of stale data should affect those algorithms that fail to keep the database fresh (e.g., *TF*) more than those that can keep it fresh (e.g., *UF*). But notice that the impact can be positive in some ways. In particular, we will see that *TF* does abort more transactions than before, but because of this, it has more time to spend on keeping the database up to date, and the data becomes fresher!

To illustrate, Figure 9(a) shows the fraction of high importance objects that is stale. We see that f_{old_h} of *TF* drops to below 20% with stale-data abortions, as compared with 99% for the (earlier) no stale-data abortion case. Figure 9(b) directly compares the abortion case with the no abortion case by presenting the ratio: $f_{old_h}(with\ abortion)/f_{old_h}(without\ abortion)$. Points on the horizontal line at 1.0 indicate no change from the previous scenario. The lower a point on this curve, the larger the improvement in data freshness as compared to the no-abort scenario. The graph very clearly shows that there are significantly more fresh objects under *TF* when aborting transactions is in effect.

Figure 10 shows the average value returned for *MA* with transaction abortion on reading stale data. Next to it (Figure 10(b)) is a direct comparison with the no-abort-for-stale-reads scenario. Although the *AV* results are similar to the previous scenario, we see that *OD* is now the clear winner. The explanation is that *TF*, which was the closest contender to *OD*, is hurt the most by the higher overall abort rate.

Also, it is surprising to see that *SU*, which is a hybrid of *TF* and *UF* returns more value than either one of them. Recall that *SU* keeps high importance data fresh, while giving low importance updates low priority. Thus, it installs fewer updates than *UF*, leaving it time to complete transactions of higher value. These transactions are not aborted because the high importance data they access is kept fresh by *SU*.

[5]For *OD*, a transaction is aborted if it reads a stale object *and* could not find a recent update in the update queue.

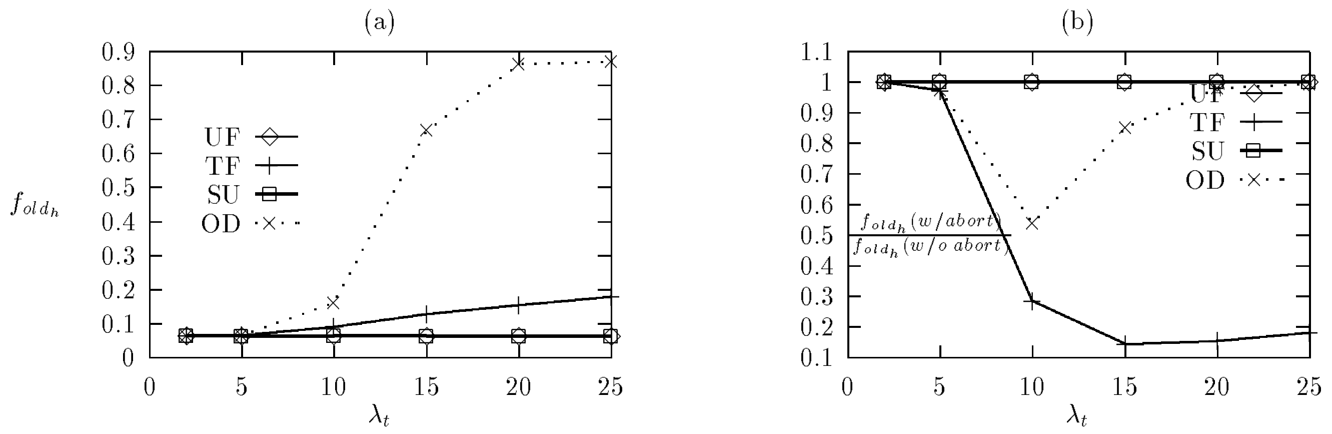

Figure 9: Effects of λ_t on f_{old} (*MA* with abortion).

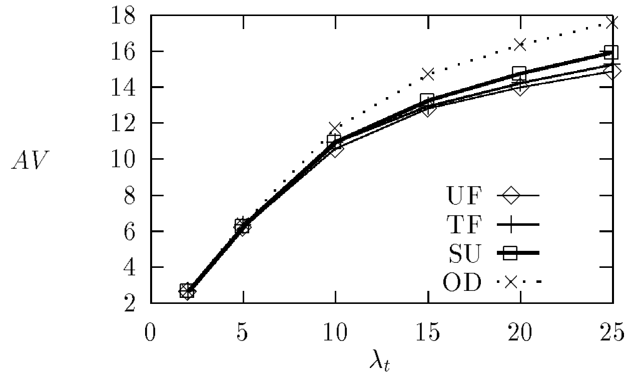

Figure 10: Effects of λ_t on AV (*MA* with abortion).

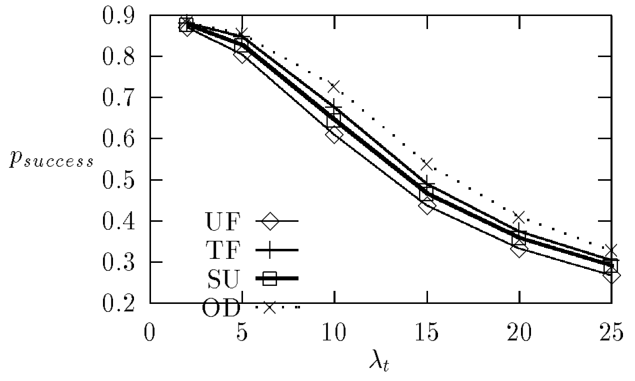

Figure 11: Effects of λ_t on $p_{success}$ (*MA* with abortion).

Finally, Figure 11 shows $p_{success}$, the fraction of transactions that read only fresh data and also meet their deadlines. Algorithm *OD* is still the winner, outperforming *UF* by a margin of 10 to 15 percentage points. *TF* which was the big loser in the no abortion case comes out in second place now, due to its low transaction miss rate and a much improved data freshness.

One important parameter that affects $p_{success}$ is p_{view}, the amount of work done before a transaction checks for staleness. (Note that this parameter is important only when transactions are aborted due to stale data.) Our results (not shown here) confirm that as p_{view} increases, more CPU resources are wasted on aborted transactions, and $p_{success}$ drops. However, all of the algorithms are affected similarly unless p_{view} is very large (close to 1.0).

6.3 UU With No Transaction Abortion Due To Stale Data

In this section, we briefly discuss the performance of the scheduling algorithms when *Unapplied Update* (*UU*) is used as the staleness criteria. Recall that, under *UU*, a data object is considered stale if there is an update on that object waiting in the update queue. This pending update carries a more recent value of the object and thus renders the *database* value obsolete, or stale. Unlike *MA*, the system is not given a time window (Δ) in which to install the updates. When an update arrives, the system has to install it immediately or else the database object involved will turn stale at once. The implication is that the use of *UU* strongly favors algorithm *UF*, which does not have an update queue to begin with and thus never lets any object turn stale. *TF*, on the other hand, buffers updates using the update queue while serving transactions first. Many objects are therefore stale with *TF*. Using *UU* instead of *MA* thus has the effect of further differentiating the two algorithms.

Notice that under *UU* algorithm *OD* must scan the update queue on *every* database read. This is because this is the only way to check if an object is stale or not. Contrast this to the *MA* case, where *OD* could determine if an object was stale by simply looking at its timestamp. This extra work makes transactions take longer, increasing the number of missed deadlines a bit.

In spite of these differences, the ranking of algorithms on the different metrics is not significantly changed between *MA* and *UU* (in both cases, with no aborts due to stale data). We conducted numerous experiments to verify this, but here we only show one graph: Figure 12

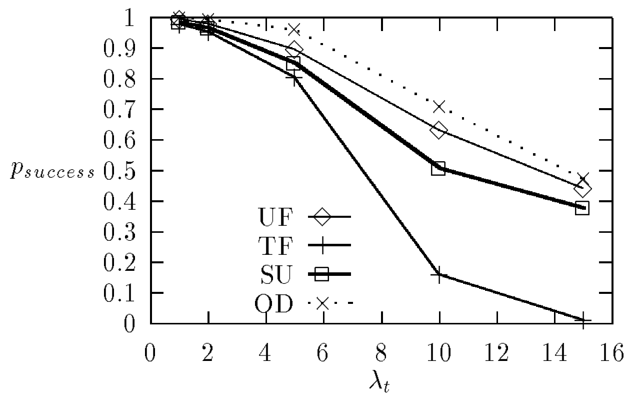

Figure 12: Effects of λ_t on $p_{success}$ (*UU*).

shows $p_{success}$, the probability that a transaction makes its deadline and read only fresh data. This figure best summarizes the performance issues by showing how each algorithm strikes a balance between data and transaction timeliness. Comparing to Figure 6, we see that the ranking, from best to worst, is still *OD*, *UF*, *SU*, *TF*.

Incidentally, notice that it does not make sense to compare the actual $p_{success}$ numbers of each scenario. The success rate under *MA* depends on the value of Δ, and can be made larger or smaller, depending on the "shelf life" that objects have. For *UU*, however, the success rates are only determined by the update stream and the ability of each algorithm to keep up.

7 Conclusions

As a rule of thumb, algorithm *On Demand* (*OD*) gives the best overall performance: fewer missed deadlines, higher value returned, and a higher $p_{success}$ rate. The only time *OD* fails is when the cost of the update queue operations (lookup, insertion, etc.) is abnormally high. However, *OD* may not be applicable if one cannot easily identify the queued updates that may impact the value of a database object. For example, say a database object X represents the average price of stocks in a particular portfolio. If a transaction wants to read X, *OD* would have to figure out what updates in the queue refer to stocks in the given portfolio, and then apply those.

In case *OD* is not suitable for an application, the designer has to decide whether data timeliness or transaction timeliness is more important: If meeting transaction deadlines and gaining value is critical, we should always schedule transactions first; If it is important that transactions do not read stale data, we should apply updates first. In case keeping data timeliness is important but not critical, *Split Update* (*SU*), which keeps high importance objects fresh while allowing transactions time to run was seen to be a good compromise. The key principle here is that even if we cannot keep every object fresh, we should at least keep up to date those that are used by the more valuable transactions.

In the course of our study, we also evaluated the algorithms under many other scenarios. For example, we studied the effects of changing the size of the maximum age of objects used in *MA*, the effects of applying the updates in the update queue in LIFO or FIFO order, and the effects of varying the update arrival rate. In any case, the results support our general conclusions about the algorithms' relative performance. Due to space limitations, these results are not reported in this paper. Readers are referred to [1] for a more complete discussion on these experiments.

References

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. Technical report, Stanford University, 1994.

[2] B. Adelberg, H. Garcia-Molina, and B. Kao. A real-time database system for telecommunication applications. In *Proceedings of the 2nd International Conference on Telecommunication Systems*, 1994.

[3] M. Adiba and B. Lindsay. Database snapshots. In *Proceedings of the 6th VLDB Conference*, pages 86–91, 1980.

[4] J. A. Blakely, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *ACM SIGMOD Proceedings*, pages 61–71, 1986.

[5] M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. In *Proceedings of the 20th VLDB Conference*, pages 714–721, 1994.

[6] E. Hanson. A performance analysis of view materialization strategies. In *ACM SIGMOD Proceedings*, pages 440–453, 1987.

[7] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In *Proceedings of NATO Advanced Study Institute on Real-Time Computing. St. Maarten, Netherlands Antilles, Springer-Verlag*, 1993.

[8] T. W. Kuo and A. K. Mok. SSP: A semantics-based protocol for real-time data access. In *IEEE Real-Time Systems Symposium*, pages 76–86, 1993.

[9] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *ACM SIGMOD Proceedings*, pages 53–60, 1986.

[10] M. Livny. **DeNet** user's guide. Technical report, University of Wisconsin-Madison, 1990.

[11] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1:199–226, 1993.

[12] N. Roussopoulos and H. Kang. Perliminary design of ADMS $\pm$: A workstation-mainframe integrated architecture for database management systems. In *Proceedings of the 12th VLDB Conference*, pages 355–364, 1986.

[13] A. Segev and H. Gunadhi. Temporal query optimization in scientific databases. *IEEE Data Engineering*, 13(3), sep 1990.

[14] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–84, 1989.

[15] X. Song and J. Liu. How well can data temporal consistency be maintained? In *IEEE Symposium on Computer-Aided Control System Design*, pages 275–284, 1992.