CUCS-336-87

Performance Evaluation of Global Reading of Entire Databases

Calton Pu, Christine H. Hong,[1] *and Jae M. Wha*

Department of Computer Science
Columbia University

Abstract

Using simulation and probabilistic analysis, we study the performance of an algorithm to read entire databases with locking concurrency control allowing multiple readers *or* an exclusive writer. The algorithm runs concurrently with the normal transaction processing (on-the-fly) and locks the entities in the database one by one (incremental). The analysis compares different strategies to resolve the conflicts between the global read algorithm and update. Since the algorithm is parallel in nature, its interference with normal transactions is minimized in parallel and distributed databases. A simulation study shows that one variant of the algorithm can read the entire database with very little overhead and interference with the updates.

1 Introduction

In many situations we would like to read an entire database (usually called a checkpoint). Data in a database must satisfy certain assertions called *consistency constraints*. In order to preserve data consistency under concurrent access, the usual locking concurrency control allows multiple readers *or* an exclusive writer. A common assumption in the literature is that a consistent and complete picture can be obtained only with a quiescent database. The reason is that 2-phase locking [2] -necessary for consistency- would require a naive reader of the entire database to lock all data at least for a moment, thus updates must stop.

We should note that checkpointing in databases that allow multiple readers *and* a writer presents no problem. In principle, any database that maintains two versions of its data can provide this level of concurrency [1]. However, for efficiency reasons, most practical databases write in-place. It is in these cases that our work should become useful.

We have previously described an algorithm [7,8] that can read the database entities one by one (it is *incremental*), avoiding deadlocks and allowing update activities to proceed concurrently (it works *on-the-fly*). The algorithm has two characteristics that facilitate its implementation. First, our algorithm consumes modest hardware resources; it does not maintain extra copies of the database and produces only sequential output. Second, no additional disk storage is required, so only modifications on the concurrency control are needed to adapt the algorithm to existing database systems.

In this paper, we analyze the different strategies to resolve conflicts between the global-read algorithm and update transactions. We show that one variant of the global-read (the Save-Some strategy) avoids aborting the updates, delaying them very little, and carries very small system overhead. A simulation study provides a quantitative confirmation of the qualitative analysis. The global-read is especially useful in parallel databases, since it increases system concurrency (and availability) at modest resource consumption, abundant in a parallel or distributed environment.

The paper is organized as follows. The global read algorithm [7] is summarized in section 2 to make the paper self-contained. In section 3, we expand on a previous paper [8] describing in detail the different strategies to resolve the conflicts and estimate their performance

[1] Current address: AT&T Bell Labs, Middletown, NJ.
This work has been partially supported by New York State Center for Technology in Computer and Information Systems under grant Number NYSSTF-CAT(87)-5.

In section 4, we outline the simulation program written to study quantitatively the performance of the most promising strategies. Section 5 outlines the application of global-read to database checkpoints, including a simple availability analysis. Finally, section 6 concludes the paper.

2 The Algorithm

2.1 Definitions and Introduction

A database is a set of *entities* [2]. Each entity may be individually read through shared locks or written under an exclusive lock. We will use the term *global-read* to denote an incremental query reading the entire database. Normal transactions on the database will be referred to as either *update transactions* or *read-only transactions*. Even though data movement may be in larger chunks (e.g. pages) for I/O efficiency, the lock granularity of the global-read and normal transactions is the same - entities.

The algorithm has three parts. First, we read entities one by one. Second, the global-read divides the entities in the database into two subsets; entities not yet read (*white*), and the ones already processed (*black*). Third, update transactions writing both white and black entities are not allowed to commit, because they cannot be serialized either before or after the global-read. For simplicity of presentation, we summarize only the algorithm to perform one global-read at a time. Concurrent global-reads may be used for totalizations and statistics in addition to checkpointing the database.

2.2 Basic Global-Read Algorithm

The following data structures are needed in the volatile storage, as an addition to the lock table:

- One *entity color bit* per entity. (Entities can only take one of two "colors", black or white.)
- One *paint bit* per database, used in a trick to repaint all entity color bits.
- Accompanying the paint bit we have a *global-read semaphore* to guarantee only one global-read runs at any one time.

At database (lock table) initialization time, the paint bit is copied onto all entity color bits. Global-Reads can start only after all entity color bits agree with the paint bit. We also assume that the update transactions will start only after the initialization is complete. In case of a crash, the recovery consists simply of a re-initialization.

Figure 1 describes the Basic Global-Read algorithm, which is incremental and works on-the-fly. The global-read's consistency is maintained by ensuring that all update transactions writing both white and black entities (*gray transactions*) are aborted. In order to enforce this rule, if a global-read is in progress, every update transaction needs to pass an additional *color test* before it can execute and commit. After the acquisition of all exclusive locks (before commit), the color bits of exclusively locked entities have to be checked. If all color bits are the same, the update can proceed, otherwise it is aborted. Please note that if no global-read is executing, all entity color bits are the same and the updates will always pass the color test. A formal proof of global-read consistency can be found in a previous paper [8].

3 Performance of Strategies

3.1 Basic Algorithm Performance

In the previous paper [8], we have shown that the gray transactions must be aborted. Therefore the basic algorithm is optimal in the sense that it aborts only the update transactions that cannot be serialized with respect to the global-read. However, a simple probabilistic analysis shows that the abort rate increases rapidly with the increasing number of entities being updated.

For simplicity, we assume that the update transactions access the entities in the database uniformly. This assumption ignores the locality of access, so our analysis is a conservative estimate. Since the interference exists only during an active global-read, let us consider a database with n entities, r of them painted black. An update transaction writing on k randomly chosen

```
{ Pre-condition: all entity color bits are the same as the paint bit (black). }
step 1:   P(semaphore)                  { Global-Read runs in a critical section. }
          change the paint bit.         { This re-paints all entities white. }
step 2:   while there are white entities
          do begin                      {This loop paints the white entities black. }
               if all white entities are exclusively locked        { Optimization. }
                    request shared lock on a white entity and
                    wait until lock is granted
               else lock any sharable white entity;
                    read entity, change entity color, release entity lock.
          end while                     { All entities are black, the same as the paint bit. }
step 3:   V(semaphore)                  { Let the next checkpoint go. }
```

Figure 1: Basic Global-Read

entities does not conflict with the global-read if all k entities are either white or black. The probability of this happening is:

$$P_{n,r}(k) = \frac{\binom{r}{k}}{\binom{n}{k}} + \frac{\binom{n-r}{k}}{\binom{n}{k}}.$$

Figure 2 shows this probability as a function of the r/n ratio for $k = 1, 2, 3$, and 4. Taking an example, the probability of non-interference of an update writing on 3 random entities, in the middle of a global-read, is slightly less than 1 in 4.

For the duration of a global-read, the average probability of abort is equal to the area above each curve integrated from 0 to 1. After some algebraic transformations, we obtain the following expression for the average probability:

$$P(k) = \frac{2}{n+1} \sum_{r=k}^{n} \prod_{i=0}^{k-1} \frac{(r-i)}{(n-i)}$$

Numerically calculating the probability for $k = 2, 3, 4, 5$ we obtain 0.333, 0.500, 0.600, 0.667, respectively. These numbers will be used in section 4.3 to validate the simulation program.

3.2 Turn-White Strategies

Besides the abort, there are two basic ways to resolve the conflicts between global-read and updates (summarized in table 1). First, we can turn the gray transactions white, forcing the global-read to backtrack and

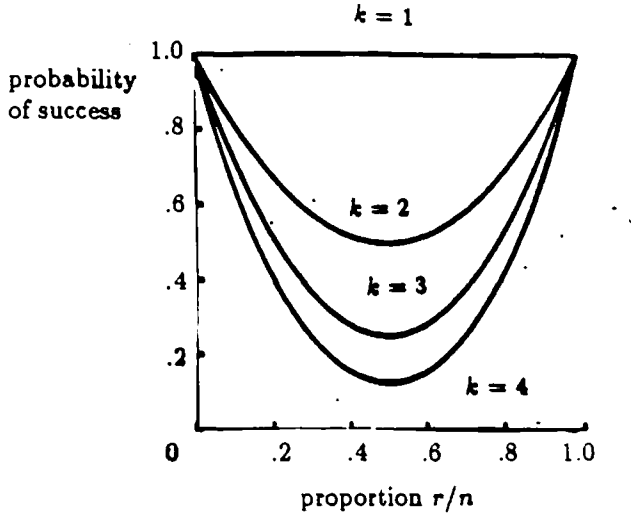

Figure 2: Non-interference Probability

read some black entities again. Second, we can turn the gray transactions black by making them wait. Here, we shall consider the turn-white strategy first. In section 3.3 we describe the turn-black strategy.

The main idea of turn-white approach is to make the global-read include the updated values from the gray transactions, serializing them before the global-read. The basic difficulty with this method is that the black entities in a gray transaction may have been updated by some black transactions. Therefore, if we turn the gray transaction white, we have to find those black transactions and turn them white, too.

Since the main advantage of two-phase locking concurrency control is to serialize database access without

explicitly calculating the dependencies between transactions, to maintain these dependencies only for the global-read will be impractical for two reasons. First, significant processing overhead will be necessary to calculate the dependencies at the transaction commit time for the locks held by the committing transaction. Second, considerable main memory would have to be dedicated to the maintenance of data structures storing the dependencies. Therefore, we do not consider this strategy (called Repaint-Some) further.

To avoid maintaining the dependencies, we might repaint all updated black entities white. This Repaint-All strategy effectively serializes the global-read after all update transactions. However, towards the end of global-read, when almost all updates are black, global-read may make little progress because of the re-reading. As soon as the rate of update surpasses that of reading in global-read, the global-read will be unable to complete. Since Repaint-All will not work for large databases with frequent updates we will not consider it further.

3.3 Turn-Black Strategies

Compared to the turn-white strategy, the turn-black approach is easier to implement. An update transaction, realizing that a global-read is in progress, checks the color of its entities before locking them. If an entity is white, the update waits until the entity has been painted black. In other words, we prevent the "white" transactions from turning gray by executing them as black transactions. To decrease the waiting time, the global-read may be modified to read the waited-for entities before the idle ones. In *step 2*, figure 1, instead of requesting a lock on any white entity, we look for those entities with updates waiting and read those first.

Near the beginning of a global-read, the above Wait-All strategy may be suboptimal for two reasons. First, the update transactions that started before the global-read may have already locked a white entity and therefore cannot turn black. Second, many white transactions that could have started and finished as white transactions now wait to turn black.

Instead of preventing white transactions from turning gray, an alternative avoids aborting the already gray transactions by passing their white entities to the global-read. This Save-Some strategy turns a gray transaction black by storing the before-images of its white entities in a buffer, and painting their entity color bits black. The global-read will see none of the then-gray transaction's updates, which are now all black. The white entities have been saved in the buffer for the global-read.

The only implementation complication of Save-Some is that at the beginning of a transaction, it may not know whether it is going to be white or gray. Consequently, all before-images of white entities should be saved in a private buffer before they are updated. If the transaction turns gray, the private buffer is transferred to the global-read buffer and the appropriate entity colors are painted black. At the beginning of the global-read, most transactions remain white. It is only towards the middle of the global-read that more transactions become gray and buffer transfers become intense. In a parallel or distributed environment, the global-read buffer can be distributed among several processing elements, minimizing the interference with normal transactions. In particular, if multiple disk/tape devices are available for global-read output, arbitrarily large transaction loads can be accommodated. We simply add as many output devices (and its buffer and processing element) as the peak gray transaction rate requires.

3.4 Ordered Reading

Since the global-read accesses the disk only for the necessary entity reading (and writing if copying the entity), its main cost is in memory requirements. Specifically, an entity color bit per entity may occupy significant areas of memory for large databases. As explained in the previous paper [8], reading the database entities in order allows the global-read to encode the entity color bits, thus reducing memory requirements. (The reading order may be a key or physical adjacency.)

However, both the turn-white and turn-black strategies depend on the ability to paint individual entity color bits to avoid conflicts. For example, Save-Some requires the gray updates to save the white entities over the entire database and paint their color bits. The

STRATEGY } / *Transaction Type*	Turn-White		Turn-Black	
	Global Remedy	Global Prevention		Local Remedy
	REPAINT-SOME	REPAINT-ALL	WAIT-ALL	SAVE-SOME
Gray	Repaint black entities		Wait for white entities to turn black	Save before-images of white entities
White	—	—	Wait for all entities to turn black	—
Black	Keep dependencies, repaint if necessary	Repaint all black entities	—	—
Global-Read	Read again selected black output	Read again all black output	Read waited-for entities first	Read saved before-images

Table 1: Summary of Abort-Avoiding Strategies

Wait-All strategy relies on the speedy reading of global-read, but since the global-read may be delayed more often in an ordered reading, the waiting of the updates will be even longer. Since the cost of main memory has been dropping the trade-off between memory and response-time would favor Save-Some and Wait-All.

4 Simulation Study

4.1 The Simulation Model

The main motivation for this simulation study is the difficulty in the probabilistic analysis of the abort-avoiding strategies. Therefore, the simulation study starts from the results of probabilistic analysis (section 3.1), which are used for validation, and investigates the (most promising) turn-black strategies.

Since our database is reasonably abstract, the simulation model follows closely the assumptions of the algorithm. A database is a set of entities; it supports shared read and exclusive write operations. Read-only transactions do not conflict with the global-read. Even though they may delay the update transactions, the net effect is the same as longer updates. Therefore, we have omitted read-only transactions from the simulation model.

The goal of the simulation is to compare different strategies regarding their throughput, response time, and number of aborted transactions. Of the three strategies being compared, only the Basic Algorithm aborts update transactions. To obtain the throughput, we run the simulation and count the number of successfully committed transactions. As we shall see in section 4.2, we have chosen to measure "time" (clock ticks between events) indirectly in the simulation program. So we estimate the response time by looking at the average waiting time of each transaction due to the global-read.

To isolate the above variables of interest, we assume that the database system has a constant multiprogramming level during the simulation, which starts and ends with a single run of global-read. To smooth out the

transient at the beginning of the simulation, we create a number of initial update transactions before starting the global-read. As soon as one of these transactions finishes, another is created to take its place. So global-read always executes with a constant number of competing update transactions.

Each update transaction locks k entities and releases them. In the simulation, we do not require significant amount of work to be performed by the update transactions. In other words, we assume the update transactions are traditional short transactions, in accordance to a database that supports multiple readers or an exclusive writer. Each successful lock acquisition implies a read or write, so it is counted as an I/O operation to the disk by an update transaction. Similarly, each entity locked by the global-read counts as a read operation plus a write operation.

4.2 The Simulation Program

The simulation model was implemented using Concurrent Euclid (CE) [4]. CE supports concurrent processes and monitors for their synchronization. Global-Read runs in its own process, and the other processes execute update transactions. Each CE process may be seen as a transaction manager, running on the same or different physical processors. The multiprogramming (multiprocessing) level in our study is ten. A monitor containing the lock table maintains concurrency control.

If a transaction process requests locks being held by another transaction, then the simulation program puts the requester on a waiting queue. When the lock is released, the first process in the queue gets the lock and proceeds. The waiting time is counted in terms of implied I/O operations in the acquisition of locks. We avoid deadlocks by making update transactions lock the entities in the same order. This deadlock avoidance may decrease the conflicts with the global-read somewhat, since randomly chosen entities may have mixed colors but ordered choice would be black. We use a counter to detect this situation and the number of transactions in this category is negligible, the reason being the slow progress rate of global-read – about one entity traversed for each completed update transaction.

For a run consuming more than a few seconds of CPU time, the CE kernel scheduling of processes approximates round-robin very closely. Since all of our transactions are exactly the same, locking the same number of entities and consuming the same CPU resources, we decided to factor out the CPU part. Instead of using the CE `sleep` command to count time (which does busy wait), we use an artificial clock tick. The clock used is the total amount of I/O operations the simulation has counted from all transactions, including the global-read.

Our I/O model is abstract. Each I/O operation is counted as a unit of cost. In a sense we are using the average cost for each I/O operation. Admittedly this is a simplification. Using a realistic configuration would be more precise, but also make the conclusion specific to the machine we are simulating. The abstract I/O model applies to just about all conventional machine configurations, although at a lesser precision.

An independent monitor maintains the accounting data. The measurement data collected by the simulation program are:

- total I/O operations,
- waiting time per transaction, and
- total number of aborted and committed update transactions.

Accounting data are printed out in a log file. The statistics on I/O operations, waiting time, and update transactions are saved at regular intervals as the global-read progresses and calculated at its end.

The entities being locked by the update transactions are chosen uniformly from the set of entities by a pseudo-random number generator. To make the comparison between strategies more direct, we run the different strategies with the same seed to the pseudo-random number generator, causing the same sequence of entities to be locked by all strategies.

4.3 The Simulation Results

Simulation results shown are numbers averaged over several runs (between 10 and 20), with the results

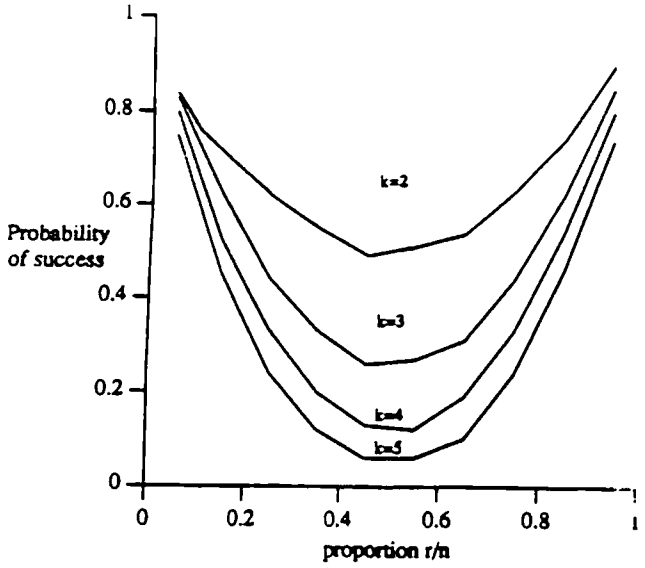

Figure 3: Validation of Simulation Program

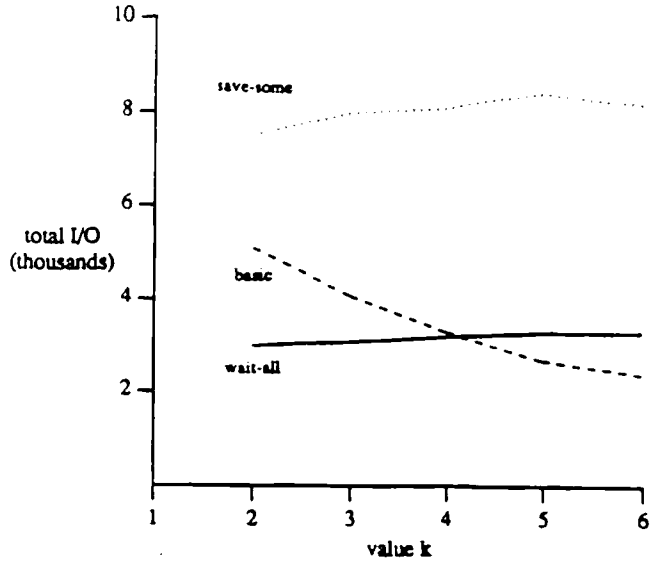

Figure 4: Total I/O Operations

within the 98% confidence interval.

In the first place, we ran the Basic Algorithm to validate the simulation program using the analytical results from Section 3.1. In this case, we ran each value of k independently. Each update transaction chooses k entities randomly from the total (1000) and locks them. The global-read chooses the next unlocked entity (according to number) and reads/paints it. Update transactions are aborted if their k entities contain both black and white entities. Simulation results are shown in the graph in figure 3, which matches well with the theoretical values in figure 2.

The main disadvantage of the Basic Algorithm is the need to abort update transactions. Both the simulated turn-black strategies avoid aborting the update transactions. Table 2 shows the abort ratio for $k = 1$ to 4 of the Basic Algorithm. The theoretical values fall within the range of values in Table 2. This is the second validation of the simulation program using the probabilistic analysis.

Once we have obtained the basic validation of the simulation program, we took the next variable of interest, the total amount of I/O activity allowed by each strategy. From figure 4, we can see that of the three strategies, Save-Some allows consistently high I/O rate, Wait-All consistently low I/O rate, and the Basic Algorithm from middle to lowest as k increases. Since the multiprogramming level is the same for all strategies, the difference in I/O rate reflects the difference in concurrency allowed.

The curves in the graph admit an intuitive explanation. The Wait-All strategy's constant low I/O rate is easy to explain. Since we have a relatively high transaction rate, global-read becomes the bottleneck delaying the updates waiting for it to paint the white entities black. The Save-Some strategy's constant high I/O rate is due to its high throughput, which we will see in figure 6. The decline of the Basic Algorithm I/O rate is due to the decline of its throughput as k increases, as shown in table 2.

The second variable of interest is the amount of time update transactions wait for each other and the global-read. Since we start all the strategies with the same seed for the pseudo-random number generator, we expect the strategy to play a major role in determining the amount of waiting in the system. Figure 5 shows that Wait-All does require the longest waiting time over the Basic Algorithm and Save-Some. At first, we were surprised that Save-Some caused as little waiting as the Basic Algorithm, since the only waiting in the Basic Algorithm is between update transactions. But remembering that Save-Some allows the update to proceed once the white-turned-black entities have been placed

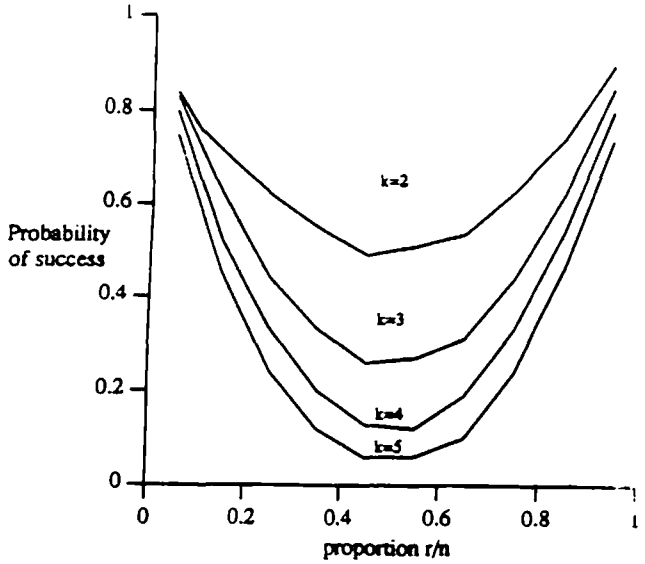

Figure 3: Validation of Simulation Program

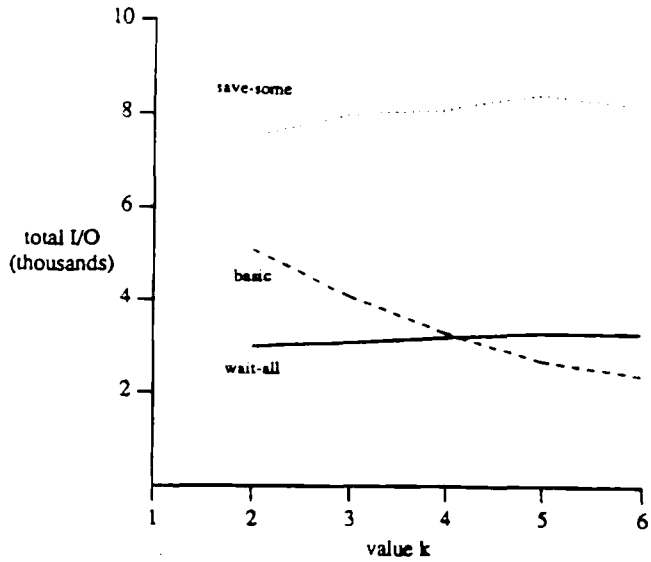

Figure 4: Total I/O Operations

within the 98% confidence interval.

In the first place, we ran the Basic Algorithm to validate the simulation program using the analytical results from Section 3.1. In this case, we ran each value of k independently. Each update transaction chooses k entities randomly from the total (1000) and locks them. The global-read chooses the next unlocked entity (according to number) and reads/paints it. Update transactions are aborted if their k entities contain both black and white entities. Simulation results are shown in the graph in figure 3, which matches well with the theoretical values in figure 2.

The main disadvantage of the Basic Algorithm is the need to abort update transactions. Both the simulated turn-black strategies avoid aborting the update transactions. Table 2 shows the abort ratio for $k = 1$ to 4 of the Basic Algorithm. The theoretical values fall within the range of values in Table 2. This is the second validation of the simulation program using the probabilistic analysis.

Once we have obtained the basic validation of the simulation program, we took the next variable of interest, the total amount of I/O activity allowed by each strategy. From figure 4, we can see that of the three strategies, Save-Some allows consistently high I/O rate, Wait-All consistently low I/O rate, and the Basic Algorithm from middle to lowest as k increases. Since the multiprogramming level is the same for all strategies, the difference in I/O rate reflects the difference in concurrency allowed.

The curves in the graph admit an intuitive explanation. The Wait-All strategy's constant low I/O rate is easy to explain. Since we have a relatively high transaction rate, global-read becomes the bottleneck delaying the updates waiting for it to paint the white entities black. The Save-Some strategy's constant high I/O rate is due to its high throughput, which we will see in figure 6. The decline of the Basic Algorithm I/O rate is due to the decline of its throughput as k increases, as shown in table 2.

The second variable of interest is the amount of time update transactions wait for each other and the global-read. Since we start all the strategies with the same seed for the pseudo-random number generator, we expect the strategy to play a major role in determining the amount of waiting in the system. Figure 5 shows that Wait-All does require the longest waiting time over the Basic Algorithm and Save-Some. At first, we were surprised that Save-Some caused as little waiting as the Basic Algorithm, since the only waiting in the Basic Algorithm is between update transactions. But remembering that Save-Some allows the update to proceed once the white-turned-black entities have been placed

k	2	3	4	5	6
Transactions Created	23099 ± 40	16539 ± 18	12488 ± 37	10244 ± 36	7114 ± 23
Percentage of aborts	$33 \pm 0.8\%$	$49 \pm 0.6\%$	$60 \pm 0.4\%$	$68 \pm 0.1\%$	$72 \pm 0.1\%$

Table 2: Basic Algorithm Transaction Aborts

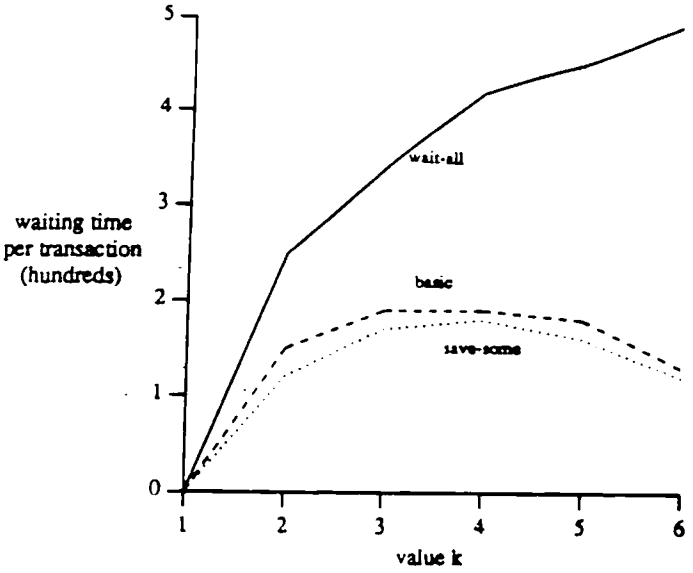

Figure 5: Waiting Time per Transaction

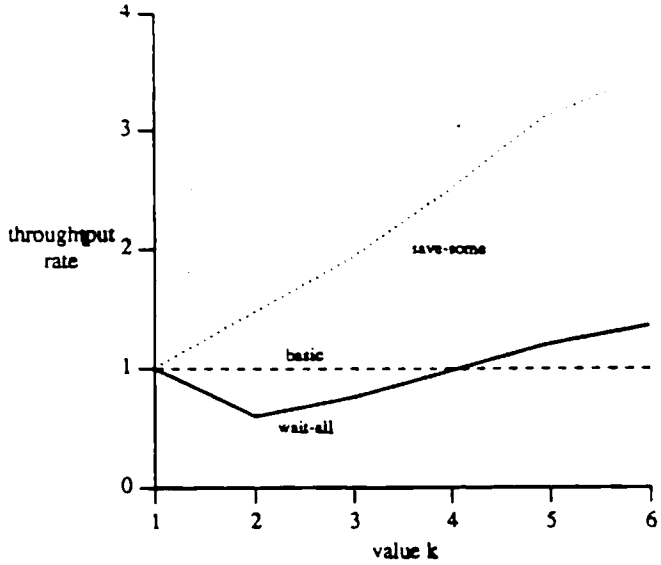

Figure 6: Relative Throughput

in the buffer, then the amount of waiting is negligible in the simulation based on I/O operations. Therefore, Save-Some seems to be the strategy of choice.

Finally, the total throughput of the three strategies is compared in figure 6. Since the throughput of each strategy varies with k, we have normalized the graph for easy visualization. We use the throughput of the Basic Algorithm as the norm, and compare the other two strategies relative to it. At $k = 1$ there is no interference and the three strategies are equivalent. At $k = 2$ Save-Some is the best, Basic in the middle, and Wait-All performs the worst. As k increases, Save-Some wins by far and Wait-All takes over Basic gradually. Since Save-Some has very low waiting time and high I/O rate, its highest throughput is not surprising. Wait-All's throughput is limited by the I/O that global-read can provide, so as the Basic Algorithm's throughput declines (table 2), Wait-All overtakes the Basic Algorithm.

5 Checkpoint and Availability

5.1 Asynchronous Checkpoints

One application of global-reads is the asynchronous checkpointing of databases. Fischer et al. [3] have mentioned several applications using global-reads, such as checking consistency constraints in a database, and media recovery. However, like their global checkpoint, our global-read is consistent but may not reflect any schedule based on chronological order. Consider a global-read that started at time t_1 and terminated at t_2. The global-read will reflect all updates committed before t_1.

plus all white transactions which must terminate before t_2. In other words, the global-read may include "later" white transactions but not "earlier" black transactions. This characteristic should not affect applications like totals, statistics or consistency checking, where the actual transaction scheduling is not important.

In order to use a backup copy made by our global-reads to recover from media failures, a log containing the committed transactions is still necessary. Logs in both shadow pages and fuzzy dump methods are logs of actions on "physical addresses" because their backup copies are not necessarily consistent. Since a backup copy made with our algorithms is transaction-consistent, we need only logs that contain transaction actions. There are two possibilities for recovery. First, we can redo the black transactions onto the backup copy to reach the database at t_2. Alternatively, we can undo the white transactions from the backup to find the database at t_1. In either case, in addition to actions, the log must include the color of each transaction.

5.2 Availability Analysis

Studies on the performance of backup procedures [6,10] have assumed that update transaction processing is not allowed during the backup copying time (synchronous checkpoints). Several optimization criteria and optimal checkpoint policies [5] are based on the above assumption, trading interrupted transaction time for short recovery time. In contrast, our algorithm provides overall consistent availability with little interference.

The increase of availability using global-read may be significant. For example, Tantawi and Ruschitzka [10] define the system availability in the case of their checkpoint strategies as

$$A = 1 - n(a + b * E(X) + t)$$

where a and b represent the initial loading time for the database and the proportionality factor for reprocessing, respectively. During a fixed interval of real time, there are on average n failures. For each failure the mean error recovery time is $E(X)$ and the total checkpointing time (with a quiescent system) of t. Since mean error recovery time is defined as half the interval between checkpoints, we can use their equation to calculate the benefits of using global-read for backup and recovery. Global-Read makes $t = 0$.

If the global-read interfered significantly with the normal transaction processing, we would have to take the interference into account when calculating the system availability. For example, the Basic Algorithms makes a good percentage of the update transactions "unavailable" for a good interval of time. However, in the case of Save-Some, there is no interference other than resource consumption in the form of buffer and I/O requirements. In a database with parallel hardware the global-read would be able to proceed with enough buffers and I/O bandwidth. It is under this assumption that we compare synchronous and asynchronous checkpoints directly.

This point was brought home in a recent study comparing different checkpointing algorithms on main memory databases [9]. According to the authors, "Most of the [high] cost comes from rerunning transactions that are aborted for violating the two-color restriction." But even the basic algorithm has cost and performance comparable to other database checkpoint algorithms, such as Copy-on-Update Checkpoints by Dewitt et al. Since the abort-avoiding strategies (e.g. Save-Some) show significant gains compared to the Basic Algorithm, we expect more interesting results from comparing Save-Some to the other checkpointing algorithms.

In a numerical example [10], the optimal system availability with equidistant checkpoints is 0.9818 for an inter-failure time $1/n = 60$ hours, a mean checkpoint time of 1 minute per interval, a restart and loading time of $a = 6$ minutes, a reprocessing proportionality factor $b = 0.5$ and the optimal checkpoint interval of 118.9 minutes. If we simply substitute the global-read for the quiescent checkpoint, we obtain an availability of $0.9818 * (1 + 1/118.9) = 0.9901$ by gaining the checkpoint time of 1 minute. To improve the system availability further, we can reduce the interval between global-reads to 30 minutes, since there is no checkpoint quiescent time. Substituting the numbers in the equation, we have $1 - (1/3600)(6 + 0.5 * 15) = 0.9963$ which is significantly higher.

6 Conclusion

We have studied the performance of the global-read algorithm to checkpoint entire databases on-th-fly using a combination of simple probabilistic analysis and simulation. The global-read algorithm does not voluntarily abort, does not cause deadlocks, does not incur excess writes to disk, and terminates given a fair lock management. The Save-Some strategy reads and writes each entity only once for the entire global-read and avoids aborting the update transactions. We have written a simulation program validated by probabilistic analysis. Using the simulation study, we have shown that the Save-Some strategy allows high concurrency, and causes negligible delays in update transaction response time.

According to recent research [9], the main cost of the global-read algorithm is due to rerunning the aborted transactions, not the global-read itself. Since Save-Some avoids aborts with little additional cost, we believe that the global-read algorithm with the Save-Some abort avoiding strategy is a promising solution to incremental, on-the-fly, consistent reading of entire databases. Since global-read algorithms are parallel in nature and they only compete with normal transactions in memory buffer and disk I/O, we expect the global-read algorithm to be even more useful for databases running on parallel or distributed hardware.

7 Acknowledgment

We would like to thank Peter Caiazzi for implementing the deadlock avoidance mechanism, Michael Carey for suggesting the simulation approach and A. A. Helal for encouragements. One of the referees made detailed comments that improved significantly the presentation of the paper.

References

[1] R. Bayer, H. Heller, and A. Reiser.
Parallelism and recovery in database systems.
ACM Transactions on Database Systems, 5(2):139–156, June 1980.

[2] 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 ACM, 19(11):624–633, November 1976.

[3] M.J. Fischer, N.D. Griffeth, and N.A. Lynch.
Global states of a distributed system.
In *Proceedings of Symposium on Reliability in Distributed Software and Database Systems*, July 1981.

[4] R.C. Holt.
Concurrent Euclid, The Unix System, and Tunis.
Addison-Wesley Publishing Company, 1983.

[5] C.M. Krishna, K.G. Shin, and Y.H. Lee.
Optimization criteria for checkpoint placement.
Communications of ACM, 27(10):1008–1012, October 1984.

[6] G.M. Lohman and J.A. Muckstadt.
Optimal policy for batch operations: backup, checkpointing, reorganization and updating.
ACM Transactions on Database Systems, 2(3):209–222, September 1977.

[7] Calton Pu.
On-the-fly, incremental, consistent reading of entire databases.
In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, pages 369–375, Stockholm, August 1985.

[8] Calton Pu.
On-the-fly, incremental, consistent reading of entire databases.
Algorithmica, 1(3):271–287, October 1986.

[9] K. Salem and H. Garcia-Molina.
Checkpointing Memory-Resident Databases.
Technical Report CS-TR-126-87, Department of Computer Science, Princeton University, December 1987.

[10] A.N. Tantawi and M. Ruschitzka.
Performance analysis of checkpointing strategies.
ACM Transactions on Computer Systems, 2(2):123–144, May 1984.