Department of Computer Science Technical Reports

Department of Computer Science

1990

Adaptability Experiments in the RAID Distributed Database System

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Karl Friesen

Abdelsalam Helal

John Riedl

Report Number:
90-972

ADAPTABILITY EXPERIMENTS IN THE RAID DISTRIBUTED DATABASE SYSTEM

Bharat Bhargava
Karl Friesen
Abdelsalam Helal
John Riedl

CSD-TR-972
April 1990

Adaptability Experiments in the RAID Distributed Database System[1]

Bharat Bhargava
Karl Friesen
Abdelsalam Helal
John Riedl

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
(317) 494-6013

[1] This research is supported in part by a grant from AIRMICS, and National Science Foundation grant IRI-8821398

Abstract

Adaptable systems can improve reliability and performance by allowing dynamic reconfiguration. We are conducting a series of experiments on the RAID distributed database system to study the cost and performance implications of providing static and dynamic adaptability, and for increasing the availability of data items. Our studies of the cost of our adaptable implementation were conducted in the context of the concurrency controller and the replication controller. The experimentation with dynamic adaptability focuses on concurrency control, and our examination of the costs of providing greater data availability studies the replication control and atomicity control subsystems of RAID. We show that for concurrency control and replication control, adaptable implementations can be provided at costs comparable to those of special purpose algorithms. We also show that for our concurrency controller dynamic adaptability can result in performance benefits and that system reconfiguration can be accomplished dynamically with less cost than stopping the system, performing reconfiguration, and then restarting the system. In some cases, reconfiguration could be performed without aborting any transactions. We demonstrate some costs associated with increasing availability through replication control methods and use of a three-phase commit protocol. A system that can dynamically change to algorithms that increase availability can result in a 25-50% performance improvement over systems that continuously employ the algorithms that provide the better availability. We show that the algorithms selected for replication control can significantly impact the time required for transaction commitment.

Contents

1 Introduction

Adaptability and reconfigurability are needed to deal with the changing performance and reliability requirements of a distributed system. An adaptable system can meet a variety of application needs in the short term, and can take advantage of advances in technology over the years. There are many aspects of adaptability that have been studied. The issues surrounding adaptable systems include the performance costs of utilizing an adaptable implementation, the costs of dynamic adaptation, and the problem of deciding when to perform system adaptation [BR89a]. In addition, there are questions regarding the selection a good mix of algorithms for a given transaction stream, and how to perform dynamic reconfiguration when site failures and network partitions occur. We are conducting scientific experiments on the RAID distributed database system that focus on the performance costs of providing an adaptable implementation, specific costs of dynamic adaptability and costs attributable to increased data availability.

The remainder of this section is devoted to a brief description of RAID. RAID is an experimental distributed database system [BR89b] developed on SUN workstations under the UNIX operating system. RAID has proven useful in supporting experiments in communication [BMR87, BMR91], adaptability [BMR89], and transaction processing [BR89b]. However, several new features were desired to support extensive experiments in adaptability and reliability. To achieve these goals the RAID group has changed and re-implemented the control flow for transaction processing, and created a second version of the system called RAID-V2. Version 2 has the same principles and goals as version 1, but takes advantage of the lessons learned from the original RAID implementation to offer improved support for adaptability and reliability. The details of version 2 can be found in [BFH$^+$90].

There are six major subsystems in RAID-V2: User Interface (UI), Action Driver (AD), Access Manager (AM), Atomicity Controller (AC), Concurrency Controller (CC), and Replication Controller (RC). Figure 1 depicts the latest version of the RAID[1] system.

The major differences between RAID-V1 and RAID-V2 are:

- On-line replication control
- Facilities for partial replication
- On-line concurrency control
- Improved flow of control for adaptability
- Use of XDR to support communication in a heterogeneous system

[1] In the rest of the paper RAID will be used to mean RAID-V2. RAID-V2 will be used only for emphasis or to improve clarity.

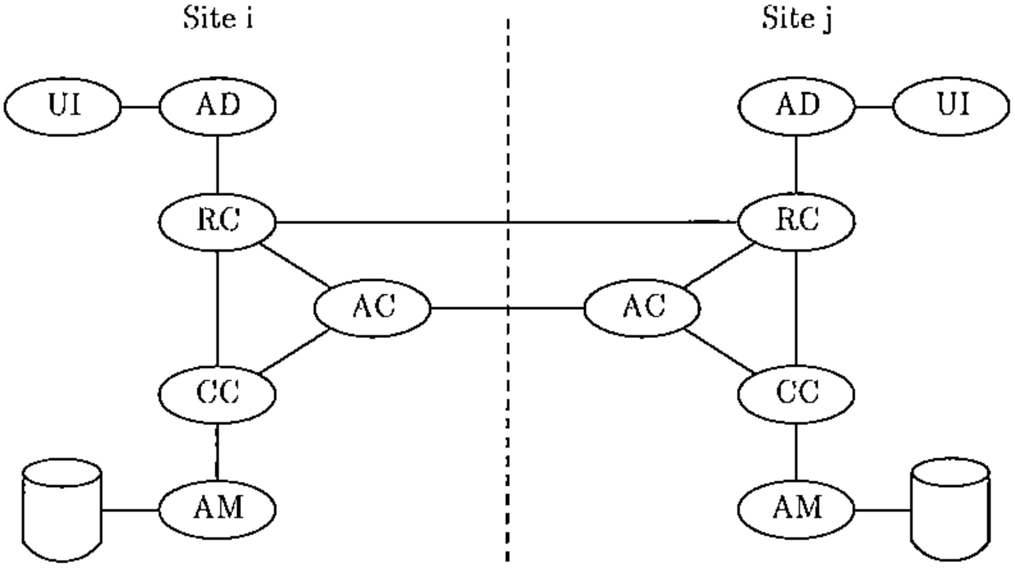

Figure 1: Communication paths in RAID-V2.

There are three other projects that facilitates experimental work in our laboratory. Mini-RAID [BNS88] allows the prototyping of new ideas prior to their implementation in RAID. Seth [HSB89] is a system that supports research in quorum-based replication methods. Push [BMR89] is a utility that allows users to implement kernel-level services to enhance database performance. Other academic experimental systems that have been developed include Locus/Genesis [PW85, TWPWP85], Camelot [S+86], Argus [LCJS87], the V distributed system [Che88], and the Synthesis kernel [PMI88].

In section 2, we describe our experimental infrastructure. This includes discussion of our benchmarks, transaction restart policies, and procedures for running experiments. In section 3, we report our findings on the costs that must be paid to use an adaptable implementation using the concurrency controller as an example. We also explore the benefits of dynamic adaptability in the concurrency controller. Finally, we measure the costs incurred by using methods which increase the availability of data, comparing different quorum methods in the replication controller, and different commit protocols in the atomicity controller. In section 4, we summarize our conclusions and outline future experimental work.

2 Experimental Infrastructure

In this section we describe the experimental infrastructure of the RAID project at Purdue University. The Raid laboratory has five Sun 3/50s, and four Sun SparcStation-1s, all with local disks connected by a 10Mb/s Ethernet. The SparcStations were aquired recently so some of the reported experiments were done on Sun 3/50s. Measurements are facilitated by microsecond resolution timers that were obtained from Zytec Corporation. Adaptability features in RAID make it possible to test different algorithms and implementation techniques under the same conditions using the same benchmarks. A single independent variable can be chosen, and can be varied over a range of values while the rest of the system remains constant. For instance, many different replication controllers can be tested with the same atomicity controller, concurrency controller, and access manager, and under the same workload. This provides a fair comparison between the performance of the different implementations. In the following subsections, we discuss the benchmarks for distributed databases that we developed by extending the DebitCredit benchmark [A$^+$85]. We outline the action driver simulator which parametrizes and applies the benchmark to the RAID system. We also describe the transaction restart policy and its effects on the stability of experiments. Finally, the RAID experimental procedure is detailed.

2.1 Adaptability Features in RAID

Three of the RAID servers have built in adaptability features — the concurrency controller (CC), the replication controller (RC), and the atomicity controller (AC). Each of these servers implements a number of algorithms and has the mechanism necessary to convert from one algorithm to another.

The CC implements five algorithms for concurrency control: timestamp ordering (T/O), two-phase locking (2PL), generic timestamp ordering (gen-T/O), generic locking (gen-2PL) and generic optimistic (gen-OPT). The first two algorithms are implemented using specialized data structures, while the last three use the same general data structures. In the case of T/O and gen-T/O, the implementations enforce the same concurrency control policy, but one uses a generic data structure specifically designed for adaptability, while the other uses an ad hoc data structure designed specifically for T/O.

The RC also implements several algorithms to perform replication control. Among these are the read-one-write-all (ROWA) algorithm and a quorum consensus (QC) algorithm. In QC, quorum parameters can be chosen to have a quorum version of ROWA (QC-ROWA), a quorum version of read-same-as-write (QC-RSW), and a quorum version of read-all-write-one (RAWO). Quorum consensus methods can model many of the standard replication control policies, and all such methods have a common structure.

The AC implements centralized two-phase commit (2PC) and centralized three-phase commit (3PC). Transactions in the AC are independent of each other, so the selection of a commit protocol can be performed on a per-transaction basis. In practice, this selection is done by the RC, which may elect to utilize the AC default protocol.

```
begin
    update teller <teller-id> by <value>
    update branch <branch-id> by <value>
    update account <account-id> by <value>
    insert history <teller-id> <branch-id> <account-id> <value>
end
```

Figure 2: The basic DebitCredit transaction.

2.2 Benchmark Data

Several benchmarks for database systems exist [BDT83, A$^+$85]. However, for distributed database systems there are no well-accepted benchmarks. The data and the workload can be distributed among the sites in many different ways, especially in systems that support data replication. Distributed systems vary widely in their model of transactions, including support for concurrency control, reliability, and replication. Designing general benchmarks for different systems presents a difficult problem for benchmark developers.

DebitCredit Benchmark The DebitCredit (or TP1 or ET1) benchmark is described in [A$^+$85]. DebitCredit is intended to be the simplest realistic transaction processing benchmark. There is only one form of DebitCredit transaction, representing a simple banking transaction. This transaction reads and writes a single tuple from each of three relations: the teller relation, the branch relation, and the account relation. In addition, a tuple is appended to a special write-only sequential history file describing the transaction. Figure 2 shows a DebitCredit transaction. The benchmark requires that the entire transaction be serializable and recoverable [BHG87].

The teller, branch, and account tuples are 100 bytes long, and the history tuples are 50 bytes long. Each teller, branch, and account tuple must contain an integer key and a fixed-point dollar value. Each history tuple contains a teller, branch, and account id as well as the relative dollar value of the transaction.

Extensions to the DebitCredit Benchmark The DebitCredit benchmark is designed for a variety of sizes, with the database size decreasing with the strength of the transaction processing system. Table 1 summarizes the sizes of the various relations for different transaction processing speeds. DebitCredit is designed so the branches and, to a lesser extent the tellers, form a hot spot in the database. In our experiments we wanted to vary the hot-spot size, so we chose to make all relations the same size and explicitly choose a hot spot size

Table 1: Sizes of DebitCredit relations.

TPS	branches	tellers	accounts
> 10	1000	10,000	10,000,000
10	100	1,000	1,000,000
1	10	100	100,000
< 1	1	10	10,000

for each experiment. Since the experiments were performed on low-end machines we use the database size for one transaction per second, and make all three relations 100 tuples.

The DebitCredit benchmark specifies that the database system should perform data-entry from a field-oriented screen. In the laboratory environment, we instead simulate transactions by randomly generating four values: teller id, account id, branch id, and relative value.

In order to obtain a greater variety of transaction streams, we extended the DebitCredit benchmark to support changes in transaction length, percent of accesses that are updates, and percent of accesses that are to hot-spot items. Each transaction consists of some number of actions, each of which accesses a random tuple of one of the three relations. The access is either an update of the balance field or a select on the key field. Some percentage of the updates are directed to a hot-spot of the relation, which is the first few tuples of that relation. Each transaction that performs at least one update ends with an insert to the history relation.

We built a transaction generator to generate a stream of random DebitCredit transactions based on the following input parameters:

- transactions: number of transactions to generate.
- branches: number of tuples in branch relation.
- tellers: number of tuples in tellers relation.
- accounts: number of tuples in accounts relation.
- average length: average number of actions in a transaction.
- probability long: probability that average length is used for a particular transaction. Otherwise one-fifth average length will be used. The default for the probability is one, which creates a unimodal distribution around average length. Transaction length is a normal distribution, with standard deviation 1/3 the length.
- update percent: percent of the actions that are updates rather than just selects.

- hot-spot size percent: percent of the database comprising the hot-spot. Each action is checked to see if it should be a hot-spot action. If so, it accesses tuples number $[0, \ldots,$ hot-spot size percent $*$ n_tuples$]$ for the chosen relation. Note that all relations have the same hot-spot size.

- hot-spot access percent: the chance that an action on a relation will access the hot-spot of that relation.

The transaction length is bimodal, in an attempt to reflect real systems with a mix of large and small transactions. The standard deviation was chosen so that zero is three standard deviations away from the average, to decrease the number of times the normal distribution has to be truncated to avoid zero-length transactions. The hot-spot is over a fixed number of tuples across three relations. Since RAID does tuple-level locking, this is the same as having a single hot-spot in one of the relations.

This modified version of the DebitCredit benchmark is no longer restricted to a real banking application as in the original benchmark. However, it uses the same relations and the same types of actions as the original benchmark, and has the advantage of supporting transaction streams with heterogeneous characteristics.

Data Distribution RAID supports a wide range of distributions of data among sites, with differing transaction processing characteristics. Since the benchmarks are created for single-site database systems we measured a variety of different data and workload distributions.

The key in the data distribution is the degree of replication. The range is from full replication, in which each site has a copy of all data, to no replication, in which only one copy of each item is shared among the sites. Most real systems are likely to use something in between, balancing the need to have copies in case of site failures with the performance cost of performing operations on multiple sites. In our experiments we used both partial and full replication.

2.3 Action Driver Simulator

The AD simulator is used –in place of the RAID AD– to provide an easily controllable workload. It processes transactions in a special benchmark language. The commands in the language all consist of a single line starting with a verb and ending with a number of arguments. The verbs are `begin`, `end`, `update`, `select`, `insert`, `update-rel`, and `delete`. Figure 3 shows the arguments for the verbs. Each verb except for `begin` and `end` takes a database id and a relation id as its first two arguments, so these are left out of the table. The meaning of the verbs are:

- `begin`: begin a transaction.

Verb	arguments
update	<select col> <select key> <update col> <update value>
insert	<new tuple value>
delete	<select col> <select key>
select	<select col> <select key>
begin	none
end	none

Figure 3: Simple benchmarking language

- **end**: begin commit processing for a transaction.

- **update**: reads tuples from the database that match a key value in a particular column. It then changes the value of the attributes of those tuples in some other column. `update` and `update-rel` are only supported for integer and float columns.

- **update-rel**: (relative update) same as update, except that the value is added to the attribute in the update column rather than just replacing the old value.

- **insert**: insert a tuple into the relation.

- **delete**: retrieve all tuples from a relation that match a key value in a particular column, and delete them.

- **select**: retrieve all tuples from a relation that match a key value in a particular column.

Transactions are specified by enclosing a number of other actions in a `begin/end` pair. All of the transactions from the DebitCredit benchmark can be expressed in this language.

The AD simulator accepts a parameter β that specifies the inter-arrival rate of the transactions. β is used as the average for an exponential random variable. When an arrival occurs the AD parses the next transaction from the input file, creates an AD transaction data structure for the new transaction, and begins executing it by issuing commands to the RC. When a transaction completes, statistics are compiled on its execution profile and its data structure is returned to a common pool. When the file is empty the AD simulator waits for the active transactions to complete, and prints its statistics.

The AD runs a timer for each transaction. It maintains a delta list[2] of alarms of various types. Depending on the state of the transaction, the timer can be of type life-over, restart,

[2] A delta list is a list of times in increasing distance from the present [Com84]. The time for an element tells how long after the preceding element an alarm should occur.

or ignore. Life-over timeouts cause a transaction to abort and restart[3]. These timeouts are used to resolve deadlocks and to handle lost messages. Restart timeouts cause a previously aborted transaction to restart. Ignore timeouts are used to safely disable the alarm that is currently active. Removing the alarm from the head of the alarm queue is not safe, since the alarm signal may already be on its way. Arrivals are also handled with special alarms of type arrival that are not associated with a transaction. The life-over alarms represent a transaction timeout, presumably because of deadlock or lost messages. The timeout interval is a constant number of milliseconds per action, chosen to maximize system throughput.

Control Relation Some of the experiments require that algorithms be changed dynamically while RAID is processing transactions. To support dynamic adaptability, each RAID database has a special control relation. This relation contains one tuple for each site, containing one column for each server type. Each attribute is an integer representing the state of a particular server on a particular site. The interpretation of the integer is different for each server type, but in general the integer specifies the algorithm being executed by the server. The control relation is write-only[4], and is updated by special control transactions issued by the AD. Each of these control transactions accesses only the control relation.

The control transactions are processed like normal transactions until they are committed. At this point, each server checks to see if the transaction is a control transaction for that server. If so, the server selects the integer from the column corresponding to the server's logical site id, and interprets it independently. The control relation is fully replicated, and is set up by the replication controller so that writes occur on all sites. Since control transactions are serialized just like other transactions, there is automatic protection against multiple operators introducing an inconsistent state. Furthermore, control transactions synchronize the adaptation in serialization order.

Control transactions originate in the AD simulator by normal update transactions in the input file that use the tuple id instead of a key to select tuples. Usually each control transaction writes all tuples in the control relation, changing the value for one server on all sites.

2.4 Open versus Closed Experiments

The AD simulator is set up to run two basic types of experiments. In open experiments the transaction inter-arrival gap is varied to control the system load. In closed experiments the multiprogramming level is fixed. When one transaction completes another is started. Open experiments are more representative of the type of load found in on-line transaction systems.

[3] If the transaction is already in commitment it may not be abortable. In this case, the AC takes the abort request as a timeout and retries the commit request for the transaction.

[4] Actually, the relation can be read to learn the state of the server during debugging, but its value is not used by the servers.

Arrivals are separated by an exponential random variable, representing, for instance, arrivals of customers to a teller. Actual applications probably fall somewhere in between these models, behaving like an open system when the load is low, and behaving like a closed system when the load is high.

We ran a series of open and closed experiments on concurrency controller and compared the information returned, to the accuracy of the confidence intervals. The results of the closed experiments were consistently easier to understand and interpret than the results of the open experiments. The problem is that at a high degree of concurrency an open system is very unstable, and at a low degree of concurrency the choice of concurrency controller does not matter since there are very few concurrency conflicts [CS84]. In summary, it is difficult to maintain a high degree of concurrency over a range of independent variable values in an open experiment, which makes it difficult to gather experimentally meaningful results.

Our experiments, described in section 3 are performed on a closed system. For the concurrency control experiments using a closed system makes maintaining a high degree of multi-programming easier, which allows a better exploration of the differences between concurrency controllers over a wide range of parameter values. For the atomicity control and replication control experiments using a closed system makes it easier to maintain a constant low multi-programming level without running as many pilot experiments to establish a reasonable inter-arrival gap.

2.5 Experimentation with Restart Policies

In most applications, the successful completion of transactions is required. In such applications, transactions aborted by the transaction manager must be retried until they succeed. In order to model such behavior in our experiments, transactions aborted by the system were restarted by the AD.

Performance is sensitive to the restart policy used, since restart occurs most often during high periods of conflict, and restarting transactions raises the degree of multiprogramming. Also, if transactions are restarted too quickly they are likely to again conflict with the same transactions that caused the original restart. In RAID we delay restarts to improve the chance that the conditions that caused the restart will have changed when the transaction restarts.

We studied eight different restart policies based on three binary attributes: rolling average versus total average response time for the mean restart delay, exponential random versus constant delay, and ethernet backoff[5] versus non-increasing backoff. The total average used the average response time since the system was started as the mean restart delay. By contrast, rolling average estimated the average response time of the last few transactions. Exponential ransom delay used the average response time as the mean of an exponential

[5] We call this ethernet backoff rather than exponential backoff to avoid the name conflict with exponential random.

random variable used to compute the actual restart delay. Constant delay uses the average response time directly. The ethernet backoff policy doubles the restart delay after each time an individual transaction is aborted.

We found that using the combination of non-increasing backoff, rolling average and exponential random delay methods resulted in a restart policy that was responsive and that maintained system stability. This is the restart policy that we used for all of our other experiments.

2.6 RAID Experimental Procedure

All experiments are run early in the morning, when network activity is low. All of the RAID machines are first rebooted to ensure that the experiments will run on a "clean" system. Shortly after a machine reboots a user cron[6] job runs a shell script[7] that sets up the experiments. This shell script reads a special directory and executes any benchmark files there. Each line of a benchmark file represents a complete invocation of RAID to process transactions from a temporary transaction file. After a RAID instance terminates, the data from the run is collected and stored. Figure 4 shows the logic of the experiment script, figure 5 shows the logic of the script that runs a single experiment, figure 6 shows the logic of the script that runs dynamic adaptability experiments, and figure 7 shows the logic of the script that runs a single instance of RAID.

The raw data is processed each morning with an AWK program [AKW]. This program scans the directory where the data files are stored, building tables of information containing counts of all messages sent between the servers. These tables are checked by the program for consistency to detect anomalous behavior, such as lost messages or excessive numbers of aborts. Finally, one line is printed for each experiment. This line summarizes the performance characteristics of the system as a whole and the interesting statistics for each of the RAID subsystems.

All I/O activity during an experiment is directed to the local disk. Such activity consists primarily of database accesses and updates, and writes to the transaction log. Pilot experiments in which some data were directed to a shared file server were successful when only about half the machines (4-5) were involved in experiments, but had poor confidence intervals when more machines were active. Each server also keeps a log of statistics which is written during system startup and system termination. Server logs do not impact transaction processing performance and therefore are directed to the file server.

On the Sun 3/50s care was taken to make sure that the screen was blanked when the experiments were run. The video monitor uses the same bus as the CPU to access memory, resulting in approximately a 25% slowdown when the screen is not blanked.

[6] Cron is a Unix service that arranges for processes to be invoked at specified times.

[7] A shell script is a program in the Unix command interpreter's (the shell's) language.

- Try to start an oracle on the designated oracle site. If there already is an oracle on that site, the new oracle will find it and terminate during startup.
- Unmount all remote-mounted file systems. This step reduces the possibility of remote machine failures affecting the experiments.
- Mount the two file systems needed for the experiments.
- Find the benchmark directory for this machine on this day. If it is empty or nonexistent, terminate.
- Look for an initialization file (name "Initialize") in the benchmark directory. If there is such a file, run it. These files initialize the database, usually by running `dbreset` to clear all relations and then loading freshly generated tuples.
- For each benchmark script in the benchmark directory (recognized as a file name with a ".bm" suffix):
  - For each line in the file:
    - invoke the DebitCredit script with the specified arguments.
- Re-mount the remote file systems.

Figure 4: Start Experiment Script Logic

- Generate a transaction benchmark according to the parameters of the experiment, using the `transaction` program.
- Write a command file for the AD simulator that sets its parameters (arrival gap, timeout, maximum concurrency, restart backoff method, and open/closed experiment type).
- Invoke the RAID script, handing it additional parameters and the name of the command file for the AD simulator.

Figure 5: DebitCredit Script Logic

- Generate a short transaction benchmark according to the parameters of the experiment, using the `transaction` program.

- Prepend a control transaction to convert to the initial concurency control method to the transaction benchmark file.

- Append a control transaction to convert to the final concurrency control method to the transaction benchmark file.

- Append another short transaction benchmark to the end of the transaction benchmark file to keep the load steady while conversion is occuring.

- Write a command file for the AD simulator that sets its parameters.

- Invoke the RAID script, handing it additional parameters and the name of the command file for the AD simulator.

Figure 6: Converting DebitCredit Script Logic

- Choose a unique RAID instance number for this experiment.

- Clean oracle entries for the chosen RAID instance.

- Start the RAID instance.

- Busy-wait until two consecutive oracle list commands report the same number of registered servers. (An alternative would be to check the configuration file for the database to find out how many sites are involved.)

- Check the server log files to make sure they all intialized correctly.

- Run the AD simulator using the specified benchmark transactions.

- Terminate the RAID instance, and clean up the oracle.

- Move the server log files to an archive directory, timestamping them with the date and time of the experiment.

Figure 7: RAID Script Logic

We scanned the system log files to learn about automatic system activity that might disturb the experiments. The experiments were scheduled to avoid nightly and weekly distribution of software to the workstations, and to be after the nightly file system backups.

Each experiment involved running 250 transactions on the system. 250 was chosen as a reasonable number that yielded approximately steady-state measurements despite the start-up and tail-off times. There was little difference between running 250 transactions and running 300 transactions. We did not run higher numbers of transactions to economize on computing resources.

Experiments were run on the extended DebitCredit benchmark using five independent variables:

- transaction length: the number of actions in a transaction.
- fraction updates: the fraction of the actions that are writes.
- inter-arrival gap: the delay between the arrival of one transaction and the arrival of the next for open experiments.
- multi-programming level: the number of transactions running at a given time.
- hot-spot access fraction: the fraction of accesses that access the hot-spot.
- hot-spot size fraction: the fraction of the database that comprises the hot-spot.

For each of these independent variables, we measured the performance of the system in terms of the dependent variable throughput, expressed in transactions per second.

Unless otherwise indicated, experiments were run on a system that used a timestamp order concurrency controller, a read-one-write-all replication controller, and a two-phase commit protocol. All experiments are "closed" with a fixed degree of multiprogramming. The degree of multiprogramming was set to a small number (3) to minimize serialization conflicts. Aborted transactions were restarted after a delay that was computed using an exponential random distribution with the rolling average of transaction response time as the mean. In each experiment the workload was provided by a single AD running on one of the sites. Multiple workload experiments would also be interesting, but are more difficult to synchronize and parameterize.

3 Experiments in Adaptability

This section presents the details of our experiments in distributed systems adaptability. We describe four experiments on adaptability. The first experiment measures the overhead of an adaptable implementation in the concurrency controller and replication controller subsystems of RAID. The second experiment identifies a set of conditions under which concurrency control adaptation should be performed. The third experiment measures the cost of replication and atomicity control methods that increase availability. The fourth experiment explores the performance impact of the algorithm employed by the replication controller on the atomicity controller. The choice of servers for a given experiment was based on the current infrastructure available in RAID and the potential to obtain meaningful data.

3.1 Experiment I: Cost of Adaptable Implementation

3.1.1 Statement of The Problem

An adaptable design is more complicated than a non-adaptable design. This experiment measures the difference in performance between the adaptable and the non-adaptable implementations.

3.1.2 Procedure

We conducted this experiment on the CC and the RC subsystems of RAID. Of the five concurrency controllers implemented in the RAID CC, two are implementations of the timestamp order (T/O) policy. These implementations enforce the same concurrency control policy, but one uses a generic data structure specifically designed for adaptability, while the other uses an ad hoc data structure designed specifically for T/O.

Similarly, the replication controller can use a non-adaptable Read-One-Write-All (ROWA) policy, or a Read-One-Write-All policy based on quorum consensus (QC-ROWA).

To compare the cost of using adaptable implementations, we measured the non-adaptable T/O algorithm against the adaptable T/O method varying the size of the database hot-spot. The experiment was run on a single-site database with a small hot-spot to produce a significant degree of conflict.

For replication control, we compared the performance of ROWA and QC-ROWA varying the proportions of read and write actions. A four site database was used with the branch and teller relations replicated on three sites each, the account relation replicated on all four sites, and the history relation replicated on two sites. The copies were arranged so that each site contained three copies.

3.1.3 Data

Figure 8 shows the throughput for the two concurrency controllers as the size of the hot spot increases. Figure 9 shows the response time for the two replication control methods as update percent increases. The 90% confidence intervals for both sets of data were less than 10% of the data values. All data shown is from systems running on Sun 3/50s.

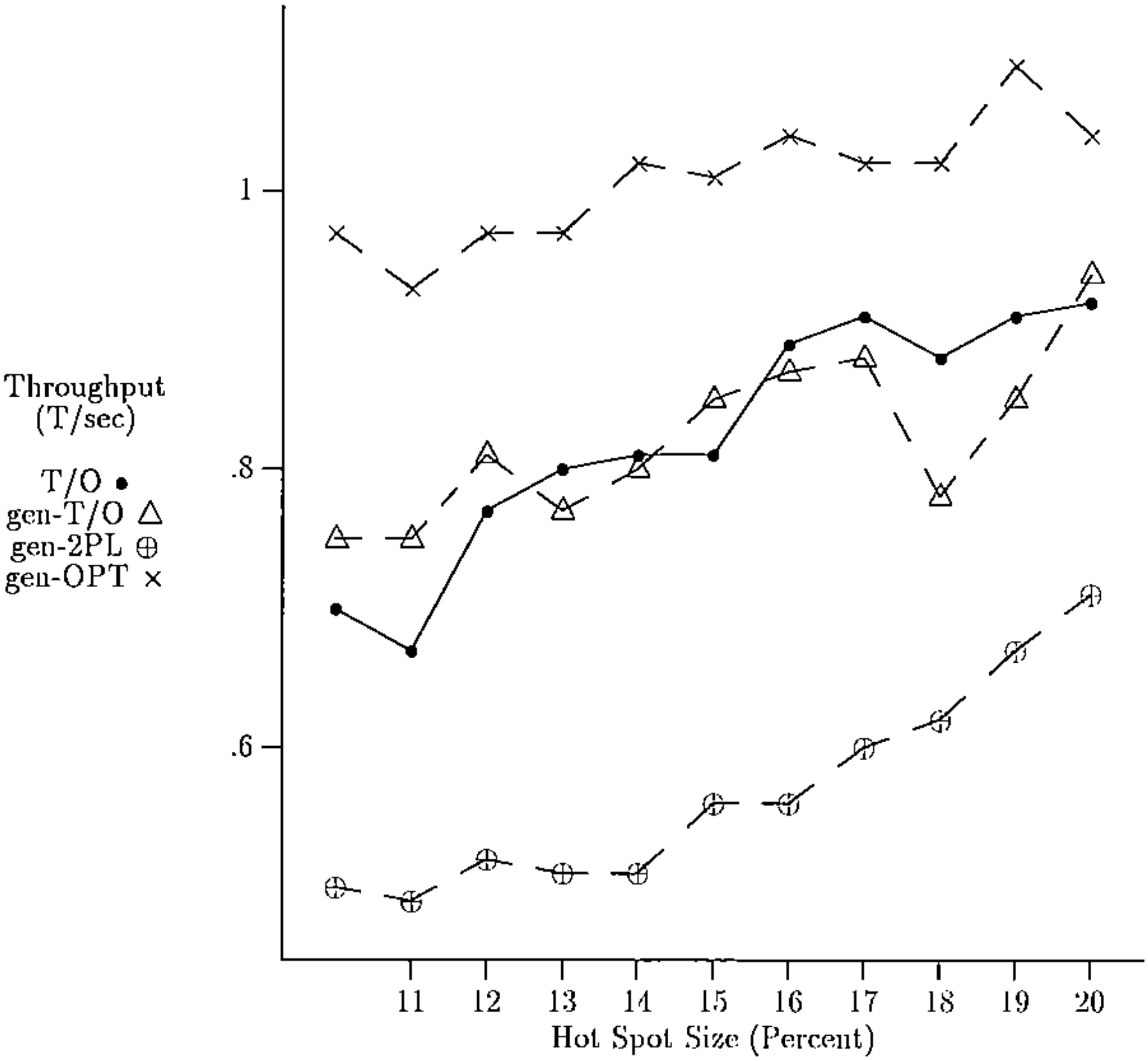

Figure 8: Performance of Adaptable Concurrency Control Implementation

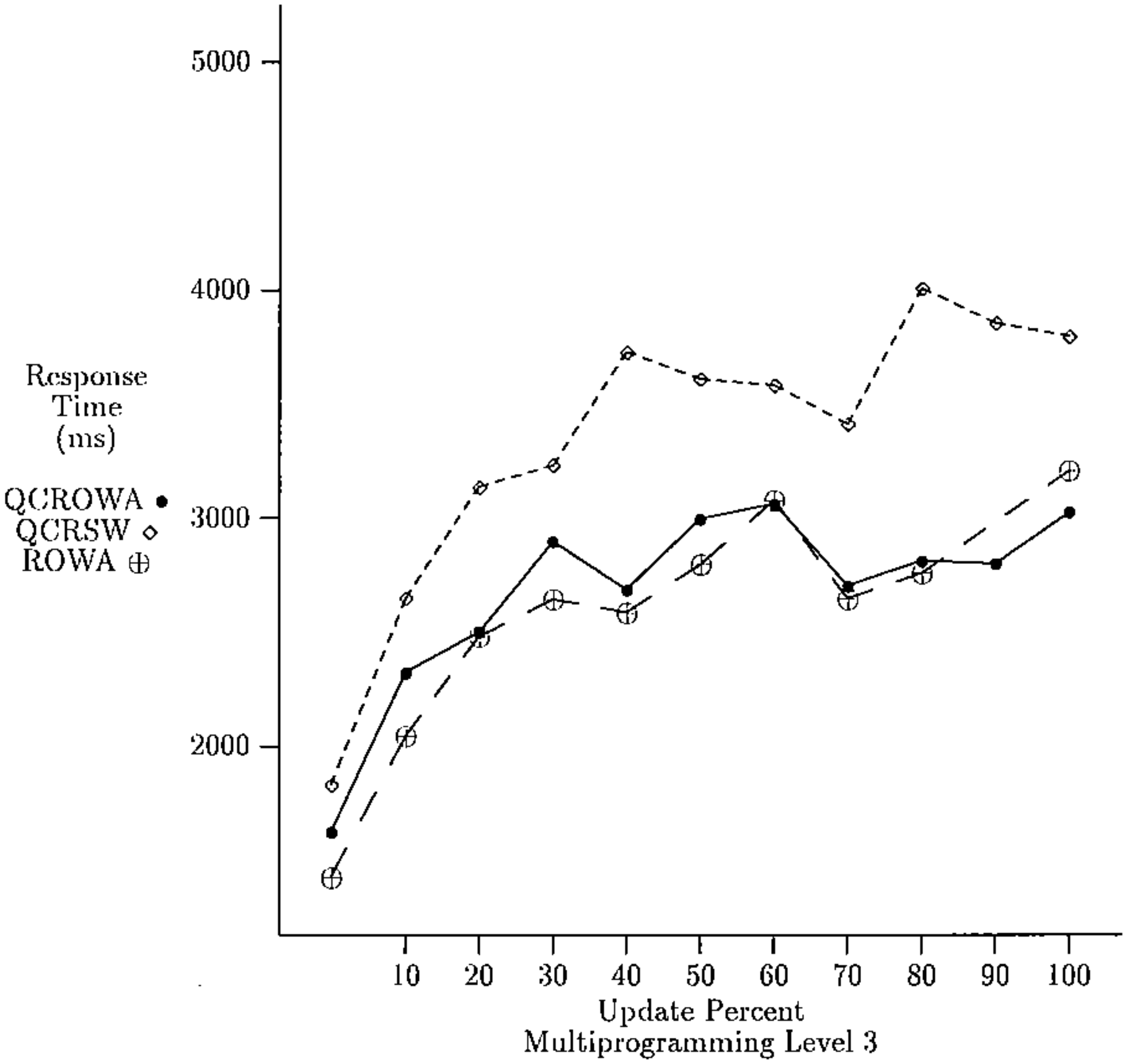

Figure 9: Quorum Consensus Replication Control Performance

3.1.4 Discussion

To the limits of the experiment there were no discernible differences in performance between the adaptable implementations and the specialized implementation. The reason is that the differences in execution time between the two algorithms are small in comparison to the execution time required to process a transaction. The algorithm selected has much more impact on performance than different implementations and execution speed of the same algorithm.

For the replication control data, we compare the response time for the two replication methods, since there was little difference in the throughput. Once again, we observe that the cost of the adaptable method is not significantly different from the cost of the non-adaptable method. In the case of concurrency control and replication control, a carefully designed adaptable implementation can perform as well as a non-adaptable implementation.

3.2 Experiment II: Cost and Benefit of Dynamic Adaptability

3.2.1 Statement of The Problem

Dynamic adaptability allows the operator of a system to change from one algorithm to another while the system is running. This experiment examines the cost of adaptation to determine the effectiveness of particular dynamic adaptations.

3.2.2 Procedure

This experiment was performed on the CC server. The item-based generic state described in [BR89a] was used to implement three concurrency controllers: generic 2PL, generic T/O, and generic OPT. Then four conversion routines were written to dynamically convert from generic 2PL to and from each of generic T/O and generic OPT, while preserving correctness. In order to preserve correctness aborts are sometimes necessary. A special benchmark was set up to test these conversion routines. This benchmark first ran a control transaction to convert to the initial concurrency controller, then ran 50 transactions to get the system to steady state, and then ran another control transaction to convert to the final concurrency controller. Finally 20 more transactions were run to ensure that the second control transaction ran under normal conditions. The multiprogramming level was set to 20 for these experiments to increase the number of transactions to be checked for abort. The number of aborts required during adaptation are reported to represent the cost of dynamic adaptation.

Measuring only aborts excludes the computation cost of the actual conversion from one method to another. In all methods except 2PL to OPT (which has no conversion cost) this cost is proportional to the number of elements of the read sets of active transactions. In RAID this cost is a small fraction of transaction processing time.

The benefit, on the other hand, is sustained over time, and is in the form of increased throughput from running a better algorithm for the current transaction mix. A measure of the net gain for dynamic adaptability is the amount of time required to make up for the cost, assuming the transaction characteristics remain the same. Thus, we propose that the *expected break-even time* be defined by

$$t = \frac{\text{aborts during conversion}}{\text{abort rate}_{\text{old}} - \text{abort rate}_{\text{new}}}$$

The numerator in this expression expresses the cost of the conversion in aborts. The denominator has units aborts/second, and expresses the benefit of running the new algorithm. The units of the result are seconds, and it expresses the amount of time the system must run with the new method and the same transaction processing conditions to recover the cost of conversion. An alternative expression for net cost of abort is to convert the cost of conversion to throughput, and express the benefit in terms of the increased throughput of the system after conversion:

$$t = \frac{\text{lost throughput during conversion}}{\text{throughput}_{\text{new}} - \text{throughput}_{\text{old}}}$$

Here throughput is expressed in transactions per second, and lost throughput during conversion is computed by estimating the throughput cost of the aborts. One such estimate is to subtract the number of *whole transaction equivalents* that were aborted. For instance, one transaction $\frac{1}{3}$ complete and one transaction $\frac{2}{3}$ complete would be combined to make one transaction equivalent.

In general, the former measure is easier to compute since the number of aborts during conversion is readily available, but the latter measure is a more accurate measure of the actual estimate of the conversion cost, especially if a good estimate of the number of whole transaction equivalents is available.

We measured the number of aborts required for each type of conversion over a range benchmarks under a high degree of multiprogramming. We measure the throughput of the old and new methods for each type of conversion. We compute the net cost of the conversion, in seconds.

3.2.3 Data

The number of aborts required for dynamic conversion under a multi-programming level of 20, for a range of hot spot sizes are shown in figure 10. The relative confidence intervals for this data are very large (in some cases as great as 100% of the data value), so the data should not be interpreted as good indicators of the number of aborts needed for conversion for each hot-spot size. However, in no case was more than three aborts needed during conversion, and for every hot-spot size the average number of aborts was no greater than two. Note

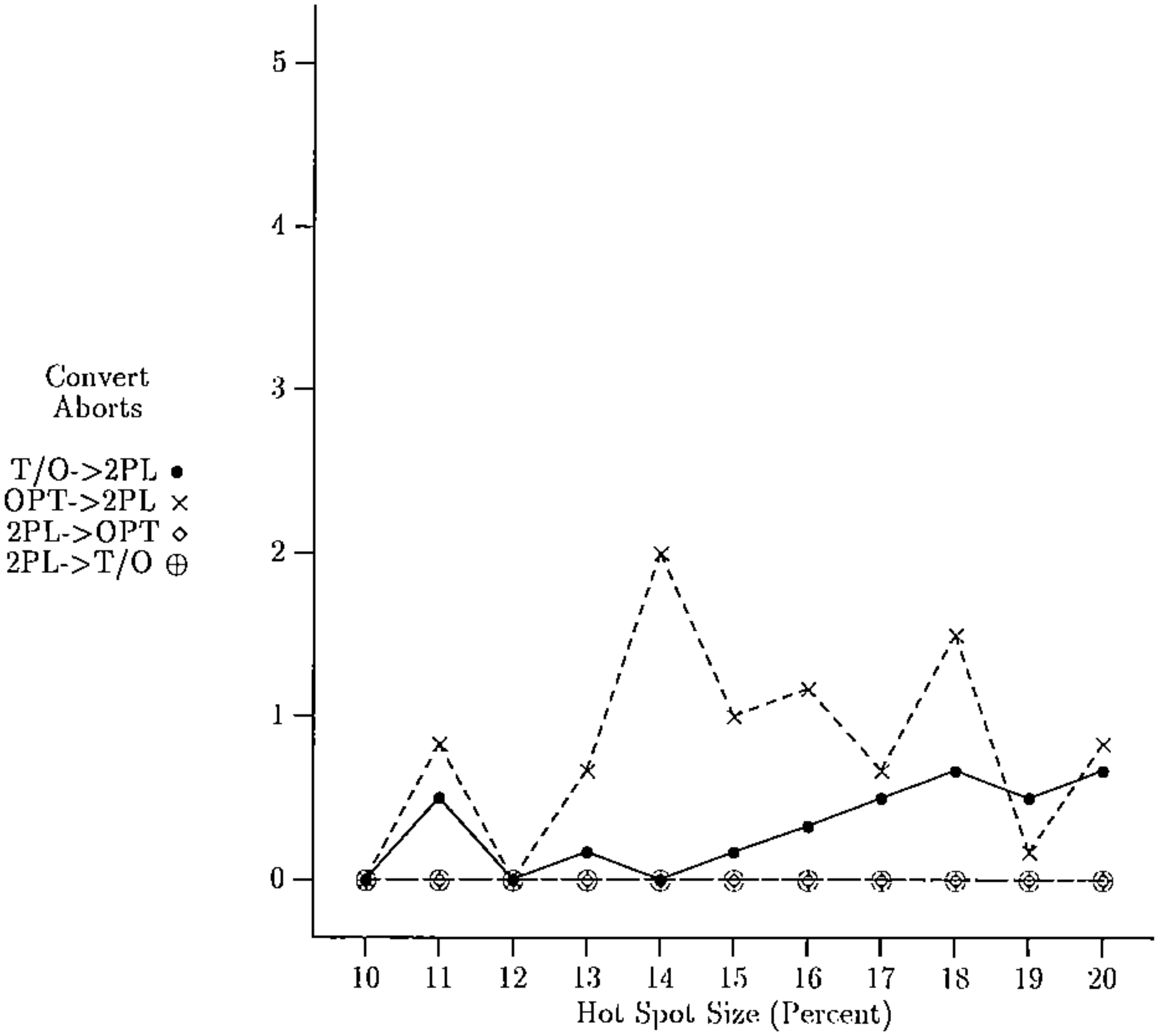

Figure 10: Concurrency Control Aborts during Dynamic Adaptability

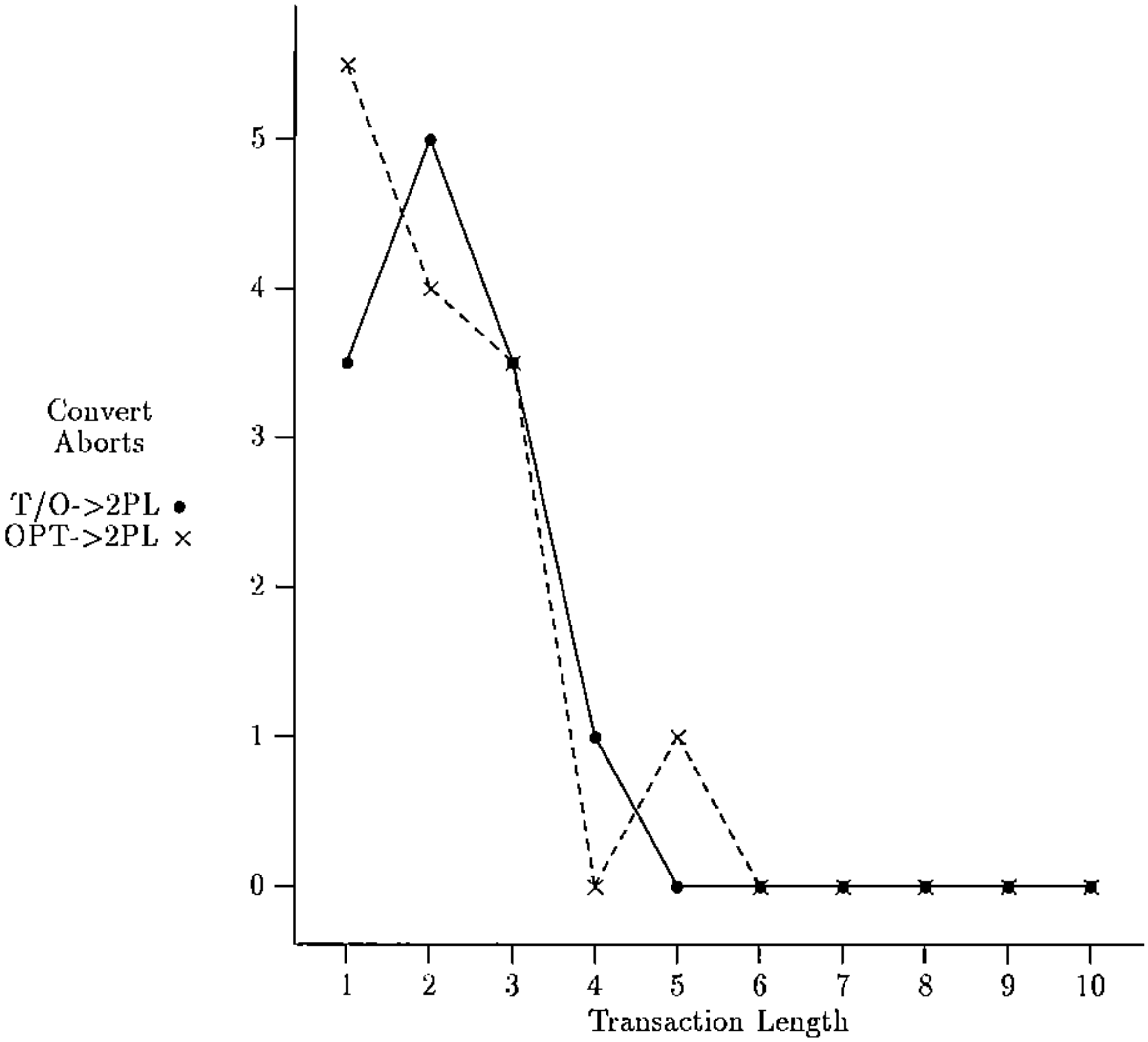

Figure 11: Concurrency Control Aborts during Dynamic Adaptability, Two Sites

that converting from 2PL to OPT and 2PL to T/O never requires aborting a transaction. The 2PL to OPT case never requires an abort because OPT uses the same rules as 2PL, but applies them at the end of the transaction. The 2PL to T/O case never requires an abort in theory because committed transactions can be implicitly renumbered to have earlier timestamps than all uncommitted transactions, thus avoiding all out-of-order timestamp conflicts. Figure 11 shows similar data for a two-site RAID instance, with transaction length as the independent variable. All the concurrency controllers were converted to the new method in a single control transaction. Again the data is not consistent, but the maximum number of aborts was eight, and the average is never much more than five aborts.

Figure 12 shows the net cost of dynamic conversion from generic 2PL to generic OPT, assuming that two transactions are aborted during conversion and that the multiprogramming level after conversion is 10. We use the throughput-based method of computing net conversion cost, and assume that each aborted transaction was halfway completed.

3.2.4 Discussion

In order to invoke dynamic adaptation, a system manager must have some measure of the benefit of adaptability versus the cost. In the case of the generic state adaptability used in the previous experiment the cost is instantaneous, and is measured in number of aborted transactions. A system manager reading figure 12 would look up the current hot-spot size and read the number of seconds the system would have to stay at this hot-spot size before the cost of converting to OPT would be regained. In this case he/she would be very likely to perform the conversion, since less than one second is required to make up the lost transactions.

In all cases the cost of conversion was relatively low in number of aborts. This is to be expected because only active transactions are candidates for abort using the generic state conversion method, so even at a degree of multiprogramming of 20 there are at most 20 transactions that could be aborted. Note that the number of aborts required for generic state adaptability is exactly the same as the number that would be required for converting state adaptability. Converting state adaptability would have a higher computation cost, but the difference is unlikely to be as high as even one transaction execution time. Estimating one extra lost transaction of cost during conversion still yields a net cost of less than two seconds for conversion. Suffix-sufficient state is harder to estimate, since conversion is not instantaneous. However, the total cost of the conversion in that case is reduced throughput for one transaction length period. Except for systems with very long transactions, or for conversions between two very different concurrency controllers the cost of the reduced throughput is likely to be less than the cost of the aborted transactions under generic state adaptability.

Dynamic adaptability of concurrency control is inexpensive. Even under a heavy load the cost can be amortized in less than one second if the new algorithm is a significant improvement over the old algorithm.

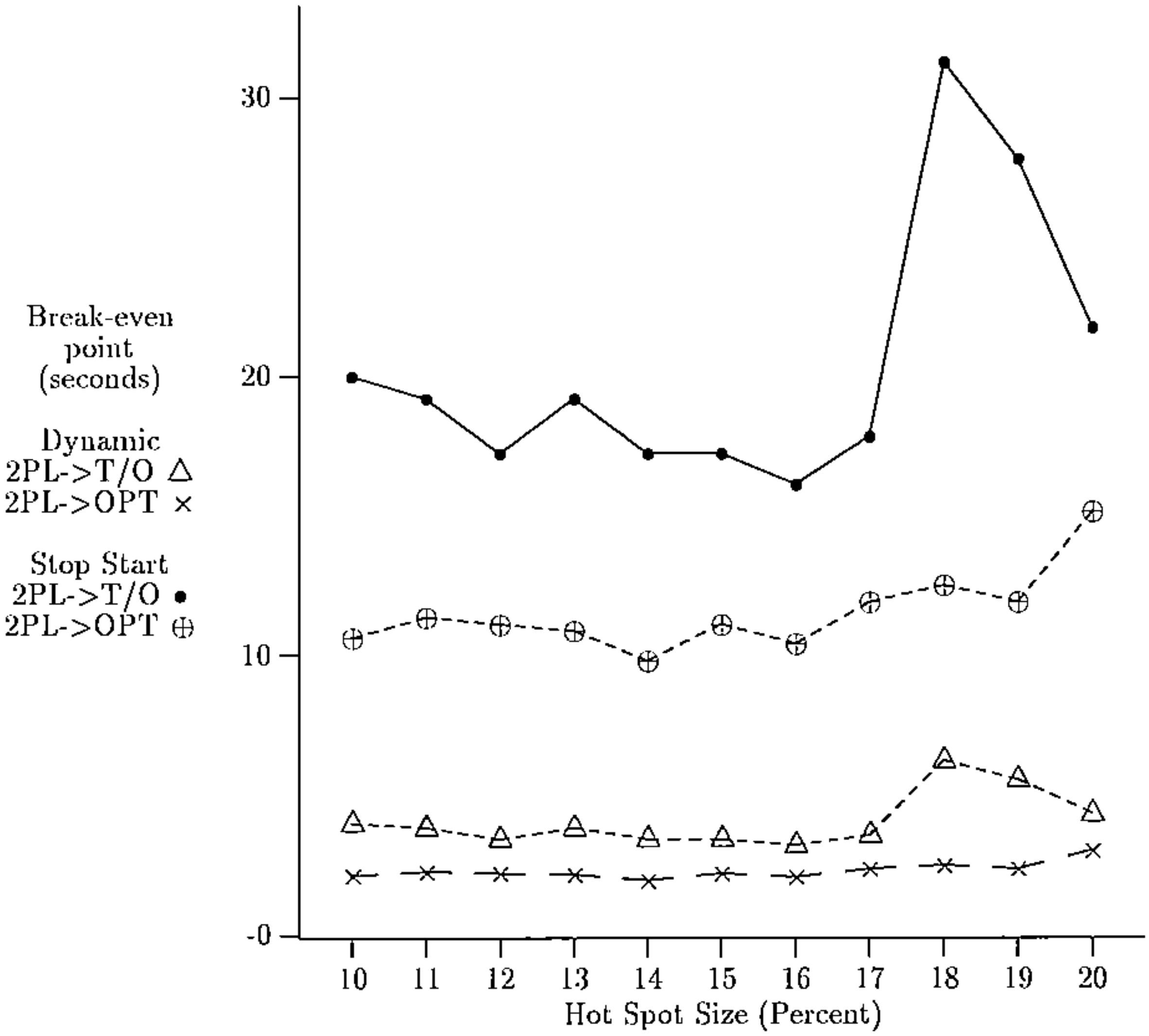

Figure 12: Net Cost of Dynamic Concurrency Control Adaptability

3.3 Experiment III: Cost Attributable to Increased Availability

3.3.1 Statement of The Problem

Many replication control methods and commit protocols provide increased availability at the cost of performance. This experiment compares the replication method read-one-write-all (ROWA) against read-same-as-write (RSW) method. We also examine the performance of two-phase commit (2PC) and three-phase commit (3PC).

3.3.2 Procedure

For replication control, we compared the performance of QC-ROWA and QC-RSW, varying the proportions of read and write actions. A four site database was used with the branch and teller relations replicated on three sites each, the account relation replicated on all four sites, and the history relation replicated on two sites. The copies were arranged so that each site contained three copies.

To explore the cost of 2PC versus 3PC, a five site, fully replicated instance of the standard DebitCredit database was used. In the first data set, the proportion of updates in the transaction stream was varied. In the second data set, the average transaction length was the independent variable. Since the number of serialization conflicts increase with transaction length, the size of the database hot-spot was increased to 40% to minimize such conflicts.

3.3.3 Data

Figure 9 compares the QC-ROWA and QC-RSW protocols. 90% confidence intervals for both experiments were less than 10% of the data values.

Figure 13 shows the throughput for 2PC and 3PC on a five-site RAID system, running on Sun 3/50s. Figure 14 shows the average commit time for the series of experiments. 90% confidence intervals for both figures are less than 10% of the data values.

Figures 15 and 16 show the throughput and commit times for the same five site RAID system with length as the independent variable.

3.3.4 Discussion

The QC-ROWA replication method consistently performed better than the QC-RSW method in some cases as much as 25% better. In the cases where the percentage of update actions is low, the differences between the two methods are not as significant. Most of the cost of QC-RSW is incurred by the additional reads that it must perform to ensure an increase in data availability.

The difference in throughput for the AC was much less pronounced. Although 3PC consistently measured slightly higher, the difference between individual points is not statistically significant. The difference in commit time, reflecting the AC cost alone, is more pronounced,

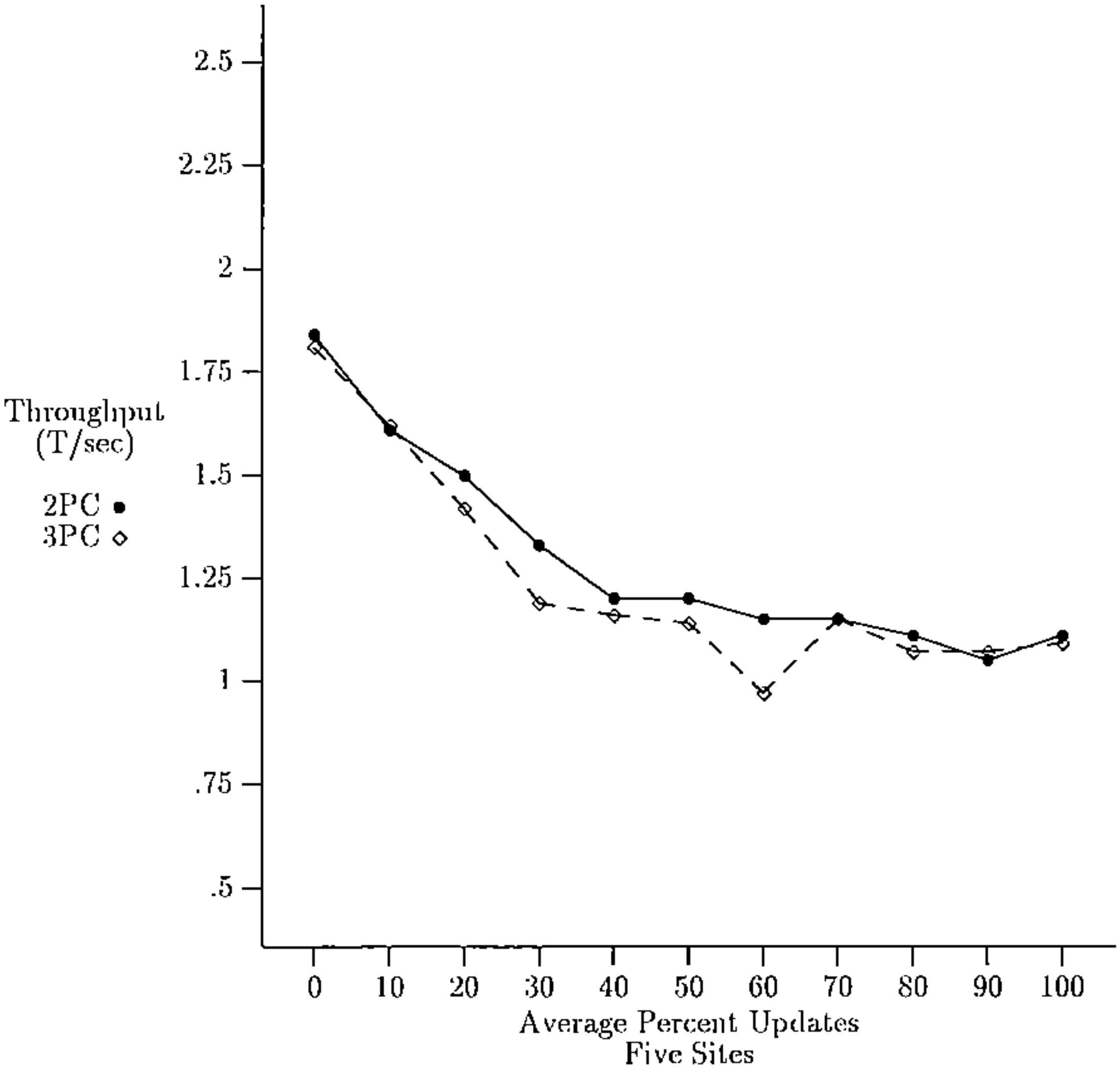

Figure 13: Atomicity Control Throughput for 5-Site RAID, on Sun 3/50s

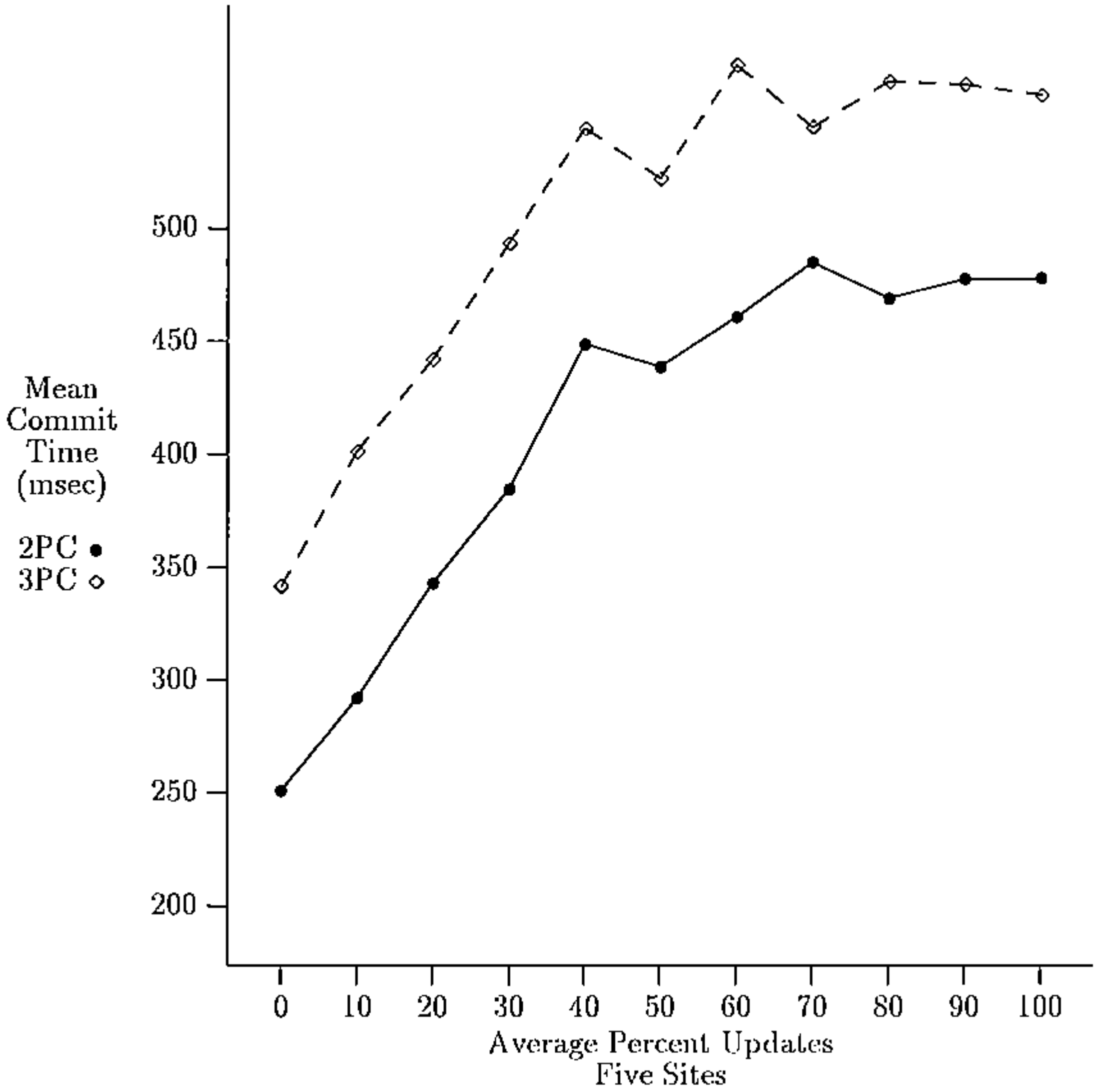

Figure 14: Atomicity Control Commit Time for 5-Site RAID, on Sun 3/50s

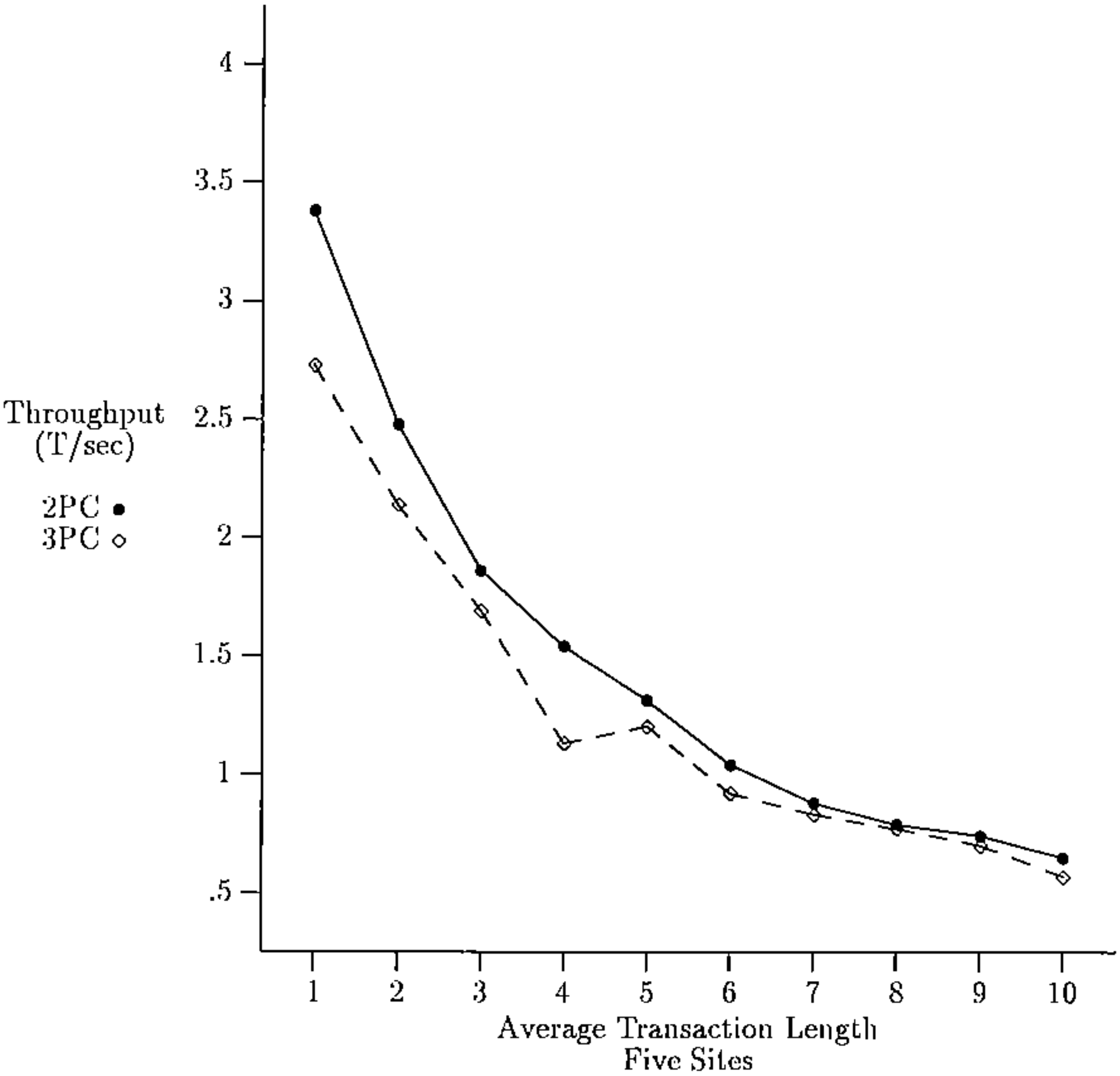

Figure 15: Atomicity Control Throughput for 5-Site RAID, on Sun 3/50s

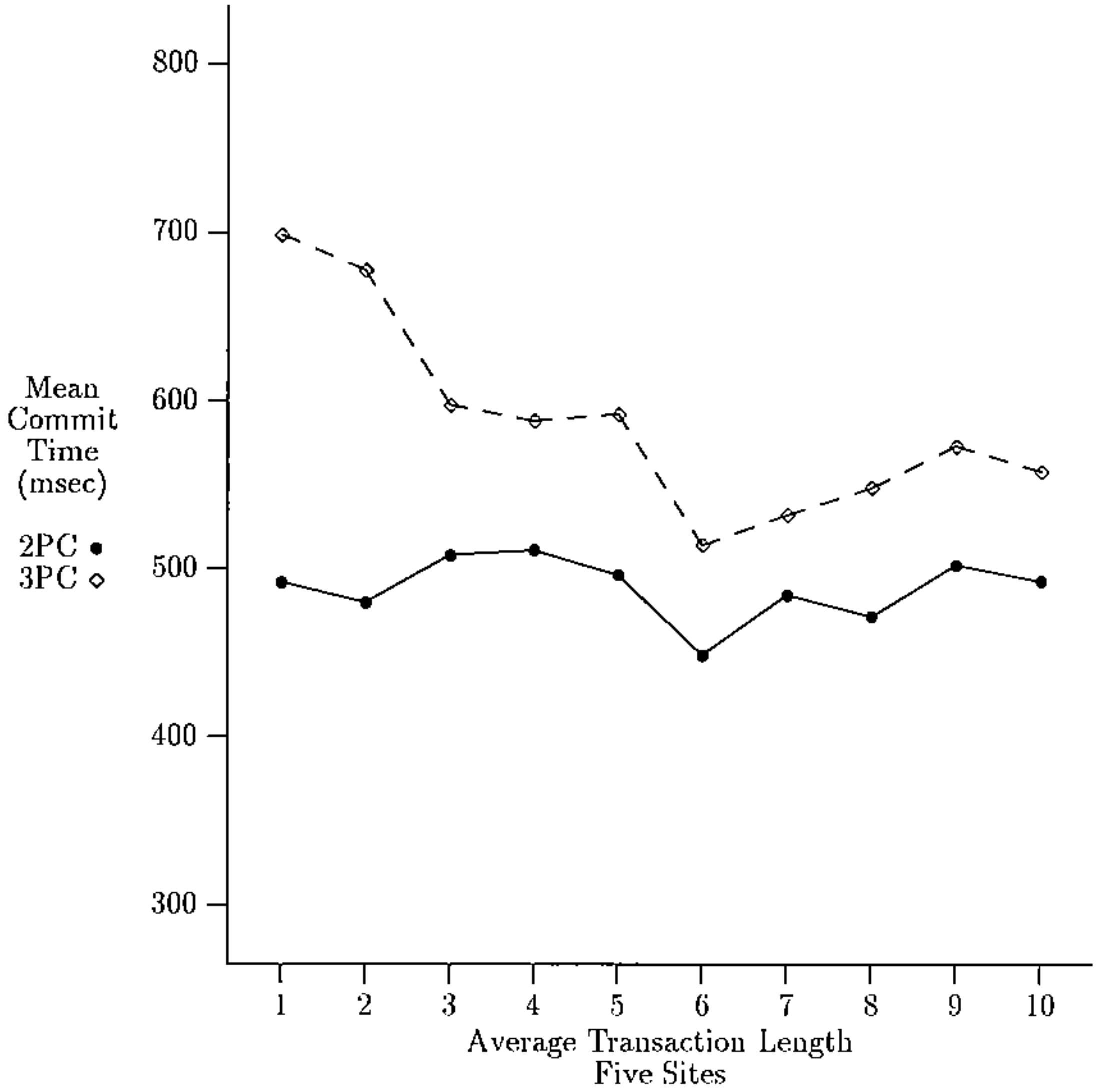

Figure 16: Atomicity Control Commit Time for 5-Site RAID, on Sun 3/50s

with 3PC requiring 30% to 50% more time than 2PC. The lack of effect on throughput of the commit protocol reflects the relatively low contribution of distributed commitment to the total transaction processing time. Systems that have lower transaction processing costs than RAID will see a more significant effect on total throughput of running the 3PC protocol.

In summary, there is some advantage that could be accrued by a system that can utilize more efficient methods (such as ROWA or 2PC) during normal transaction processing and can still provide increased availability when it is needed. Systems with relatively heavy-weight transactions, like RAID, can use 3PC all the time for increased availability at little cost. Systems with lighter-weight transactions may wish to build an adaptable commit protocol that can change between 2PC and 3PC as needed.

3.4 Experiment IV: Effect of Replication Algorithms on Commit Performance

3.4.1 Statement of The Problem

In systems that may employ a variety of algorithms, there may be configurations where the algorithms used by one subsystem may affect the performance of another subsystem. This experiment measures the effect of replication algorithms on the time required to commit a transaction.

3.4.2 Procedure

We ran ROWA and QC-RAWO (read-all-write-one) on a four site, fully replicated database, varying the fraction of actions that were updates. A two-phase commit protocol was used by the AC, and generic locking was used by the CC. For both the ROWA and the QC-RAWO cases, we measured the average commit time spent by the AC server.

3.4.3 Data

Figure 17 shows the average commit time for the two replication control methods.

3.4.4 Discussion

For transactions where read actions dominate (60%-100% reads), the ROWA protocol requires less time to commit than the QC-RAWO. For transactions that are predominantly write actions (50%-100% writes), the converse is true. Both protocols require the same time to commit transactions when updates comprise 40% of the transaction stream.

These results suggest that for a transaction stream that consists of many write actions, QC-RAWO is a better replication protocol to use. RAWO will allow writes even in the

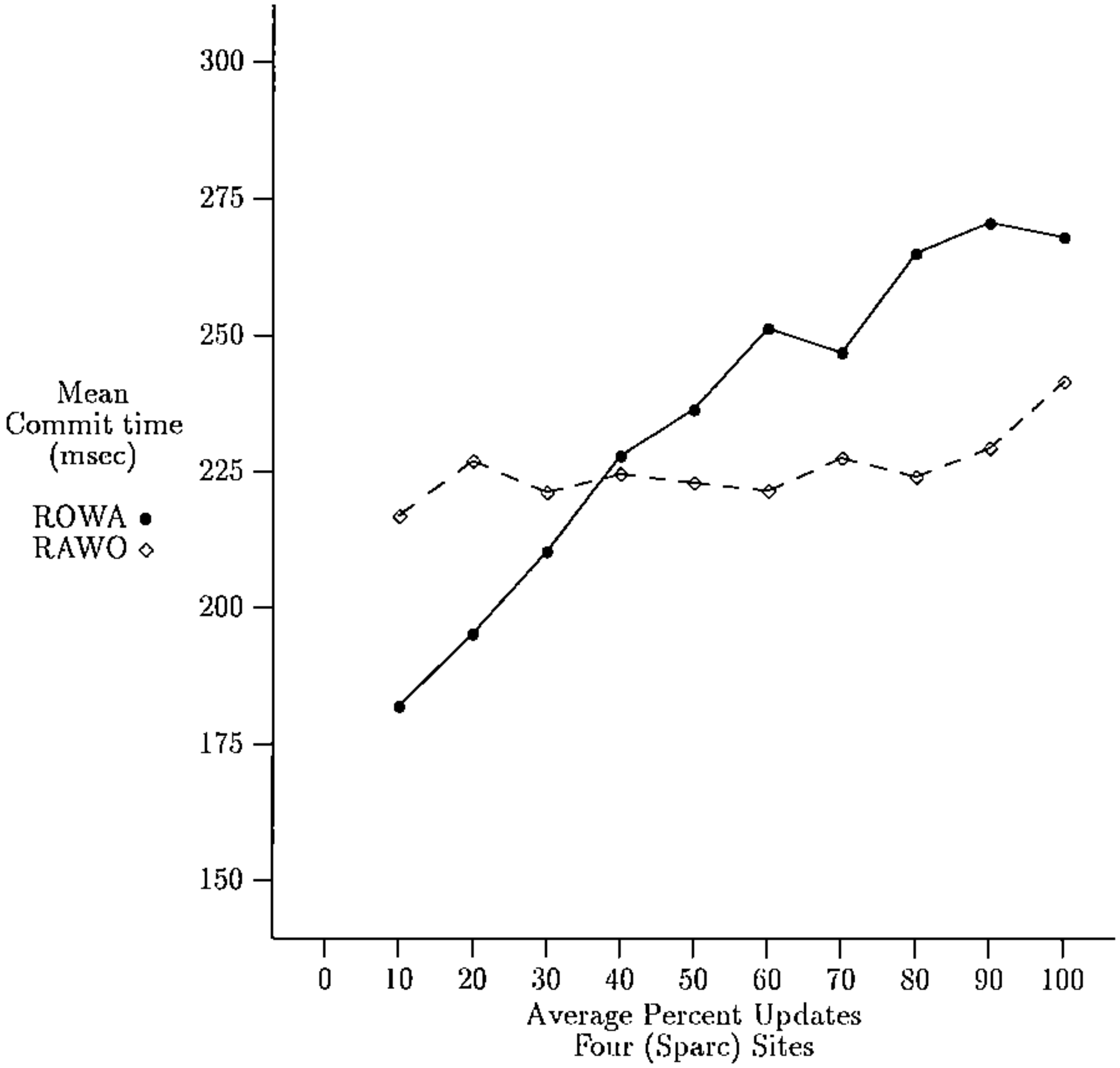

Figure 17: Effect of RC method on commit performance

presence of site failures and will commit in less time. Read actions would still block, but this may not be significant if reads are sparse in the transaction stream.

4 Conclusion

These experiments show that for certain types of transaction systems adaptability can be a useful tool for improving performance and availability. Furthermore these experiments establish a method for determining a spectrum of values of the five independent variables (transaction length, percent updates, inter-arrival gap, hot-spot, and transaction granularity) for which adaptability is beneficial in a distributed transaction processing system. This method is based on measuring the cost of adaptation in terms of the amount of time required before the decrease in throughput during conversion will be made up by the increased throughput after conversion. An adaptation is considered beneficial if the system is likely to maintain approximately the same workload characteristics at least as long as required to reach the break-even point.

Experiment I shows that the overhead of a careful adaptable design can be kept low, with respect to the overhead of a non-adaptable design. Therefore, it is possible to build systems that have efficient performance characteristics while providing services that are well suited to the current operating conditions.

Experiment II shows that dynamic switching among algorithms while processing transactions can be done efficiently. In the concurrency control subsystem of RAID we are able to switch from one concurrency controller to another at a net gain in performance after just a few seconds of running with the new concurrency controller. The crucial problems in choosing whether to switch from one algorithm to another are the relative performance of the two algorithms under existing conditions, the amount of time that the conditions will remain the same, and the cost of the conversion. Experiment III demonstrates the effectiveness of adaptability techniques in responding to changing environmental conditions.

Experiment III shows that running algorithms that can increase availability can be expensive. In a system like RAID, the cost in throughput of running 3PC is not significantly greater than that of running 2PC. On the other hand, in a system with light-weight transactions, the cost may be as high as 25% greater to run 3PC. Further, the cost of replication control algorithms that increase availability may result in as much as a 50% decrease in throughput. However, in many applications the benefits of increased availability outweigh reduced performance. These facts suggest that adaptable implementations that can change between high-performance methods and high-availability methods can serve to increase system availability at a relatively low cost in performance.

Experiment IV demonstrates that in systems that may employ a variety of algorithms, there may be configurations where the algorithms used by one system component may affect the performance of another. We measured the effect of replication algorithms on the time required to commit a transaction. The results in this experiment suggest that for a transaction stream with large ratio of updates, QC-RAWO is a better replication protocol to use.

An extension of these results would be to automate the decision to adapt to a new algorithm through the use of an expert system. The expert system could use the results of experiments like these to determine the approximate costs and benefits of dynamic adaptation in a given situation.

References

[A+85] Anon et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, April 1985.

[AKW] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *Awk - a pattern scanning and processing language.* The UNIX Programming Manual.

[BDT83] D. Bitton, D.J. DeWitt, and C. Turbyfil. Benchmarking database systems: a systematic approach. In *Proceedings of the Ninth International Conference on Very Large Databases*, October 1983.

[BFH+90] Bharat Bhargava, Karl Friesen, Abdelsalam Helal, Srinivasan Jagannathan, and John Riedl. Design and implementation of the RAID V2 distributed database system. Technical Report CSD-TR-962, Purdue University, 1990. CSD-TR-962.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BMR87] Bharat Bhargava, Tom Mueller, and John Riedl. Experimental analysis of layered Ethernet software. In *Proceedings of the ACM-IEEE Computer Society 1987 Fall Joint Computer Conference*, pages 559–568, Dallas, Texas, October 1987.

[BMR89] Bharat Bhargava, Enrique Mafla, and John Riedl. Experimental facility for kernel extensions to support distributed database systems. Technical Report CSD–TR–930, Purdue University, April 1989.

[BMR91] Bharat Bhargava, Enrique Mafla, and John Riedl. Communication in the Raid distributed database system. In *Journal of Computer Networks and ISDN Systems*, volume 21, pages 81–92, 1991.

[BNS88] Bharat Bhargava, Paul Noll, and Donna Sabo. An experimental analysis of replicated copy control during site failure and recovery. In *Proceedings of the 4th IEEE Data Engineering Conference*, pages 82–91, Los Angeles, CA, February 1988.

[BR89a] Bharat Bhargava and John Riedl. A model for adaptable systems for transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):433–449, December 1989.

[BR89b] Bharat Bhargava and John Riedl. The RAID distributed database system. *IEEE Transactions on Software Engineering*, 16(6):726–736, June 1989.

[Che88] D. R. Cheriton. The v distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Com84] Douglas Comer. *Operating System Design: The Xinu Approach.* Prentice-Hall, Inc., 1984.

[CS84] Michael Carey and Michael Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proceedings of the Tenth International Conference on Very Large Data Bases*, Singapore, August 1984.

[HSB89] A. Helal, J. Srinivasan, and B. Bhargava. SETH: a quorum-based replicated database system for experimentation with failures. In *Proceedings of the 5th IEEE Data Engineering Conference*, pages 200–207, Los Angeles, CA, February 1989.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *ACM Computing Surveys*, 1(1):11–32, 1988.

[PW85] Gerald J. Popek and Bruce J. Walker. *The LOCUS Distributed System Architecture.* The MIT Press, 1985.

[S+86] Alfred Z. Spector et al. The Camelot project. *Database Engineering*, 9(4), December 1986.

[TWPWP85] Jr. Thomas W. Page, Matthew J. Weinstein, and Gerald J. Popek. Genesis: A distributed database operating system. In *Proc of ACM-SIGMOD 1985 International Conference on Management of Data*, pages 374–387, May 1985.