A Study of Disk Update Policies in a Replicated Server Environment

by

Michael R. Foster

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1996

Author .....
Department of Electrical Engineering and Computer Science
June 20, 1996

Certified by.....
Dr. Liuba Shrira
Research Associate
Thesis Supervisor

Accepted by ....
F. R. Morgenthaler
Chairman, Department Committee on Gradute Theses

A Study of Disk Update Policies in a Replicated Server Environment

by

Michael R. Foster

Submitted to the Department of Electrical Engineering and Computer Science
on June 20, 1996, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As computer technology has progressed, the gap between CPU speed and disk speed has continually increased. As a result of this, large muti-user environments can become disk bound in their operations. Therefore, increasing disk utilization is a high priority. In order to do this, disk update policies have been developed. We will be studying several update policies under a replicated server environment, specifically the Thor database system. We are testing three different update policies. The Read-Modify-Write policy will issue operations in a FIFO manner. The Opportunistic policy will issue writes based on which page is closest to the current disk position. The MBatch policy will use the same algorithm as the Opportunistic, except it will issue the writes in batches. In order to predict how the policies will perform, we developed an analytical model of Thor. To validate our model, we developed a simulator of Thor to test the policies on. We experimented with several different sets of parameters to determine which policy would perform the best under different situations. Our conclusions were that the Opportunistic policy had the highest throughput for the system parameters we used.

Thesis Supervisor: Dr. Liuba Shrira
Title: Research Associate

Acknowledgments

First of all, I would like to give profuse thanks to my thesis advisor, Liuba Shrira. Thank you for giving this sophomore with little experience a chance and for pushing me to achieve more than I thought I ever could over the past three years. You never let me settle for "just getting it done."

I would also like to thank all of my friends and family for their support throughout this thesis and through all my years at MIT. I would especially like to thank my parents, Robert and Regina Foster, for the support they have always given me. Without them, I never would have achieved all that I did. Finally, I would like to give my highest level of thanks to Liz Pau. She was always there for me when I needed to complain, when I needed a hug, or just needed someone to say everything was going to be alright.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As technology moves forward, computer components are becoming faster and faster. Processor speed has increased, while memory access time and network latency have decreased. Disk latency has also been reduced, but at a much slower rate. Physical and mechanical limitations, such as the time required to move the disk arm, are the main reasons for this. As a result, when the workload in a computer system increases, the speed with which it can process transactions will be bounded by the speed of the disk. Figure 1-1, adapted from studies by Katz, Gibson, and Patterson, shows how this performance discrepancy will continue to increase as time progresses. Therefore, improved disk management is necessary to prevent a computer system from being I/O bound [8].

This thesis presents a study of three different disk update policies being simulated in a replicated server environment. Specifically, the environment is the distributed object-oriented database system Thor, developed by the Programming Methodology Group at MIT. To ensure that modified data is stored reliably and that the objects are highly available, Thor stores the database on multiple servers. When a client modifies an object, every server must be informed of that change, so that all servers will present an accurate view of the current database state. The servers initially store these modifications in memory, but they must eventually be propagated to disk. For this purpose, each server uses an update policy to improve its disk utilization. This policy governs how the server issues *disk writes*, used to write modifications to disk,

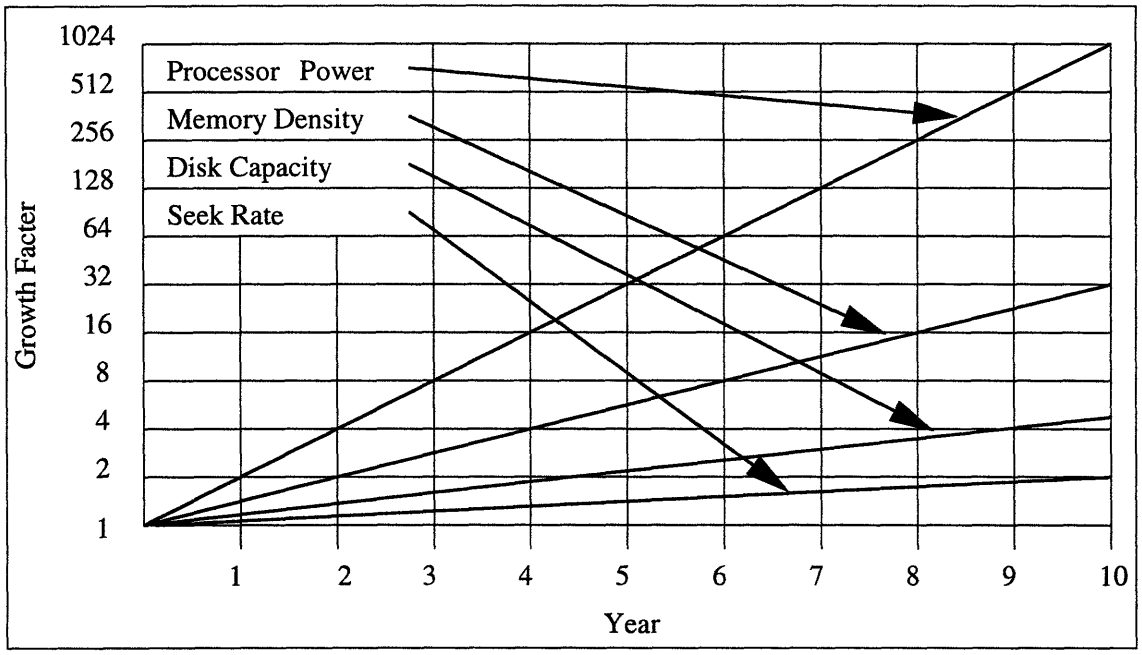

Figure 1-1: The Increasing Gap between CPU and I/O speed

and *installation reads* (ireads), used to read pages that must be modified into the cache.

Disk Update Policies

To improve disk utilization, a server can use a disk scheduling algorithm. The simplest one is a FIFO procedure that writes pages to disk in the order they are modified. Since we are working in a replicated server environment, every modification is stored by each server. Therefore, servers can delay writing a modification to disk, since the only way to loose the modification would be for every server to fail before writing it to disk. One advantage of delaying the write is that another modification might be made to the same page. Both modifications could be serviced by the same write, reducing the total number of writes that the server must perform. Also, since there will be many dirty pages, the server can choose which page to issue based on proximity to the current disk position, greatly reducing the cost of the write.

As mentioned above, we will be testing three different update policies in several

server configurations. All three of the policies will delay the ireads and writes, allowing for absorption to occur. The most basic policy will be the Read-Modify-Write, which will use a FIFO algorithm for issuing the writes. The next two policies schedule both ireads and writes based on disk position. We will experiment to see if this provides the same benefits in a database system as it does in a file system. The Opportunistic policy will delay all ireads and writes until a *trigger value* of them are waiting. Then the iread or write, depending on which trigger has been reached, will be issued for the page closest to the current disk position. Finally, the MBatch policy will use a batching scheme in which to issue the ireads and writes. In [16], issuing writes in batches was shown to improve write performance, but reduce overall read performance of the system.

Thor

The Thor database system provides clients with highly available access to persistent objects. Thor stores these objects in the form of pages. It uses replicated servers for reliability, with one server designated as the primary, the server with which all clients will communicate. The remaining servers act as backup servers for the system. A client sends a transaction to Thor either to fetch an object absent from its cache, or to request a commit. The commit-request can be either a *read-only commit*, in which the client does not modify any objects, or a *write commit*, which does make a modification. Thor responds to fetch-requests as soon as possible. If the page is present in the server cache, then it will be returned immediately. Otherwise, the server must read the page from its disk. All commit-requests are directed to the primary server. Upon receipt of the request, the primary uses a validation algorithm to determine whether or not it should allow the transaction to commit its modifications. After passing validation, the primary propagates any modifications to every replica. Each server keeps a log of these modifications, and eventually, the modified objects are written to disk [10].

In order to predict how these policies will perform under any server configuration, we will present a mathematical model of the Thor database system. By developing the model, we will be able to better understand the effects of the policies.

To verify the mathematical model, we implemented a simulator of Thor. We chose to only simulate two servers, although Thor can have multiple backups. Adding more servers would only complicate the analysis without adding anything to the tests. Clients were simulated as workload generators, which probabilistically selected which pages to use, and whether to make a fetch or a commit request.

We simulated two different server configurations that dictated how fetches and ireads were processed. In the basic server configuration, the primary processes all fetch requests; the backup was updated with modifications only for reliability purposes. Each server performed their own ireads in the configuration. The other configuration divided the pages between the two servers with a cache-splitting algorithm [5]. The fetches and installation reads were split between the two servers, each server handling only those requests for its pages.

In addition to the different configurations, we also tested different parameter values for the system. We tested different ratios of fetch and commit requests in order to see how each policy performed with different amounts of fetch requests, which use a significant portion of the disk time. We also tested different memory sizes, which affected the server cache size. We performed this test to determine how each policy would perform with having an increased number of fetch reads and ireads in the system.

Findings

This thesis will show that scheduling the ireads and writes will have a large effect on the throughput of a system, allowing the database to process transactions at a much higher rate than if they used the Read-Modify-Write policy. Of the two policies which schedule the ireads and writes, we will show that the Opportunistic policy will achieve the highest throughput under each configuration. It will achieve this because of improved fetch response time when compared with the MBatch policy. As shown in [16], batching schemes incur a larger read cost. For our system, the Opportunistic policy has the best fetch response time, which gives it the highest throughput.

We will also observe that under the parameter set we use, the MBatch and Read-

Modify-Write policies will be operating with a full log with a high number of clients. This gives the Opportunistic policy an additional advantage in that its log does not fill during the trials. When the log is full, a client with an incoming modification must wait until log space is available before the transaction will be allowed to commit, lowering the system throughput.

1.1 Related Work

1.1.1 Replicated Servers

Significant research has been done in the field of replicated servers. We are interested in ways in which replication has been exploited for the purpose of improving the overall performance of a system. However, in order to use another server in a database system such as Thor, every replica must present the same, current view of the entire database. Mohan et. al. examined different methods of committing transactions so that the backup server would receive the information as well. They proposed a hybrid of *1-Safe*, where only the primary participates in the commit process, and *2-Safe*, where every replica participates, in which the priority of the transaction dictated which scheme would be used. The 2-Safe scheme is less likely to loose any modifications, but takes longer to perform. In Thor, the 2-Safe algorithm is used as part of the two-phase commit protocol [13].

Molina and Polyzois examined one possible technique for utilizing the backup server: having it process all read-only transactions. In their system, the primary recorded every modification over a certain time interval, while the backup queued every read-only transaction. After that interval, the primary sent the modifications to the backup, so that it could service the read-only transactions. Problems only arose when a modification needed to precede a read-only transaction. Molina and Polyzois proposed several different methods for handling this situation efficiently, each one being appropriate under certain conditions. For I/O bound systems, the separation method, in which all modifications were installed before read-only transactions, was

found to be the best [3].

Another method that can be used in a primary/backup system is cache splitting, which divides the pages between the two servers. Each one will only service fetch-requests to pages for which it is responsible. The page partition is not static, so that if one server became overburdened with fetch-requests, part of the load could be repartitioned to the other server. This essentially doubles the size of the cache that the system has, since each server will only cache the pages partitioned to it. As a result, the cache hit rate improves, so fewer fetches will require a disk read, leading to an improvement in system performance. One other benefit of cache-splitting is that both servers are doing fetches. To isolate the benefits of load-balancing from the benefits of improved cache hit rate, the tests included a system in which the fetch requests were sent to a random server. The cache splitting technique had significantly higher throughput than the basic scheme, in which the primary server processed all fetch-requests, as well as the balanced fetch scheme [5].

In the Harp (Highly Available, Reliable, Persistent) file system, the workload in a multi-server environment is balanced between all servers. As in the cache-splitting technique, the database is divided into two partitions. Unlike the cache-splitting algorithm, this is a static partition. Each server acts as the primary server for one set and the backup server for another set. In a comparison under a UNIX environment against the NFS file system, Harp provided equivalent read performance and improved write performance [11].

1.1.2 Disk Scheduling Algorithms

Since the early 1970's, much research has been done on scheduling disk writes in a file system. The basic disk write policy is FCFS (First Come First Served), where disk-writes are processed in arrival order. Although disk utilization is lower in this policy, it is simple to implement, and provides the same waiting time for all disk-writes. The first improvement upon this was the SSF (Shortest Seek First) policy. In it, the write requiring the shortest seek time is processed next. Disk utilization increases dramatically with this method, but some items are forced to wait much

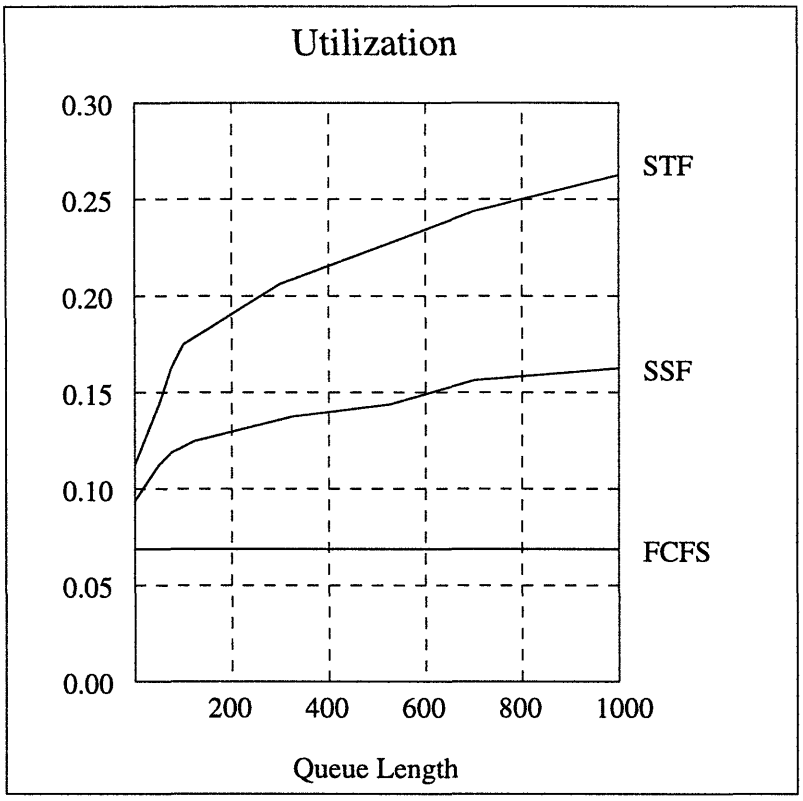

Figure 1-2: Disk utilization of several polices. Adapted from Seltzer et. al.

longer before being written to disk [18]. These results were verified by [2] and [7]. However, these tests were performed in a file system environment, where long waiting times are a problem. In a database system, we are not concerned with how long a page must wait before being written to disk, so we can ignore that shortcoming of the policy.

Seltzer et. al. retested these methods in a more modern environment. The previous tests were from the early 70's, and with the slower machines and smaller amounts of memory, long disk queues could not develop. Seltzer et. al. also introduced the STF (Shortest Time First) policy to compare with the other two. With the STF policy, the next write was the item closest to the current disk position, taking into consideration both seek and rotation time. As figure 1-2 shows, this leads to the highest disk utilization of the three policies [17].

The principle of scheduling writes in a file system can also be applied to scheduling ireads within a database system. O'Toole and Shrira simulated holding the modifica-

tions waiting for ireads in the log, just as dirty pages are held in the cache. The system scheduled ireads opportunisticly, using the STF scheduling algorithm. Because of the scheduling effects, the average time for an iread was reduced. The number of ireads was also reduced, since another modification could be made to the page while it was waiting for an iread. Therefore, both modifications could be serviced by the same iread. Also, if a client requests a fetch for a page waiting in the log for an iread, the server could install the modifications into the page once the fetch read is complete. Shrira and O'Toole's tests showed that their opportunistic log increases throughput by 50% over algorithms which issue ireads immediately [15].

The Unix file system uses a periodic update policy, writing all dirty buffers to disk every 30 seconds. Generally, this provides better overall performance, because it allows write absorption, reducing the total number of writes. However, Carson and Setia showed that although writing a large batch to disk, as in this system, improves write performance, it leads to a higher average read latency, since reads must wait for completion of the writes [16]. As a result of this finding, Mogul proposed an interval periodic update policy. In his system, write blocks were written to disk when their age reached a certain threshold. This provides the advantage of increased write absorption gained by delaying the disk write, and reduces the problem of disk reads blocking behind a write batch, since his system writes smaller batches to disk [12].

1.2 Thesis Organization

The rest of this thesis is arranged as follows. Chapter 2 will describe the Thor database system, the environment under which our tests are being run. It will pay particular attention to the system details that are relevant to the simulation. Chapter 3 will detail the server configurations and update policies that we are testing. In chapter 4, we will give the mathematical model of the replicated server system. Chapter 5 will present the simulator, giving the system parameters that we used. It will also describe the update policies and server configurations in more detail. Chapter 6 will show the simulation results, and will also include the analysis of those results. Chapter 7 will

summarize the thesis, and also present ideas for further areas of research.

Chapter 2

The Thor Database System

The environment that we are working under is the Thor database system. Thor is a client-server object-oriented database system that makes objects highly available to the clients. The high availability is maintained through replication of the servers. One of them is designated as the primary server, while the others act as backups. For simplicity, we will only consider the case where there are two servers, which will not affect the analysis.

Clients can access objects at the servers through transactions, which are either fetch requests or commit requests. To propagate modifications to the replicas, the primary server uses a two-phase commit protocol. Any client can access any object in the system, so Thor handles this with an optimistic concurrency control scheme. Because of this, the servers must use a validation algorithm on commit requests to ensure that the client is not using outdated object values.

2.1 Client-Server Architecture

The servers in Thor provide permanent storage for the objects and update the state of the database after clients make modifications. Instead of directly accessing the servers of Thor, clients use a front-end process to handle the interaction. Through the front-end, the clients can request an object that is not in their cache, or request a commit on an object set.

2.1.1 The Servers

Each server stores the objects on disk in the form of pages. Under Thor, the entire database is stored at each server, ensuring that several replicas of the database exist. One server is designated as the primary, which processes all commit requests from the clients, while the other servers act as backups. With multiple servers, it becomes extremely rare that a client would be unable to access Thor. The only way that Thor could become completely inaccessible would be if every server were simultaneously inoperable. If the primary server fails, then one of the backups would simply assume the duties of the primary. Although there can be several backup servers, we will assume throughout this thesis, without loss of generality, that there is only one backup [10].

Since the backup server may need to take over the database operations from the primary, the primary must inform the backup of every modification made. Therefore, Thor uses a primary copy scheme to propagate all modifications to the backup. After the transaction passes the validation phase, described below, the primary communicates the results to the backup, which stores the modifications and update the objects, just like the primary. Therefore, the backup will present a consistent view of the database at all times and can take over for the primary if needed with no transactions lost [14].

Figure 2-1 provides a basic model of the server architecture. To provide persistent long-term storage for the objects, each server possesses a hard disk. Even in configurations that divide the pages between servers, both servers store the entire database on disk. The memory at each server is divided between a page cache and a log. The cache stores all recently used pages and uses an LRU replacement policy when a new page enters. All modifications are installed into pages while they are in the cache, making them dirty. The transaction log creates an entry for every modification made by the clients. An entry is removed from the log once both servers have written that modification to disk. This way, any server that crashes can be updated through the disk and log of any other server. Each server is connected to an uninterruptible power

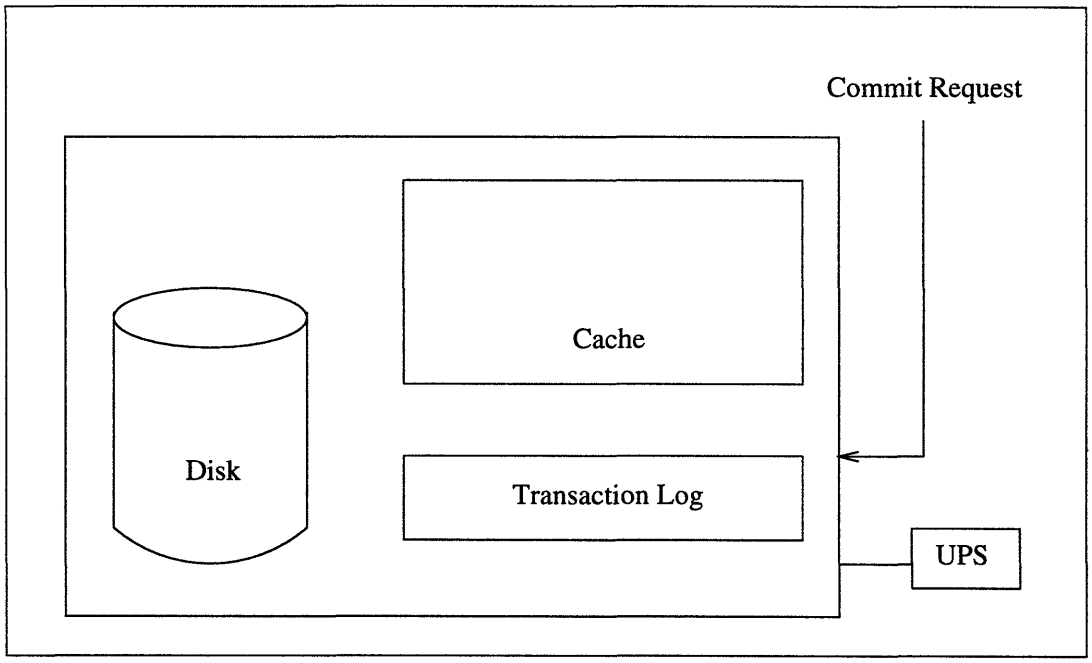

Figure 2-1: Basic Server Architecture

supply, which provides it with sufficient power to write the log to disk in the event of a power failure, preserving all modifications that have not yet been written to the disk [10].

As mentioned above, the primary server processes all commit requests. The primary uses a two-phase commit protocol, which is described below, to communicate any modifications to the backup. The fetch policy determines how fetch requests from the clients are processed. In the basic policy, all fetch requests are processed at the primary server, just as commits are. Thor can also use a dual-fetch policy, which uses the cache-splitting algorithm described in [5]. Each server is given "ownership" of a subset of the pages. A fetch for a page is then directed to the server that owns it. The database partition is not static, so that if one server became overburdened by the fetch load, some of it could be partitioned to the other [10].

2.1.2 Clients

Each client interacts with the Thor servers through transactions, which can be either a fetch request or a commit request. The clients possess an object cache that stores the objects it has been using, just as the server has a page cache. When an object that the client requires is absent from its cache, the client issues a fetch request to Thor. There are two different types of commit requests. A write commit reports an object modification to the servers, so that they can update the database. The other type is a read-only commit, which modifies nothing. Commit transactions are atomic in nature, so that once the client issues the request, no objects will be modified until the client receives notification from the server that the commit was successful [10].

The clients run a front-end to communicate with the servers. Usually, the front-end is a separate process running at the client, but it can be run at a remote location. Applications running at the client contact the front-end when they wish to access Thor. The front-end is responsible for making the connection to Thor for the application and sending the fetch request or commit request to the appropriate server. It also handles all replies from Thor, and passes the results of the transaction back to the client application. The front-end also maintains the client object cache, which stores the objects fetched from the server [10].

2.2 Concurrency Control and Validation

Thor uses an optimistic concurrency control scheme for its operations. With the optimistic scheme, client operations are faster than with a pessimistic one, since clients do not pay the overhead from acquiring a lock. However, it is possible in an optimistic scheme that two clients may both modify the same object. Because of this contention, the server will abort the latter transaction, and the client will need to fetch the new value.

For systems that expect little contention, the optimistic scheme is better to use because the savings in overhead will outweigh the lost time from transactions aborting. However, in this thesis, we will assume no contention in the workload [1].

Because of the optimistic concurrency control scheme, Thor servers need a validation algorithm to ensure that commit requests are not using outdated object values. To do this, Thor uses an invalidation based validation scheme. The implementation of this scheme depends on the fetch policy. With fetches at the primary only, validation occurs at the primary server. The dual-fetch algorithm is an extension of the primary fetch algorithm, with validation occurring at every server with objects involved in the commit transaction.

With the basic fetch policy, the primary server keeps a front-end table that stores which clients are caching which objects. When a transaction modifies an object, the primary sends an invalidation message to every client caching that object, informing the client that the object in their cache is invalid. Clients are required to acknowledge this message. Also in the front-end table, the primary stores which clients have not acknowledged the invalidation message. If one of the clients later requests a commit for a transaction involving that same object, then the primary first determines if the client has acknowledged the invalidation. The primary automatically aborts the transaction if there is an outstanding invalidation [1].

If the client has sent an acknowledgment, then the primary uses a validation queue to determine if the commit should be allowed to proceed. The validation queue stores information about all previous successfully committed transactions. The primary compares the objects in the queue with earlier timestamps to objects that are involved in the current transaction. This ensures that there are no conflicts between the new transaction and any previous one [1].

For the dual-fetch scheme, the above algorithm was adopted to use two servers. Both the primary and backup keep a front-end table for the objects that server owns. On an incoming transaction, the primary informs the backup when a modification is being made to an object it owns. Both the primary and backup handle the invalidation for the objects they own that are involved in the transaction. The clients send their acknowledgment directly to the server from which they received the invalidation notice. If a client later requests a commit on one of those objects, then each server will check its front-end table for outstanding invalidations on its objects. If that client

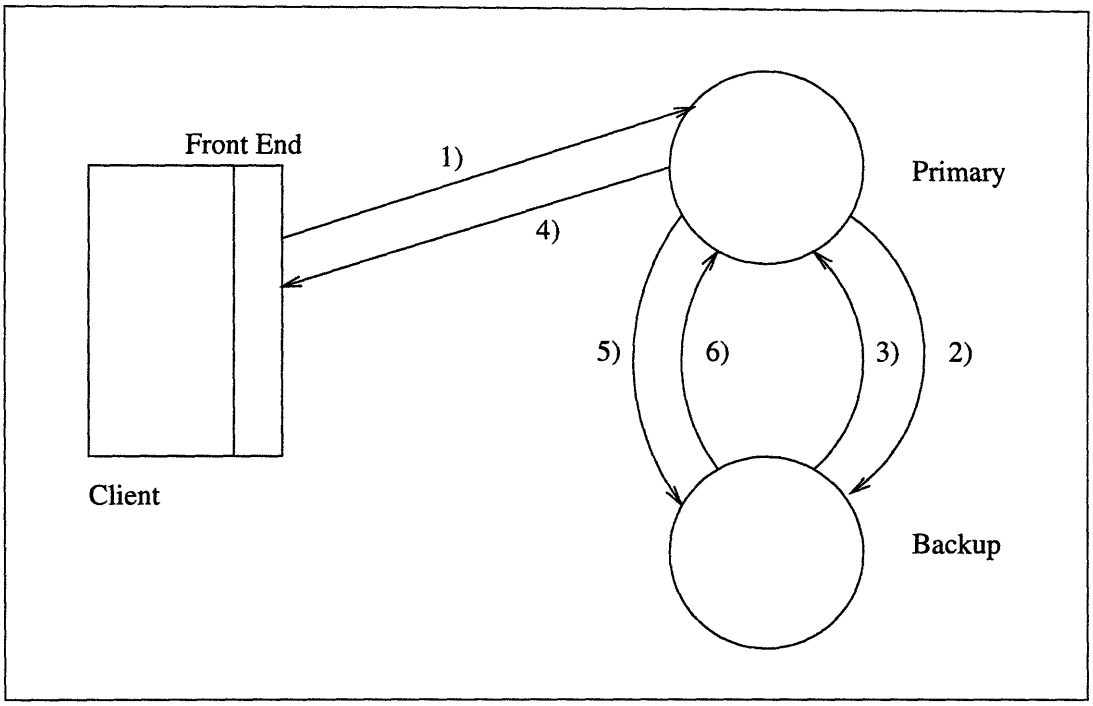

Figure 2-2: Two-Phase Commit Protocol. The arrows indicate the order in which messages are sent.

does have any, then the transaction is aborted. Otherwise, the primary proceeds with using the validation queue [1].

2.3 Commit Protocol

Thor uses a two-phase commit protocol (see Figure 2-2) for deciding whether transactions (message 1) should be allowed to commit and for propagating any object modifications to the backup server. Phase one of the process is the validation phase, in which the primary server sends a copy of the commit request to the backup (message 2). The primary and backup then use the validation algorithm described above to determine if the transaction should be allowed to commit. The backup then sends its vote (message 3) to the primary: COMMIT, if the transaction passes validation, or ABORT, if it does not. If both servers vote COMMIT, then the transaction is allowed to commit. Otherwise, it will be aborted. The primary informs the client of

the result in message 4 [9].

Phase two occurs at the server level. The purpose of this phase is to ensure that the modification is recorded at each server so that they will both present the same view of the database. After the primary has sent the message to the client, it informs the backup of the result of the transaction (message 5). For an aborted transaction, the backup does nothing. If the transaction was successful, then the backup will install the modifications in its log and send an acknowledgment to the primary (message 6). Once the primary has received the acknowledgment, phase 2 is complete [9].

2.4 Disk Updates

Object modifications are stored in a Modified Object Buffer (MOB) before being written to disk. Entries are added into the MOB at commit time. The function of the MOB is to delay the writes, allowing for *write absorption*, modifications occurring for objects in a page that already has pending modifications. All modifications for the same page can be serviced by one write, reducing the number of overall writes needed in the system.

Once the MOB is a certain percentage full, a flusher thread scans the MOB, starting from the oldest modification. As it scans, the thread creates a list of pages that will be written to disk. Once a certain fraction of the MOB has been selected for writing, the scan stops. In Thor, the flusher thread begins when the MOB is 90% full and ends once it has scanned 10% of the MOB. If it waited until the MOB was completely full, commit requests would be forced to wait until space in the MOB was available before they could be allowed to commit.

As soon as the scan is complete, the flusher thread issues the requests. For pages absent from the cache, the ireads are delayed until they can be scheduled opportunistically, as described in [15]. The writes are propagated to disk through a FIFO algorithm. The MOB scheme relies on large page sizes (32-64KB) to amortize the cost of issuing disk writes in this manner [4]. However, for this thesis, we will be interested in traditional page sizes of 4-8KB, and we will use scheduling to reduce the

write cost.

Chapter 3

Experimental Configurations

This chapter will describe in detail the different server configurations and update policies that we will be testing. Included in our presentation will be a discussion of the motivation for using each one, and the benefits we should see from them. We will be using three different update policies running under two different server configurations. The first server configuration will be a simple primary/backup scheme, where the primary processes all requests. The second one will implement the cache-splitting algorithm. The first of the three update policies will be a version of the MOB used in the actual implementation of Thor. The other two will use the STF scheduling algorithm for the ireads and writes. One of them will issue requests one at a time, while the second one will issue them in batches.

3.1 Server Configurations

The basic server configuration, shown in figure 3-1, will be the simpler of the two schemes. The clients will direct all fetch and commit requests to the primary server only. The backup is present for replication purposes only. The main advantage of using this scheme is its simplicity. If the backup server fails, then there is no deterioration in performance, since the primary will continue to process modifications. When the backup is repaired, its version of the database can be updated from the log and disk of the primary. If the primary server fails, then the backup can simply

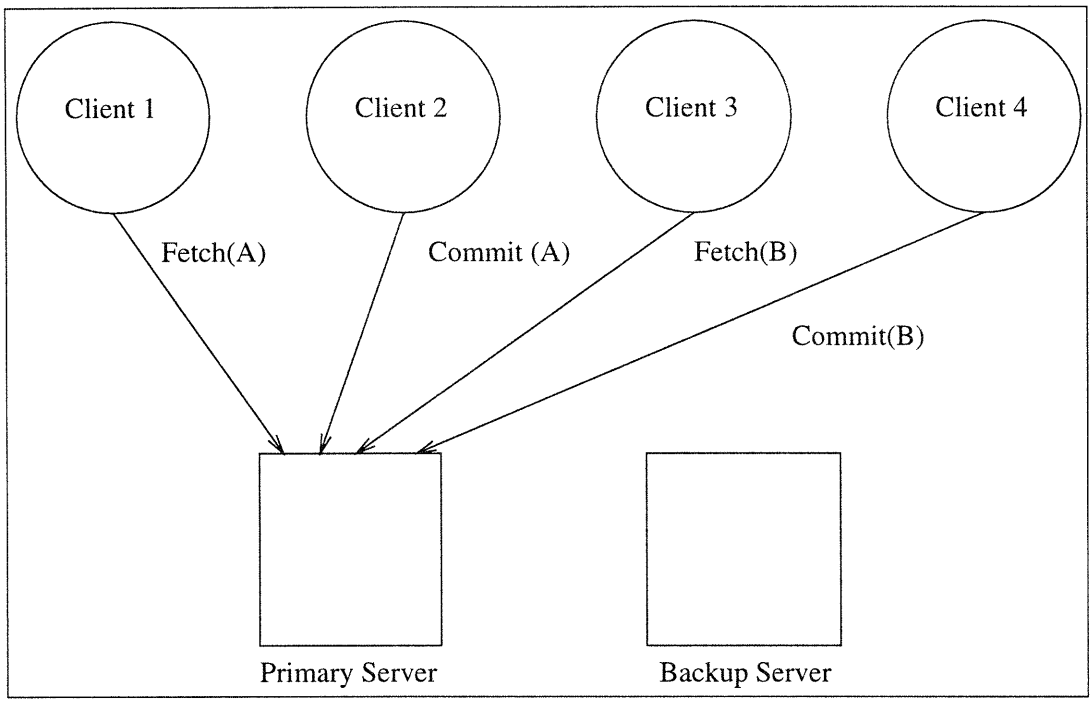

Figure 3-1: Basic Primary-Backup Configuration

take over the duties of the primary without degrading the performance. The backup's copy of the database is current, so it can act as the primary. The primary can then be updated from the log and disk of the backup when it is repaired. However, this scheme wastes resources. The backup server is only used to store another copy of the database, meaning a machine is dedicated to storing another copy of the database.

This disadvantage will be addressed by the dual-fetch scheme, shown in figure 3-2. The pages will be divided between the two servers using the cache-splitting algorithm described in [5]. In our simulator, the primary will own all even pages, while the backup will own the odd ones. Ownership of a page means that the clients will direct all fetch requests to the server that owns it. Therefore, each server will only perform fetch reads for the pages it owns. Also, each server will only perform ireads for the pages it owns.

As above, all commit requests will be processed at the primary. When the primary receives a modification to an odd page that is absent from its cache, it will request the page from the backup server. If it is present at the backup, the server will return it to

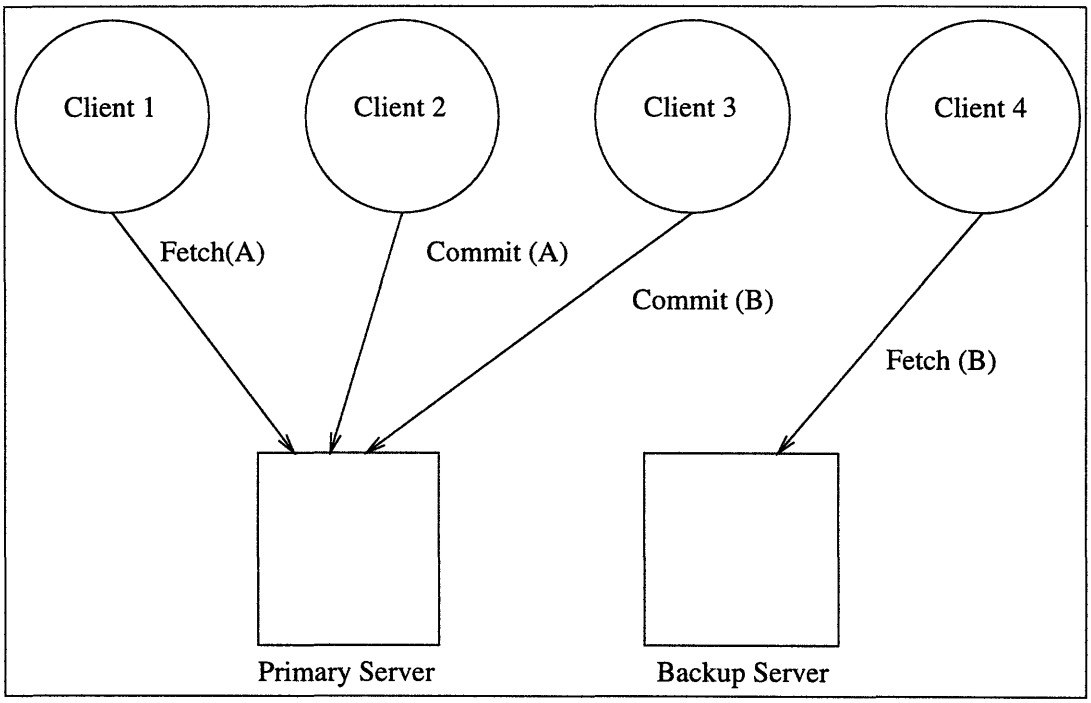

Figure 3-2: Dual-Fetch Primary-Backup Configuration

the primary immediately. If it is absent, the backup will send the page to the primary after an iread for the page has been completed. This also applies to the backup for modifications to even pages. An actual implementation of this scheme would possess a dynamic load-balancing algorithm, so that if one server became slowed by an excessive fetch load, some of its pages can be repartitioned to the other server. We will not include this in our simulator, since it would add nothing. Our workload is randomly chosen, so we should see an equal number of odd and even fetch requests.

This configuration provides us with a great advantage over the basic scheme. Since fetch requests are divided between the servers, the primary will have much fewer fetch reads in this case, given the same size cache. Since both servers are performing fetch reads, the pages will be returned quicker to the clients, meaning they will be processing modifications at a higher rate, improving performance when compared with the basic fetch policy. We will also see a reduction in the number of ireads and fetch reads that will occur in the system because of the increased cache hit rate. Since each server will only be fetching and ireading pages it owns, the cache will be predominantly made

up of pages it owns. The only way in which a page from the other server will be inserted into a server's cache is when that page is modified. After the modifications are installed and the page is written to disk, it can be removed from the cache.

3.2 Disk Scheduling Policies

Several studies, [7, 15, 16, 17], have shown that delaying disk operations, such as writes and ireads, leads to a dramatic improvement in disk utilization. Part of the benefit comes from the possibility that the client will make another modification to the page before a write or iread is issued. Therefore, both modifications can be serviced by the same operation, reducing the total number of operations that will be performed. Since we are using a replicated server environment, delaying ireads and writes will not lead to any data loss should one server fail. The only way modifications could be lost is if both servers fail without writing those modifications to disk. We can also see benefits from scheduling the writes and ireads based on the current disk position. This way, the latency for writes and ireads will be greatly reduced.

All three schemes will delay the disk writes and ireads, but only two of them will use the scheduling algorithm. The other will have a FIFO queue for the ireads. For scheduling, we will use the STF algorithm proposed in [17]. Of the two policies which schedule writes and ireads, one will use a batching scheme, while the other will issue ireads and writes one at a time. As shown in [16], issuing operations in batches should lead to an overall improvement in disk utilization, but any fetch reads must wait behind the batch before they can be processed.

The Read-Modify-Write policy will use the FIFO queue for ireads. It will be a similar policy as the MOB, described in section 2.4. New modifications will enter the log at the tail of the queue unless it modifies an object that is already present in the log. In this case, the new modification will overwrite the old one. If the modification misses in the log, then the server will increase the number of pending modifications that are awaiting ireads. Once this number reaches a certain value, the iread trigger, the server will perform an atomic read-modify-write operation on the page at the

head of the queue. Afterwards, all objects in the queue for that page will be removed, promoting the next object to the head.

Once the number of pending modifications reaches the iread trigger, the server will need to install the modified object at the head of the queue. First, the server must look up the page for that object in its cache. If the page is absent, the server will issue an iread for it. After the iread is complete, all modifications in the log for that page will be installed, and then a write will immediately by issued. Since this is an atomic operation, any fetch reads that arrive while the server is performing the iread must wait until the write is complete before they will be processed. If the page is present in the cache, then the iread is unnecessary. The server will simply install the modifications and write the page to disk. In the dual-fetch configuration, after the atomic operation is complete, the server will send the page to the other server, where it will be modified and written to disk immediately.

The first update policy which schedules its operations is the Opportunistic policy. Basically, it is a greedy algorithm that uses the opportunistic log [15] for storing ireads, and the STF algorithm [17] for scheduling the ireads and writes. When a modification arrives at the server, it checks if the page is present in the cache. If so, the server installs the modification into the page, making it dirty. Otherwise, it places the modification into the set of objects waiting for ireads.

Unlike the Read-Modify-Write, the Opportunistic policy has two trigger values: one for the ireads and one for the writes. For the ireads, the server will use the same algorithm described above for increasing the number of pending modifications in the log. For the writes, we are concerned with the total number of dirty pages in the system, not dirty objects. We counted two modifications to one page that was waiting for an iread as two pending modifications, but we will only count that as one dirty page in the cache. When a modification arrives for a page that is already dirty, the server still installs the object, but it does not increase the number of dirty pages in the cache.

The final policy we are examining is the MBatch scheme, which issues ireads and writes in batches. Unlike the Opportunistic scheme, the MBatch policy uses the exact

same algorithm for recording modified objects as the Read-Modify-Write policy does. Once the number of pending modifications in the log reaches the iread trigger value, the server will select a batch of pages using the STF algorithm. For the pages in that batch absent from the cache, the server will issue a set of ireads. After the iread batch is complete, every modification for the pages in the batch will be installed in the cache, and the batch will be written back to disk. In the dual-fetch case, the server will also send the batch of pages to the other server. Upon receipt of a batch, the server will immediately install all modifications that are waiting in the log for those pages, and it will then write the batch to disk. Unlike the Read-Modify-Write policy, the server is allowed to process fetch reads between the iread and write batches. This should not significantly affect the utilization of this policy, as shown in [17], while it will allow the fetch latency to be reduced, since fetch requests would only have to wait behind one batch instead of two.

One advantage of this policy is that it saves CPU time. The server must pay a setup charge before each disk operation, but when issuing a batch, it must only pay that cost once. This also applies to the network setup cost in the dual-fetch scheme when each server sends an iread batch to the other server. However, as shown in [16], fetch reads must wait behind the batch before they can be processed, increasing the fetch response time. Also, the server can process fetches between the iread and write batches. This essentially makes the first write a random access. Therefore, we need to choose a batch size that is small enough that fetches will not wait too long behind the batch, but large enough to amortize the cost of the first write.

In order to isolate the benefits of the scheduling effects, we will assign the parameters so that the expected write absorption is the same for each policy. The absorption can be measured in the number of expected modifications that are serviced by each write. One modification per write would mean that there was no write absorption at all. If we observed 2 modifications per write, then half of the incoming modifications did not require a separate write for the page. By keeping absorption the same, we ensure that performance differences are due to scheduling effects of the policy.

Chapter 4

Mathematical Model

This chapter will present the mathematical model of the Thor database system. The model will predict how each update policy will perform in each of the server configurations. To do this, we will define several important configuration parameters, and then compute the throughput in each configuration as a function of those parameters. We will use the simulator, described in chapter 5, to validate the expected results we see from the analytical model.

To begin our model, we needed to determine which performance components of the system are most influential on the throughput and model those thoroughly. These components are the server cache hit rate, the absorption of ireads and writes, and the cost of each disk operation. Fetch reads will result when a fetch request misses in the server cache. This will be the most expensive disk operation we perform, since we can't delay the fetch reads. In contrast, the Opportunistic and MBatch policies schedule the writes and ireads from a large set of delayed operations, reducing the cost. Another benefit of delaying the operations is that multiple modifications can be serviced by the same write or iread; they will be absorbed. For our tests, we will control the absorption values for each policy so that each one will have the same number of expected writes from the same number of modifications. This will allow us to isolate the benefits of scheduling the ireads and writes in the MBatch and Opportunistic policies. Next, we must determine the average cost of each type of disk operation. For the delayed operations, this will depend on the number of waiting

pages.

We will then present a system of equations that will determine how many fetch reads, ireads, and writes will result for each policy under each server configuration. These equations will be in terms of the components computed above. These component values will be in terms of the general system parameters, so once we present these parameters in chapter 5, we can then compute the expected throughput for each policy in each configuration.

4.1 Calculations

This section will present the calculations we performed in order to create the mathematical model of the system. Table 4.1 outlines the different variables we will be using in this section and the rest of the chapter when we perform our calculations.

Parameter	Explanation
database_size	Number of pages in the database
O	Number of objects/page
memory size	Number of pages of memory each server has
cache size	Number of pages of memory allocated to the cache
iread_trigger	Number of modifications before an iread is issued
write_trigger	Number of dirty pages before write is issued
P_f	Probability that the client issues a fetch request
P_w	Probability that a commit request is a write commit

Table 4.1: Explanation of Parameters used in the chapter

The first set of parameters in the table determine the size of the database. The second set of parameters are dependent on the server configuration. The third set

of parameters are client workload parameters; they will determine how many fetch requests and modifications will occur in a certain number of transactions.

The first component we will determine is λ, the server cache hit rate. Each server was provided with a certain amount of memory which needed to be partitioned between the page cache and the transaction log. We will first determine the amount of memory required by the log, and from that, we can compute the size of the page cache. Using this value, we can calculate the server cache hit rate.

Next, we determined the write absorption for each policy. We started with the Read-Modify-Write policy. It is the simplest policy to examine, since it issues writes in a FIFO manner. Then we proceed to the MBatch policy, which processes transactions in the same manner as the Read-Modify-Write, but it schedules its writes and ireads. Finally, we examine the Opportunistic policy, which exhibits different iread and write absorption, since it stores the pending modifications and dirty pages in different sets. Once we have determined the absorption for the Opportunistic policy, we will present the method for determining the parameter settings for the MBatch and Read-Modify-Write policies so that they will have the same write absorption.

Finally, we will examine the disk. There are several types of disk operations in our system: fetch reads, Read-Modify-Write writes, and scheduled ireads and writes. We will determine the expected time for each of these operations.

4.1.1 Memory Distribution

Each server possesses a certain amount of memory. That memory needs to be partitioned between the cache and the log in some manner. Independent of policy, the log will store every modification. Entries are removed from the log once the object has been written to disk at both servers. Therefore, our log is composed of, in the Opportunistic case, modified objects that have been written to disk at only one of the servers, object modifications waiting for an iread, and modified objects that have been installed in the cache, but are waiting to be written to disk.

We will only concern ourselves with the Opportunistic policy here. We will set the cache size the same for each policy, meaning that log size will also be the same.

For write absorption, we are going to determine parameter values for the MBatch and Read-Modify-Write policy based on the values in the Opportunistic case. Therefore, it makes sense only to deal with the Opportunistic policy here as well. We will also assume that the servers are using the dual-fetch configuration. The log space should be larger in this case, since the set of objects waiting for ireads will be completely disjoint between servers. However, each server will still store the other server's objects which are waiting for ireads, since all modified objects are stored in the log at both servers.

We will also assume the set of dirty pages is completely disjoint. In terms of log space usage, this is the worst case scenario, since an object is removed from the log only when it has been written to disk at both servers. It is possible that both servers will have written different pages to disk, resulting in disjoint dirty page sets. Therefore, we must consider that the log will have entries that are for a dirty page at either server.

Each server will have *iread_trigger* objects that it owns waiting for an iread that need log space. The *iread_trigger* objects waiting for an iread at the other server will also require log space. Each server will also have *write_trigger* dirty pages in the cache. In the log, there will be at least 1 entry per dirty page, since the log holds the object that was modified and installed into the page to make it dirty. We need space in the log for the dirty pages at both servers. Therefore, the log is at minimum:

$$log\_size > 2 \cdot (iread\_trigger + write\_trigger) \text{ objects} \tag{4.1}$$

However, each dirty page can have several modified objects, and it will only count as one dirty page in the system. As part of our absorption calculations, we will determine the number of modified objects that are serviced by each write. Given the number of dirty pages in the system, $2 \cdot write\_trigger$, we know that there will be one write per dirty page. Normally there is one write at each server per dirty page, but we have already asserted that the dirty page sets are disjoint, meaning that a dirty page at one server has already been written to disk at the other. Given the

total number of writes and the expected number of modifications per write, we can determine the number of modifications required to make that many pages dirty.

$$num\_modifications = 2 \cdot write\_trigger \cdot E(\text{writes/modification}) \tag{4.2}$$

We can now use this number to compute a more accurate lower bound for the log size.

$$log\_size > 2 \cdot iread\_trigger + num\_modifications \tag{4.3}$$

We still need more log space than this to account for writes and ireads that are in progress at any time, since we will still need to hold log space for the objects associated with these disk operations.

Once we have the log size, we can directly determine the cache size by subtracting the size of the log from the total size of the memory.

$$cache\_size = memory\_size - log\_size \tag{4.4}$$

4.1.2 Server Cache Hit Rate

Now that we can determine the size of the server cache, we can compute the expected cache hit rate with the two server configurations. Since the dual-fetch configuration uses the cache-splitting algorithm, it will have a higher cache hit rate than the basic configuration. We will assume a uniform access to pages, i.e. any page is equally likely to be used in a transaction request.

For the basic fetch policy, the primary handles all fetch requests. Therefore, pages will enter the cache from fetch reads and ireads. At the backup, pages will only enter the cache after an iread has occurred. Since all pages have an equal probability of being used, the cache hit rate is just the chance that the page would be in the cache, i.e.

$$\lambda = \frac{cache\_size}{database\_size} \tag{4.5}$$

With the dual-fetch case, each server will only tend to cache pages that it owns.

Pages owned by the other server will enter the cache, but after they are written out to disk, they can be removed from the cache. Therefore, the cache will primarily be made up of pages that the server owns, giving us:

$$\lambda = \frac{cache\_size}{\frac{database\_size}{2}} = \frac{2 \cdot cache\_size}{database\_size} \tag{4.6}$$

as the expected server cache hit rate.

One problem with this equation is that the dirty pages will take up space in the cache at both servers. Regardless of policy, after an iread is complete, the page is sent to the other server to be modified, making it dirty at both servers. For the MBatch and Read-Modify-Write policies, the pages received from the other server are almost immediately written to disk, so they will only be present in the cache for a short period of time. Also, there will only be a few pages owned by the other server in the cache at any time, a small enough number that it will barely affect the cache hit rate. With the Opportunistic policy, it is different. After a page is received from the other server, it is modified and placed in the dirty page set until a write can be scheduled. Therefore, both servers will tend to cache the dirty pages. Since the dirty page set is a non-trivial portion of the cache, we must take that into consideration. This gives us:

$$\lambda_{opp} = \frac{2 \cdot cache\_size - write\_trigger}{database\_size} \tag{4.7}$$

as the server cache hit rate. Since the dirty pages are in the cache at both servers, we must subtract *write_trigger* from the combined cache size of the two servers to avoid double counting.

4.1.3 Write Absorption, Read-Modify-Write policy

As stated previously, reducing the number of writes and ireads is one of the benefits of delaying those operations. *Write Absorption* tells us the number of disk-writes that are avoided because of multiple modifications being made to the same page while it waits. *Iread Absorption* is the same for ireads. In this section, we will predict the

absorption for the Read-Modify-Write policy.

With the Read-Modify-Write policy, write absorption will be the same as iread absorption, so we will only handle write absorption. A write for a page is issued immediately after the iread, so the number of modifications serviced in that write will be the same as the number served by the iread. In some cases, no iread will be necessary, but we should still see the same number of expected modifications per iread as we do writes. Through this analysis, we will assume that the number of pending modifications is at the iread_trigger, i.e. another modification will lead to a write being issued.

To determine the absorption, we will calculate the expected number of objects per write. We will make the following assumptions to start the analysis:

1. Object P_1 had just been inserted into the log
2. Object P_1 is stored in page P
3. There are no other modifications to objects from page P in the log

First, we must calculate the expected number of modifications that will occur before object P_1 is propagated to the head of the queue. At that point, the iread and write for page P will be issued. From that, we must predict how many of the modifications went to objects from page P, giving us the expected number of modifications per write.

Expected Waiting Time

The first step in determining the E(modified_objects/write) is to calculate the waiting time of P_1 in the log. We will define waiting time as the number of modifications that will occur before a write for page P is issued.

Figure 4-1 shows how object P_1 progresses through the log, beginning at insertion time. At that point, there were iread_trigger objects waiting for an iread to be issued, so inserting P_1 causes an iread to be issued. Eventually, more modifications arrive, pushing P_1 to the head of the log. When it is time for the iread for page P to be

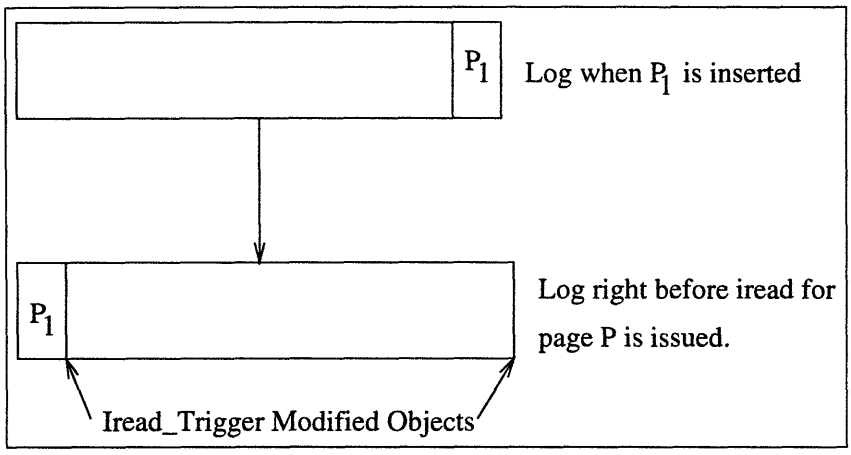

Figure 4-1: Log when P_1 is inserted and finally removed

issued, there will be iread_trigger other objects in the log, just as in the previous state.

In order to get to the later state in figure 4-1, *iread_trigger* modifications must enter the log. Therefore, at least *iread_trigger* modifications must occur before P_1 propagates to the head, since that is the number of modifications behind it in the queue. However, some transactions will modify objects already in the log. These modifications will simply replace the previous modification, not adding to the number of pending modifications. With *iread_trigger* pending modifications and $database\_size \cdot O$ objects in the database, the probability that an incoming transaction modifies an object already in the log is:

$$P(\text{modification overwrites}) = \frac{iread\_trigger}{database\_size \cdot O} \tag{4.8}$$

since each object can be chosen with equal probability. Therefore, the probability that a modification misses in the log and enters at the tail is:

$$\begin{aligned} P(\text{modification misses in log}) &= 1 - P(\text{modification overwrites}) \\ &= 1 - \frac{iread\_trigger}{database\_size \cdot O} \end{aligned} \tag{4.9}$$

We can approximate the incoming modifications as trials in a Bernoulli process.

This is not truly a Bernoulli process, because the probability that a modification overwrites one already in the log is dependent on the previous modifications, and is not independent. However, we can approximate the probability as independent since we are assuming that the choice of an object in a transaction is a uniform, independent selection. Therefore, we can use the Bernoulli process approximation. Each incoming modification will be considered a trial, with a successful trial defined as the modified object entering the log. Above, we calculated the probability that this would occur, so we can calculate the expected number of modifications that will occur before the required *iread_trigger* modifications enter the log. We should expect to see:

$$E(\text{waiting time}) = \frac{iread\_trigger}{1 - \frac{iread\_trigger}{database\_size \cdot O}} \tag{4.10}$$

modifications before an iread for page P is issued.

In the dual-fetch case, the ireads are divided between the two servers as well as the fetches. Therefore, in figure 4-1, the modified objects in the set of pending modifications will all be for objects owned by the server that owns page P. The other server will keep the set of pending modifications for the objects it owns. Since each server has *iread_trigger* objects in its pending modification set, we will have $2 \cdot iread\_trigger$ modifications waiting to be installed.

Since the number of total objects waiting in the log has increased, the probability that a modification will hit an object already in the log will also increase. Our new probability is:

$$P(\text{modification overwrites}) = \frac{2 \cdot iread\_trigger}{database\_size \cdot O} \tag{4.11}$$

This makes the probability that a new modification will enter the log:

$$P(\text{modification misses in log}) = 1 - \frac{2 \cdot iread\_trigger}{database\_size \cdot O} \tag{4.12}$$

Above, we approximated the incoming modifications as a Bernoulli process. This still applies to the dual-fetch case, but we need to compute the expected number of trials necessary to add $2 \cdot iread\_trigger$ entries in the log. Each server adds a

modification to its pending modification set if and only if it modifies an object the server owns. Therefore, we should expect half of the transactions to modify pages that each server owns. In figure 4-1, all the objects in that set are owned by one server. We know that *iread_trigger* modifications must be added to the log before P_1 propagates to the head, but these modifications will only be to pages owned by that server. *Iread_trigger* modifications will also occur to objects owned by the other server during this time. Therefore, we must wait until $2 \cdot iread\_trigger$ modifications have added entries in the log, giving us an expected waiting time of:

$$E(\text{waiting time}) = \frac{2 \cdot iread\_trigger}{1 - \frac{2 \cdot iread\_trigger}{database\_size \cdot O}} \tag{4.13}$$

Calculating Absorption

Now that we have computed the expected number of modifications that will occur before the write for page P is issued, we can compute the expected number of additional modifications that will occur to page P while it is waiting. The probability that a transaction modifies page P is:

$$P(\text{page P is used}) = \frac{1}{database\_size} \tag{4.14}$$

since each page can be used with equal probability.

Pages are chosen randomly, independent of the previous choices. Therefore, we can model the incoming modifications as a Bernoulli process. We will define a success as page P being used in the modification, and we just computed the probability that this will occur. In the previous section, we calculated the expected number of modifications that would occur before the write for page P was issued. Each of these modifications will be a trial in the process, so we can compute the expected number of successes (modifications to page P) that will occur while P_1 is propagating through the log. By Bernoulli process laws, this is:

$$E(\text{hits to page } P) \quad = \quad P(\text{page P is used}) \cdot E(\text{waiting time})$$

$$= \frac{iread\_trigger}{database\_size - \frac{iread\_trigger}{O}} \tag{4.15}$$

This gives us the expected number of additional modifications to page P before the write is issued. To get the final expected modifications/write value, we need to take into consideration the initial modification to object P_1. This gives us:

$$E(modifications/write) = E(\text{hits to page } P) + 1 \tag{4.16}$$

For the dual-fetch case, we should substitute the waiting time computed for the dual-fetch case into equation 4.15 in place of the waiting time computed for the basic fetch case.

The expected number of modifications per write that we predict for a given iread_trigger value are shown in figure 4-2. For comparison, we have included the expected number of modifications per write observed in the simulation trials. Notice that for a small trigger value, the simulation results correspond to the mathematical model. As we increase the trigger value, the observed absorption is only slightly higher than the predicted value.

4.1.4 Write Absorption, MBatch Policy

To compute the absorption for the MBatch policy, we can use the same method as we did with the Read-Modify-Write policy. Incoming modifications are handled the same way by each policy, so we should expect to see the same absorption. However, the Read-Modify-Write policy uses a FIFO algorithm, while the MBatch policy schedules the ireads and writes. We should still observe the same expected waiting time with the MBatch policy, since some modifications will be written out quicker, but some will wait in the log longer than in the FIFO case. If we approximate the scheduling algorithm as a random process, we will see that this is true.

If it were a random scheduling algorithm, then each page with a modified object in the set would have an equal probability of being issued. If we expect to see E(modifications/write) modifications written out by each write, with *iread_trigger*

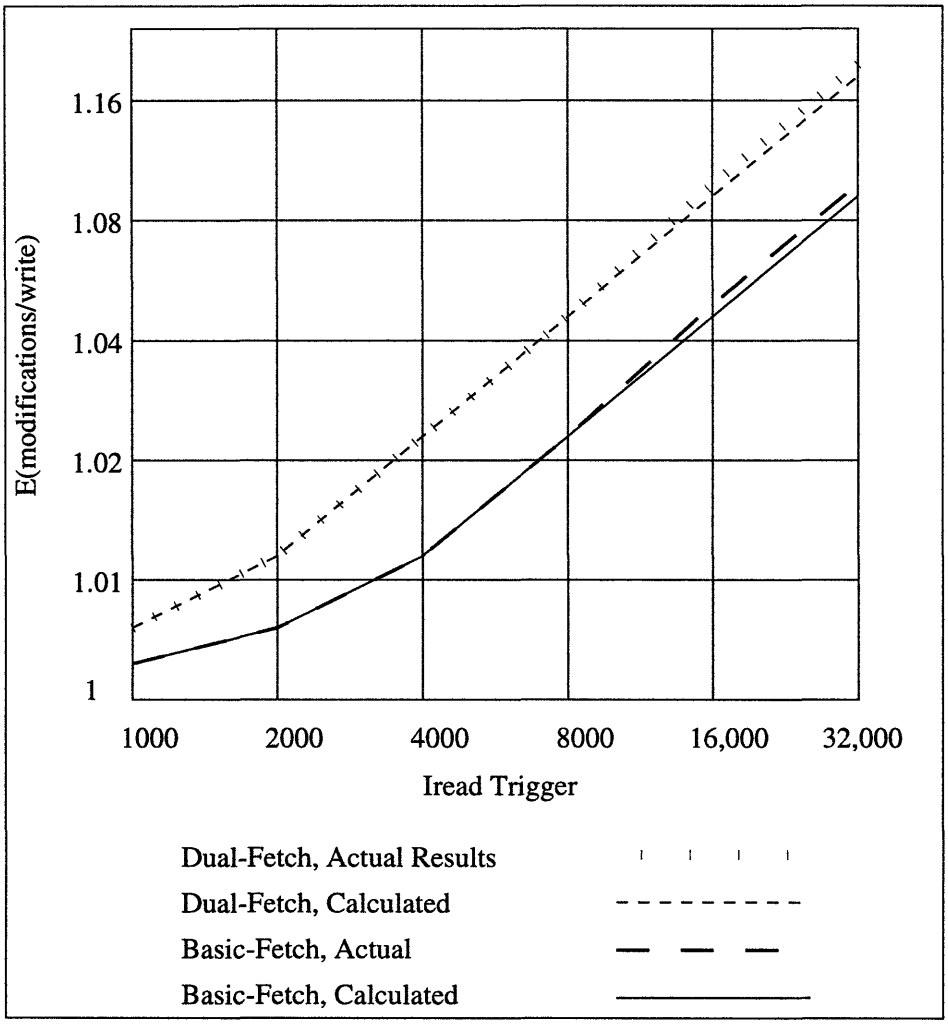

Figure 4-2: Iread Absorption for several trigger values

total modifications in the set, then there should be:

$$\text{num pages in the log} = \frac{iread\_trigger}{E(\text{modifications/write})} \tag{4.17}$$

total pages with pending modifications in the log. Therefore, the probability that a write would be issued for page P would be:

$$P(\text{write for page } P) = \frac{1}{\text{num pages in log}} = \frac{E(modifications/write)}{iread\_trigger} \tag{4.18}$$

Since we are assuming that the scheduling is an independent random process, we

can model the page selection as a Bernoulli process. We will define a write being issued as a trial, and a success as page P being selected for a write. We want to compute the expected number of writes that will occur before the write for page P is issued, which is the first interarrival time in a Bernoulli process. We should expected to wait:

$$E(\text{number of writes}) = \frac{1}{P(\text{write for page P})} = \frac{iread\_trigger}{E(\text{modifications/write})} \tag{4.19}$$

writes before the write for page P is issued.

Each of the writes that will occur in the system will write E(modifications/write) modified objects to disk. Therefore, E(modifications/write) modifications will occur for each disk write. This gives us:

$$E(\text{number of writes}) \cdot E(\text{modifications/write}) = iread\_trigger \tag{4.20}$$

modifications that will occur before the write for page P is issued. However, this only counts modifications that do not hit objects already in the pending modification set. In the analysis for the Read-Modify-Write policy, we concluded that *iread_trigger* modifications that missed in the log would need to occur before the write for page P was issued. We have concluded the same thing here, so, since we still assume the uniform probability distribution, we should expect to see the same absorption with the MBatch policy as we did with the Read-Modify-Write.

4.1.5 Iread and Write Absorption, Opportunistic Policy

In the Opportunistic policy, ireads and writes are scheduled concurrently, so we must separately compute the iread and write absorption. However, we will still use the same methodology as above; computing the expected number of modifications before the iread/write for page P is issued, and then determining the expected number of modifications that will be for page P. The probability that page P will be modified is the same, since we still assume the uniform probability distribution, so the only

difference will be in computing the expected waiting time,

Iread Absorption

For this analysis, we will make the same assumptions about page P that we did for the Read-Modify-Write analysis. For this section, we will assert that page P is absent from the server cache, so it will require an iread before the modifications can be installed.

Above, we used:

$$E(\text{waiting time})_{rmw} = \frac{iread\_trigger}{1 - \frac{iread\_trigger}{database\_size \cdot O}} \tag{4.21}$$

as the expected number of modifications that occurred before an iread was issued for page P. We will use the same waiting time, but here it will represent the number of modifications that require ireads that occur before the iread for page P is issued. To determine the total number of modifications that will occur, we need to determine the probability that a modification needs an iread.

An iread will be required for an incoming modification if the page it is modifying is absent from the server cache. If the modification is preceded by a fetch, then the page will always be present in the cache, since it will be present in the cache after the fetch. So, the probability that a modification will require an iread is:

$$\begin{aligned} P(\text{modification needs iread}) &= P(\text{server cache miss}) \cdot P(\text{not a fetch}) \\ &= (1 - \lambda) \cdot (1 - P_f) \end{aligned} \tag{4.22}$$

As with the Read-Modify-Write analysis, the probability of a cache miss is not an independent probability. It is dependent on the previous fetch requests and ireads. However, we can approximate the modification arrival as a Bernoulli process. For this case, we will define a success as the modification requiring an iread. We have already given the probability of a success, and we have already shown that we will need to wait $E(\text{waiting time})_{rmw}$ successes before the iread for page P is issued. Therefore,

we should expect:

$$E(\text{iread waiting time}) = \frac{E(\text{waiting time})_{rmw}}{P(\text{modification needs iread})} \tag{4.23}$$

modifications to occur before the iread for page P is issued. As with the Read-Modify-Write analysis, the expected number of modification per iread is:

$$E(\text{modifications/iread}) = E(\text{iread waiting time}) \cdot P(\text{page } P \text{ is used}) + 1 \tag{4.24}$$

For the dual-fetch case, we should substitute the waiting time computed for that case into equation 4.23 instead of $E(\text{waiting time})_{rmw}$. This will give us a longer expected iread waiting time, since we must wait for modifications to occur to pages at both servers before an iread for page P will be issued.

Write Absorption

For the write absorption analysis, we will assume that page P is now present in the cache. For simplicity, we will assume that page P was present in the cache at the time it was first modified. However, the analysis would still be the same for the case where P was inserted into the cache through an iread; this part would provide how many additional modifications were made to page P while it waited in the cache. We will also assume that the simulator is in the steady state, and that there are *iread_trigger* pending modifications and *write_trigger* dirty pages.

In computing the number of modifications that will occur before the write for page P is issued, we must not only consider modifications that are to pages in the cache. Even if a modification is for a page absent from the cache, then an iread will be issued, resulting in a new dirty page, which causes a write to be issued. The only ways in which a new modification would not cause a write would be if the modification overwrote an object in the pending modification set, or if it modified a page that was already dirty. The server does not increase the number of pending modifications if the modification overwrites an object already in the set. If the transaction modifies

a page that is already dirty, then the number of dirty pages does not increase; the server only counts each page once towards the *write_trigger* value.

Previously, we computed the probability that a modification overwrites one in the log. We will use a similar computation here, taking into account the fact that the modification must require an iread first before we are concerned if it overwrites an object in the pending modification set. This gives us:

$$\begin{aligned} P(\text{mod hits in log}) &= P(\text{mod needs iread}) \cdot P(\text{object in log}) \\ &= (1-\lambda) \cdot (1-P_f) \cdot \frac{iread\_trigger}{database\_size \cdot O} \end{aligned} \tag{4.25}$$

The probability that a modification is to a page already in the dirty page set is:

$$\begin{aligned} P(\text{mod absorbed}) &= P(\text{mod doesn't need iread}) \cdot P(\text{page dirty} \mid \text{page in cache}) \\ &= (1-((1-\lambda) \cdot (1-P_f))) \cdot \frac{write\_trigger}{cache\_size} \end{aligned} \tag{4.26}$$

We need to use the probability $P(\text{page is dirty} \mid \text{page present in cache})$ rather that just the probability of a page being dirty. In computing this probability, we are assuming that the modification is to a page already in the cache. This limits the set of possible pages the modification could hit to the ones in the cache. Therefore, we are computing the probability that a page in the cache is dirty, giving us $\frac{write\_trigger}{cache\_size}$ as the value.

The sum of equations 4.25 and 4.26 represents the probability that a modification will not cause a write. We can add the two since the two events are disjoint; a page can not be dirty in the cache and require an iread simultaneously. This gives us the probability that a modification causes a write as:

$$P(\text{mod causes write}) = 1 - P(\text{mod absorbed}) - P(\text{mod hits in log}) \tag{4.27}$$

We must do an analysis similar to what we did for the MBatch policy in order to determine the number of writes that will occur before the write for page P is issued. Again, we will approximate the scheduling process as a random process, with the

probability that page P will be picked as:

$$P(\text{write for page } P) = \frac{1}{write\_trigger} \tag{4.28}$$

since there are *write_trigger* dirty pages in the cache. Therefore, as with the MBatch case, we get:

$$E(\text{num writes}) = \frac{1}{P(\text{write for page } P)} = write\_trigger \tag{4.29}$$

Write_trigger writes are expected to occur before the write for page P is issued.

For the iread absorption, we were able to approximate the transaction arrivals as a Bernoulli process. We will use the same reasoning to determine how many modifications will occur before the write for page P is issued. We will define a success in the trials as the modification causing a write, either through making a page dirty or forcing an iread. We computed the probability of this occurring before, and we just determined that *write_trigger* writes will occur before the write for page P is issued. Therefore:

$$E(\text{write waiting time}) = \frac{write\_trigger}{P(\text{mod causes write})} \tag{4.30}$$

modifications will occur before the write for page P is issued.

To compute the write absorption, we must consider two cases: when P requires an iread before the write, and when P is present in the cache when the first modification is made. This will provide us with a more accurate value for the number of modifications that each write will service. We should use the equation:

$$\begin{aligned} E(\text{modifications/write}) &= [P(\text{modification need iread}) \cdot (P(\text{page } P \text{ is used}) \\ &\cdot (E(\text{iread waiting time}) + E(\text{write waiting time})))] \\ &+ [(1 - P(\text{modification need iread})) \cdot P(\text{page } P \text{ is used}) \\ &\cdot E(\text{write waiting time})] + 1 \end{aligned} \tag{4.31}$$

to calculate this. The first term is the expected number of modifications that will occur while a page is waiting for an iread before the write. We sum the two waiting times, because the page will wait for E(iread waiting time) modifications before the iread is issued and then E(write waiting time) modifications before the write is issued. Their sum is the total number of modifications that will occur before the write is issued. The second term covers the case where the page is present in the cache when the modification is made. It will only wait for E(write waiting time) modifications before the write is issued. Finally, we add 1 to account for the initial modification to the page.

4.1.6 Iread Trigger Values

After computing the expected number of modifications per write for the Opportunistic policy, we can compute the iread trigger values for the MBatch and Read-Modify-Write policies. We will initially set the trigger values for the Opportunistic policy since there are two of them. If we were to try and compute them given the iread_trigger for the MBatch policy, there would be several combinations of iread and write triggers that would yield the same absorption. Choosing one combination over another could significantly affect the system performance. Therefore, we will set the iread and write triggers for the Opportunistic policy and solve for the iread trigger in the MBatch and Read-Modify-Write policies.

To solve for the iread_trigger value, we will backsolve the equation:

$$E(modifications/write) = \frac{iread\_trigger \cdot O}{database\_size \cdot O - iread\_trigger} + 1 \qquad (4.32)$$

from section 4.1.3. For the dual-fetch policy, we will be using the equation:

$$E(modifications/write) = \frac{2 \cdot iread\_trigger \cdot O}{database\_size \cdot O - 2 \cdot iread\_trigger} + 1 \qquad (4.33)$$

Solving the equation for the iread_trigger gives us

$$iread\_trigger = \frac{database\_size \cdot O \cdot (E(modifications/write) - 1)}{E(modifications/write) + O - 1} \tag{4.34}$$

for the basic fetch policy. In the dual-fetch case, the equation would be the same, but we would be solving for $2 \cdot iread\_trigger$ instead.

Given the iread and write trigger values, we can use the equations from the previous section to compute the expected number of modifications per write. We can then insert that value into equation 4.34 to compute the iread_trigger for the MBatch and Read-Modify-Write policies.

4.1.7 Mathematical Disk Model

There were three types of disk accesses that we needed to consider: random reads, Read-Modify-Write writes, and opportunistic accesses. Each update policy needed to use the random reads to process fetch requests that missed in the cache. These were unscheduled, and the disk head moves to any location on the disk. In addition to the fetch reads, the MBatch policy uses a random access for the first write in a write batch. Between the iread and write batches, fetch reads can be processed. This could move the disk head to a random part of the disk, so the MBatch policy must pay the random access time to return the disk head to the batch of dirty pages. The Read-Modify-Write policy also used the random access time for its ireads, since it issued them in a FIFO manner.

The Read-Modify-Write also used a special write access time. It issued the write immediately after the iread, so the disk head simply needed to rotate once before the write could be issued.

Opportunistic accesses were the fastest disk access times. They occurred when the MBatch and Opportunistic used the STF algorithm to schedule an iread or write based on proximity to the current disk position. This resulted in the improved performance over the random accesses.

Random Reads

We computed the average random access with the following equation:

$$t_{rand} = t_{avg\_seek} + .5 \cdot t_{rot} \tag{4.35}$$

With the random access, the disk head could be moving from any location on the disk to any other. After the seek, the disk head may need to rotate one sector, or one complete revolution. Since any page on that cylinder could be the page being read, we take the average rotation time that will be required to reach a page.

Given the number of revolutions per minute, we can calculate the rotation time in milliseconds per rotation by the equation:

$$t_{rot} = \frac{1000 \cdot 60}{disk\_rpm} \tag{4.36}$$

Dividing the disk_rpm by 60 converts it to the number of revolutions per second that the disk spins. To convert to milliseconds, we divide that result by 1000. This will provide the number of revolutions per millisecond, so we can use 1 over this amount to obtain the milliseconds per revolution.

With the random access, the starting and ending points could be anywhere on the disk, with every location being equally likely. Therefore, when computing the average seek, we considered every possible combination of beginning and ending cylinder. There were num_cyls^2 of these combinations. To calculate the average seek time, we wrote a program that summed the seek time between every combination of starting and ending cylinders. We included cases when the starting and end cylinder were the same, since in a random access, the page had a $\frac{1}{num\_cyls}$ chance of being on the same cylinder as the disk head. To compute the average seek time, the program divided the total seek time by num_cyls^2. The values for the rotation time, average seek, and random access time are shown in table 4.2 for both present and future parameters.

Read-Modify-Write Writes

This write is used in the atomic iread-modify-write cycle of the Read-Modify-Write policy. It is only used when the write is preceded by an iread. If the page is present in the cache, then no iread is necessary, so the modifications can be installed, and the page can be written to disk. This write will incur a random access charge, since we are using the FIFO method of propagating modifications to disk.

After the iread is complete, the disk continues rotating to move the disk head back to the beginning of the page. While the disk is spinning, the modifications are installed into the page in the server cache. When the disk reaches the beginning of the page, the page is written to disk.

Equation 4.36 determined the rotation time for the disk. From this, we need to compute the time required to rotate past each sector on the cylinder. This time is given by:

$$t_{sector} = \frac{t_{rot}}{sectors\_per\_track} \tag{4.37}$$

After the completion of the iread, the disk head will be at the end of the page's sector. Therefore, the time required to perform the write will be:

$$t_{rmw\_write} = t_{rot} - t_{sector} \tag{4.38}$$

Table 4.2 shows the values we computed for both the random reads and Read-Modify-Write writes, providing times for both the present and future disk speeds.

Opportunistic Accesses

Several factors influence the opportunistic access time. The number of pages in the set we are choosing from, which pages are in the set, and the current disk head location. Correlating these factors into one expression would be difficult to perform mathematically, since even if we set the number of pages, the results will be heavily influenced by which pages we selected for the set and where we start our search from.

Instead of attempting to model this mathematically, we used the simulator results

	Present	Future
Average Seek	11.18 msec	8.39msec
Rotation Time	11.11 msec	6.30msec
Random Access Time	16.73 msec	11.54 msec
RMW Write Time	9.99 msec	5.67 msec

Table 4.2: Present and Future Disk Access Times

to determine the average opportunistic time for each value of the iread trigger. As part of its output, the simulator reports the average iread and write times, so we set the iread trigger at different values and used the printed results for the average time. The results from this are graphed in figure 4-3. The top line, with the slower times, represents the average opportunistic access time with the present disk parameters, while the bottom line shows the values with the future disk parameters.

4.2 Throughput Equations

This section will present equations that will determine the maximum steady-state throughput for each disk update policy. Since we are studying the I/O bound, the disk will be the bottleneck in determining the maximum throughput. If we can calculate the number of fetch reads, writes, and ireads that should result from a certain number of modifications, we can determine the amount of time required by the disk to process those operations.

With I/O bound workloads, the maximum throughput will be achieved when the disk is fully utilized. The system will not be able to handle a higher transaction arrival rate, since the disk is already operating at full utilization. We have determined the cost of each type of disk operation, and the equations we are about to present will

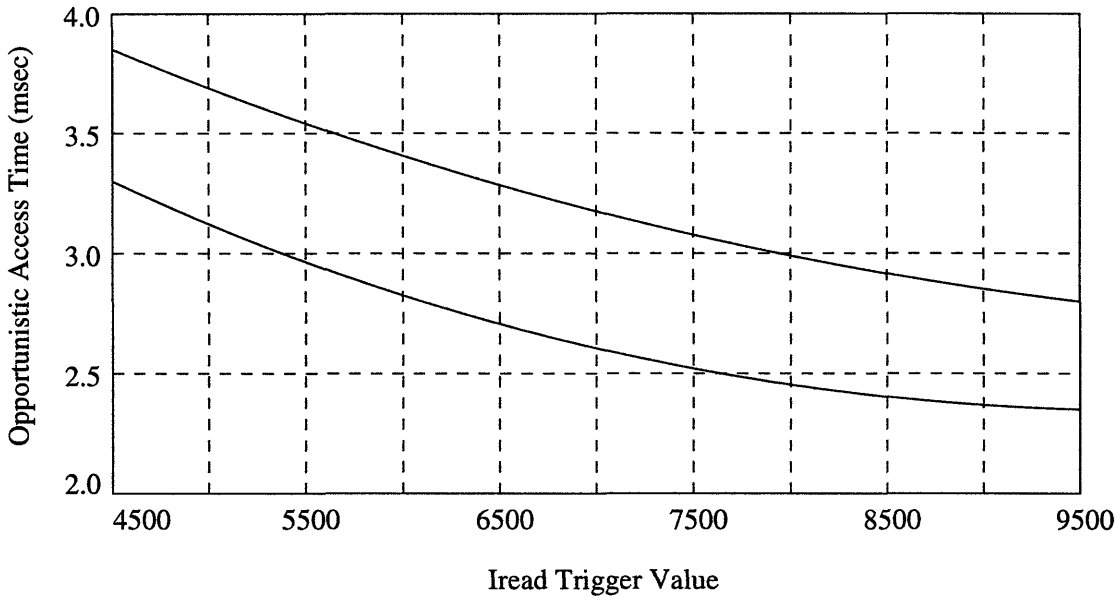

Figure 4-3: Opportunistic Disk Access Times

tell us the number of each disk operation. This will allow to to calculate the amount of time required by the disk to process all of the requests resulting from a certain number of transactions. Then we can calculate the throughput by:

$$throughput = \frac{num\_transactions}{\text{total disk time}} \tag{4.39}$$

Table 4.3 shows the parameters that we will be using for the equations. The parameters A_i and A_w are related to iread and write absorption respectively. They represent the number of ireads or writes that will be generated at one server by each incoming modification. We have already showed how to calculate the expected number of modifications per iread and write, so we can calculate A_i and A_w in terms of those values.

$$\begin{aligned} A_i &= \frac{1}{E(\text{modifications/iread})} \\ A_w &= \frac{1}{E(\text{modifications/write})} \end{aligned} \tag{4.40}$$

For the MBatch and Read-Modify-Write policies, $A_i = A_w$, since the absorption is

Parameter	Explaination
T	Number of transactions
P_f	Probability that the client issues a fetch request
P_w	Probability that a commit request is a write commit
B	Batch size in the MBatch policy
λ	Server Cache Hit Rate
A_i	Number of ireads each modification needs
A_w	Number of writes each modification needs

Table 4.3: Parameters used in the throughput equations

the same for both ireads and writes with these policies. In the Opportunistic policy, we calculated iread and write absorption separately, so A_i and A_w will be different. However, A_w will be the same for all policies, since we are holding write absorption the same for each policy.

4.2.1 MBatch Policy

We will start by presenting the equations for the MBatch policy. Most of the equations will be the same for all three policies, or the version of the equation for another policy will be derived from the equations for the MBatch policy.

The total number of fetch requests that results from T transactions is:

$$num\_fetch\_requests = T \cdot P_f \tag{4.41}$$

However, a fetch read will only result when the page being requested is absent from the cache. Therefore, we should expect to see:

$$fetch\_reads = T \cdot P_f \cdot (1 - \lambda) \tag{4.42}$$

fetch reads at the primary server with the basic fetch policy. In the dual-fetch case, the fetch reads are divided between the two servers. Therefore, the primary server will only be doing half of the fetch reads, so we modify the equation to:

$$fetch\_reads = \frac{T \cdot P_f \cdot (1 - \lambda)}{2} \tag{4.43}$$

for the dual-fetch case.

The total number of modifications in the system will be:

$$num\_modifications = T \cdot P_w \tag{4.44}$$

This will yield a total of

$$num\_writes = T \cdot P_w \cdot A_w \tag{4.45}$$

writes, since some of the modifications will be absorbed, reducing the total number of writes in the system. Since we are working with the MBatch policy, we must determine the number of write batches that will be issued. We must charge the first write in the batch the random access charge, while the other writes will only be charged the opportunistic time charge. Since there are B writes in a batch, there will be:

$$first\_writes = num\_batches = \frac{T \cdot P_w \cdot A_w}{B} \tag{4.46}$$

Therefore, we will have :

$$regular\_writes = \frac{B - 1}{B} \cdot T \cdot P_w \cdot A_w \tag{4.47}$$

writes incur the opportunistic charge, since only one write in each batch will be charged the random access time.

Finally, we need to examine the ireads. Ireads will only occur on a write commit that was not preceded by a fetch. If a fetch requests did occur before the modification, then the page will already be present in the cache, so no iread will be necessary. For the write commit not preceded by a fetch, an iread will result only in the case of a

cache miss. When we factor in the iread absorption factor, we get:

$$num\_ireads = T \cdot P_w \cdot (1 - P_f) \cdot (1 - \lambda) \cdot A_i \tag{4.48}$$

As with the fetch reads, in the dual-fetch case, we will be dividing the ireads between the two servers. Therefore, each server will only perform:

$$num\_ireads = \frac{T \cdot P_w \cdot (1 - P_f) \cdot (1 - \lambda) \cdot A_i}{2} \tag{4.49}$$

ireads.

4.2.2 Read-Modify-Write

For the Read-Modify-Write policy, we can use most of the analysis from above in our analysis here. We should expect to see the same number of fetch reads, ireads, and total writes, since none of the parameter values will be changing. A_w is the same for all three policies, and $A_i = A_w$ for the MBatch and Read-Modify-Write policy. Even though there will be the same number of each type of disk operation, we will need to divide the writes into two categories, as we did with the MBatch case.

The first type of write in the Read-Modify-Write is the write that follows an iread, when the disk head simply rotates back to the page for the write. Since these writes only occur after an iread, we know that:

$$rmw\_writes = num\_ireads = T \cdot P_w \cdot (1 - P_f) \cdot (1 - \lambda) \cdot A_i \tag{4.50}$$

For the dual-fetch case, we will again have the same number of rmw_writes as ireads, so we will need to divide the above total by two to account for the splitting of the ireads.

The second type of write occurs when the page that is being written to disk is already in the cache, eliminating the need for an iread. These writes will incur a random access disk charge, since the Read-Modify-Write uses a FIFO method for issuing writes. Since we know the total number of writes and ireads that will occur,

we can compute the number of writes that will have a random charge by subtracting:

$$random\_writes = num\_writes - rmw\_writes \tag{4.51}$$

For the dual-fetch case, we do not need to make any modifications to this equation. After a server has read a page into its cache for modification, it will send the page to the other server, which will write the page to disk immediately. Therefore, this write will incur a random charge as well. The only writes that will not are the rmw_writes which follow an iread.

4.2.3 Opportunistic

Again, we can use most of the same equations that were used for the MBatch policy. The fetch reads, ireads, and writes will all be computed in the same manner. However, $A_w \neq A_i$ with the Opportunistic policy, since the ireads and writes are scheduled separately. Therefore, we should use the iread absorption and write absorption computations from section 4.1.5. Also, every write will have the same access time, since the Opportunistic policy uses the STF algorithm to schedule all ireads and writes.

For the dual-fetch policy, we can again use the same equations used in the dual-fetch case with the MBatch policy. However, in addition to the separate value for A_i, we need to use the value of λ computed for the Opportunistic policy. In the dual-fetch scheme the dirty pages will be in the cache at both servers, lowering the server cache hit rate. This will increase the total number of fetch reads and ireads that need to be performed. We do not need to modify the write equations, since all writes are selected from the dirty page set, independent of which server they are owned by. When one server issues an iread for a page and sends it to the other server, the other server simply adds it to its dirty page set, and issues it when it is the closest page to the current disk position.

Chapter 5

Simulation Environment

In chapter 4, we presented a mathematical model of the Thor database system. This model allows us to predict how each update policy will perform in any server configuration in terms of the database parameters. In order to determine the accuracy of this model, we could test the different update policies on the actual implementation of Thor. This would give us realistic, accurate results as to the validity of our model.

However, validating the model using Thor is unfeasible. For one thing, we would need to implement each update policy in Thor. Also, changing parameter values for the system would require major changes in Thor and possibly the hardware running Thor. Performing our tests on Thor also makes it difficult to infer how the update policies will perform on future hardware. We can estimate how technology will improve in five years, so we can predict how these policies will perform in the future with our analytical model. Testing the future disk and processor speeds would be impossible on the actual implementation of Thor, because we are testing technology that has not been developed.

Therefore, we developed a simulator of Thor on which to perform our tests. Using the simulator allows us to easily vary the system parameters. Also, we can test the future hardware just as easily on the simulator; we simply change the parameters. The results will not be as accurate as they would be on an actual implementation, but they will indicate how each update policy will perform.

This chapter will present the simulation model and describe how we implemented

it. Section 5.1 describes the simulator model, providing all of the relevant parameters. The tables show both the present values for system parameters and predicted values for five years in the future. Section 5.2 will describe more of the implementation details of the simulator. Section 5.3 shows the statistics that the simulator gathers as it runs. We use these statistics to analyze the results.

We will also derive some specific parameters that were presented in the previous chapter, such as the amount of memory dedicated to the log, and the iread trigger for the Read-Modify-Write and MBatch policies, which are based on the absorption achieved by the Opportunistic policy. After we have derived the parameters, we will be able to then use the equations from section 4.2 to compute the expected throughput each policy will achieve under each configuration.

5.1 Simulator Model

This section provides the details of the simulation model. The environment that we are simulating is the Thor database system, described in chapter 2. We implemented some aspects of the simulator in much more detail than others. Some parts of the model would have no impact on the final results, so they did not need to be fully modeled. The following sections will show these areas and provide the reasons why they are less important. There are also several parameters that we did not take into consideration. However, we only ignored parameters which would not affect the outcome of the trials, i.e. charges to the server that would be the same for every policy.

5.1.1 Database

With 330,000 pages, at 4KB per page, we have about 1.3GB of information in the database. This amount will completely fill a standard hard disk that is standard in most personal computers at present. Most workstations will have larger hard drives, but they must also store additional information other than the database, so we chose these values to reflect a reasonably sized hard drive available. As mentioned

Parameter	Value
Number of Pages	330,000
Page Size	4KB
Objects per Page	10

Table 5.1: Database Parameters

in section 2.4, Thor's modified object buffer relies on a large page size in order to amortize the write cost. However, for our simulation, we will use a more tradition page size of 4KB.

The number of objects per page will affect the number of entries that we can store in the log. Log size is allocated in pages, while the entries in the log are objects. Although transactions in Thor reference objects, our transactions simply reference pages. We will assume a uniform distribution of objects within a page, meaning each object in the page will be accessed in turn. Therefore, the number of objects per page will only be used during allocation of log space. Each entry in the log represents an object modification. Therefore, the maximum log space we will allocate to one page is the number of objects/page. If more modifications than that are made to a page, then the new modifications will overwrite an old one, not increasing the amount of log space used.

For the future trials, we will not change the value of the database parameters. This allows us to isolate the benefits each policy will see by using faster hardware. Currently, we have set the size of the database to be approximately the size of an average hard drive. If we were to resize the database for a standard hard drive five years from now, then we would need to take the new size of the database into consideration when we analyze the future results in comparison to the present results.

5.1.2 Clients

In our simulation, we modeled the clients as probabilistic workload generators.

Parameter	Present Value	Future Value
Clients Logged into System	2, 4, 8, 16	2, 4, 8, 16
Client Cache Hit Rate, Normal Fetch Load	95%	95%
Client Cache Hit Rate, Heavy Fetch Load	80%	80%
Client CPU Speed	25 MIPS	100 MIPS
Client Compute Time	25,000 Instructions	25,000 Instructions
Percent Read Only Transactions	80%	80%

Table 5.2: Client Parameters

Therefore, we did not explicitly model the client cache; we only used the mathematical probability of a cache hit. This decreases the overall accuracy of the simulation, because a page may hit in the cache in one transaction, but may require a fetch in the next one. However, this method will allow us to explicitly control the workload at each server, ensuring a certain percentage of fetch requests to the servers. We tested two different values for the client cache hit rate: 95% and 80%. In the first case, we will examine a system in which there were few fetches, meaning that the emphasis of the disk activity at the servers will be on the writes and ireads. With the second case, the fetch reads will take up a significant percentage of the disk activity, testing how each policy handles the increased number of fetch reads.

For each server configuration, we ran trials with several different number of clients using the database. This allowed us to control the number of requests that were being sent to the database. We chose not to simulate 32 clients, since prior to that, each client would have reached its maximum throughput. Therefore, the additional clients would only make the tests take longer without yielding any new results.

For their transactions, each client had access to the entire database. The clients chose the page for the transaction with a uniform probability distribution over the database. There were two types of commit requests in the system: read-only commits, and write commits. Read-only commits were simply acknowledged by the server,

while write commits modified objects. Only 20% of the commits were write transactions. After the commit request had been sent to the server, the client waited until receiving a reply from the server that the transaction had committed. After the reply, the clients would start a new transaction. In the cases when a fetch request was necessary, the client waited until the server returned the page and then initiated the commit request.

We only chose to simulate the client CPU speed at 25 MIPS. Although faster processors are available now, we expect that in addition to the front-end, the clients will be running other applications concurrently. Although the CPU would be running at a faster rate, when viewed as a single process system running just the front-end, it would appear that the CPU was running at a slower rate. Therefore, we simply chose to use 25 MIPS as the benchmark for clients. By using a projection from [8], we set the future value at 100 MIPS for the client CPU speed. Again, we assume that the clients will be running several concurrent processes.

5.1.3 Servers

Parameter	Present Value	Future Value
Number of Servers	2	2
Server CPU Speed	50 MIPS	200 MIPS
Memory Size, Large	30,000 Pages	30,000 Pages
Memory Size, Small	10,000 Pages	10,000 Pages
Iread Trigger Value	4,500 Delayed Installations	4,500 Delayed Installations
Write Trigger Value	4,500 Dirty Pages	4,500 Dirty Pages
Transaction Validation Time	50,000 Instructions	50,000 Instructions
Modification Installation Time	50,000 Instructions	50,000 Instructions

Table 5.3: Server Parameters

We modeled the servers in much greater detail than the clients. Because we were

using scheduling algorithms based on disk head position, we needed to know the location of the disk head at all times. Therefore, the simulator needed to keep track of which pages were in the cache so that the fetch requests which required a disk read could be determined. We also needed to store the dirty pages in the cache, so that we could use the STF algorithm to schedule the closest one. As with the cache, we needed to fully simulate the log, since all modifications are stored in the log. The server also schedules its ireads from the log, so we need to accurately record the set of objects awaiting an iread so that the server could perform the scheduling algorithm on that set. We needed to know the disk head position at all times, so that meant we needed to completely model the cache and log so that we would know the page involved in each disk operation.

Each server has a certain amount of memory that was divided between the cache and the log. Later in this chapter, we will derive the amount of memory that should be partitioned to the log. There were two sizes of memory used, but we allocated the same amount to the log regardless. For the Opportunistic policy, we kept the iread and write triggers the same for all tests, so we needed enough log space to store all of the modifications. Although the maximum log size was determined in pages, the log entries were objects. Therefore, if there were X pages in the log, there were $10 \cdot X$ entries in the log, since there are 10 objects per page.

We did not charge the server for cache lookups. Since we are testing the configurations in an I/O bound system, performing the charge would not affect the comparison between the policies. Cache lookups were performed by the server after a fetch request, and the number of fetch requests was independent of the update policy. For the MBatch and Read-Modify-Write policies, a cache lookup was performed before issuing an iread. In the Opportunistic policy, the lookup was performed when a new modification arrived. Similarly, we did not charge the CPU for log activities either. The log was accessed after every write and every modification. Therefore, there should be the same amount of log activity for each policy, so this would not affect the final outcome as well. Both of these time charges would be minor when compared with the other charges on the server, so not charging for them would not

affect the final outcome.

The two trigger values listed in the table are for the Opportunistic policy only. Since it issues ireads from the log and writes from the cache, one trigger value is needed for each set. The iread trigger determines the number of objects awaiting ireads which will be allowed to accumulate in the log before an iread is issued. The write trigger determines the number of dirty pages in the cache before a write is issued. The MBatch and Read-Modify-Write both issue the write for a page after issuing an iread for it. All of the scheduling comes from the log, so no dirty page set is needed. Since we have set the trigger values for the Opportunistic policy, we will calculate the write absorption (see chapter 4) for each configuration, and then set the iread trigger value for the MBatch and Read-Modify-Write policies.

When a transaction arrives at the primary server, we charged it for running the validation algorithm. We assumed that no aborts would occur, so we only charged for the time required to run the algorithm. The effects of an abort would be the same as a read-only commit in the way it was treated by the simulator. The client sends a commit request to the server, the server immediately replies, and the client starts a new transaction.

A server machine will be dedicated to running the Thor database system. Therefore, we chose the higher value of 50 MIPS for the present server processor speed. Again, we used a projection from [8] to determine the processor speed for 5 years in the future.

5.1.4 Disk

Since we are studying I/O bound systems, we needed to model the disk accurately in order to see the full benefits from using each update policy. The server kept track of the disk head position after the last operation on the disk queue. The STF scheduling algorithm took this into consideration when choosing the next iread or write to be issued. Storing the disk head position also allowed us to accurately determine the time required by each disk operation.

The size and speed of the disk were taken from the disk simulated in [15]. For the

Parameter	Present Value	Future Value
Number of Cylinders	3,300	3,300
Number of Tracks/Cylinder	10	10
Number of Sectors/Track	10	10
Disk Rotation Speed	5400 RPM	9520 RPM
One Track Seek Time	2.5 msec	1.875 msec
Full Disk Seek Time	20.9 msec	15.675 msec
Disk Setup Time	2,500 Instructions	2,500 Instructions
Batch Size, MBatch policy	20	20

Table 5.4: Disk Parameters

future parameters, we did not change the size of the disk. Our reasoning for this is similar to our reasons for not changing the size of the database between trials. By not changing the disk size, we will isolate the benefits we see from using faster hardware without having to consider the effects of changing the disk size.

To generate the metrics for seek time, we used the simulator. We simply used the procedure for calculating seek time to obtain the one-track and full disk seek times. To determine the future parameters, we used projections from [8] on the progression of disk technology. Their studies show that every ten years, the disk seek rate doubles, implying the time to perform the seek is halved. Katz, et. al. assumed a linear progression in the improvement of the disk seek rate, so we should see a 25% percent improvement in seek time after five years. Therefore, we multiplied our seek times by a factor of .75 to obtain the future values. Borrowing a projection from [6], we assumed that rotational speed improved by 12% per year. We compounded this over 5 years to determine the future value for rotational speed.

For the MBatch policy, we chose a batch size of 20 ireads or writes to be issued at one time. This will ensure that fetch reads will not have to wait too long before they can be processed. It will also reduce the effects on the overall system performance

that the first write incurring a longer cost will have.

5.1.5 Network

Parameter	Present Value	Future Value
Network Latency	1 msec	500 usec
Network Setup Time	2,500 Instructions	2,500 Instructions

Table 5.5: Network Parameters

Since we are interested in disk bound systems, it is less important to completely and accurately model the network. Therefore, we decided to simulate it as a simple time charge. The effects of variable network delay, packet collisions, network contention, and lost packets would not affect the final outcome of the simulator. In fact, it would slow down the server, which would lessen the effects of the disk bound.

We assume that there will be no network contention between the two server, so we only need to model the latency. Additionally, the network is used the same amount by all policies. As with the cache and log, we could dismiss the charges, but the network latency is significantly higher than the time required for a cache lookup or a log allocation. Also, not charging for the network could lead to timing problems within the simulation if messages arrived instantly at their destinations.

Again, we borrowed a projection from [6] concerning the reduction of network latency. For his tests, he predicted that network latency would be reduced by a factor of two every five years. Therefore, we used this exact projection for our simulation trials.

5.2 Simulator Implementation

This section will provide some of the implementation details of our event-driven simulator of Thor. We will describe how the clients generate the workload for the servers

and how the server process the transactions. We will also discuss how each update policy handles the modifications, and how the servers manage the disk.

5.2.1 Transactions

All transactions begin at the client. First, the client must select which page to use. It does this by a random selection, each page having an equal probability of being chosen. After selecting the page, the client must determine whether or not it will modify the page. Again, this is done by random choice. We assign the percentage of read-only transactions so that the simulator will have that percentage chance of selecting a read-only commit. Finally, the client must determine whether or not it needs to issue a fetch request for the page. The probability that a page is present in the client's cache is simply determined by the client cache hit rate, not the client's history of page selection. Again, we determine if the page is present in the cache by random selection. If the page is present in the cache, then the client sends the commit request to the primary server.

If a fetch request is required, the client sends it to the appropriate server. With the basic fetch policy, all fetch requests are directed to the primary server. In the dual-fetch case, fetch requests for even pages go to the primary, while the requests for odd pages go to the backup. The server will then check its cache to see if the page is present. Unlike the clients, the server cache is fully modeled. If the page is present, the server will immediately return it to the client. Otherwise, the server will issue a disk read for the page. Once the read is complete, the page will be returned to the client. When the client receives the page from the server, it will proceed with the commit request, just as in the case where no fetch request was required.

All commit requests are directed to the primary server. Upon receipt of a commit request, the primary validates the request. As mentioned above, we assume that there are no aborted transactions in our system, so we just charge the time required to run the algorithm. If the commit request is a read-only commit, the primary simply replies to the client. For a write commit, the primary passes the modification to the backup server before replying to the client. Once the client receives the reply from

the server, it begins a new transaction.

The update policy determines how each server proceeded after the commit. Each server recorded the modification in the log, regardless of update policy. The MBatch and Read-Modify-Write schemes incremented the server's iread trigger if the commit requests did not overwrite a modification already waiting for an iread. We assumed a uniform access pattern for each object in the page; i.e. they were modified in turn. The only time an object in the log could be overwritten was if every object in the page had already been modified. The Opportunistic scheme performed a cache lookup after the modification was inserted into the log. If the page was present in the cache, then the modification was installed immediately, making the page dirty. If it was absent, then the Opportunistic policy used the same algorithm as the MBatch policy did.

In the dual-fetch server configuration, each server only performed ireads for the pages it owned. For the MBatch and Read-Modify-Write policies, the server sent the page to the other server after completing the iread for it. The pages from the other server would then be written to disk immediately after the modifications were installed. With the Opportunistic policy, when a modification was made to an odd page, the primary server would request the page from the backup. If the page was present in the backup server's cache, the backup would return it to the primary, which would install the modification. If the page were absent from the backup's cache, the page would be sent to the primary after the backup issued an iread for it.

5.2.2 Disk Scheduling

We modeled the disk as two sets of arrays. One of the arrays held the dirty pages, while the other held the pages waiting for ireads. Each element in the array was a linked list the held all of the pages in the cylinder which were awaiting ireads or writes. These arrays made the scheduling algorithms run much faster, since they could begin the search for the closest page from the current disk cylinder. The search stopped once the seek time between two cylinders became greater than the travel time between the current disk position and the closest page.

All disk operations are placed at the end of the disk queue. We used a FIFO disk queue for processing the operations, with no priority given to any operations over any other.

After a fetch read or iread is complete, the page is placed into the server's cache. For fetch reads, the page is then returned to the client. For ireads, every modification in the log waiting for the iread for that page is installed, making the page dirty. After a write is complete, space will be freed from the log only if that page has been written to disk at both servers. In our simulation, the two servers compared their log entry for that page. If both servers have written any of the objects to disk, those objects are removed from the log. In the actual implementation of Thor, this information is propagated asynchronously between servers; however, for simplicity reasons, we chose not to implement this, as it would complicate the simulator without adding accuracy to the overall system behavior.

5.3 Simulator Statistics

The simulator kept and updated statistics about each trial, allowing us to further analyze the results. The length of each run was 70,000 disk writes, with the simulator printing out statistics every 10,000 writes. These output bursts were stored in a file so that they could be examined later. Figure 5-1 shows an example printout obtained from one trial run.

We used the number of transactions per second that each configuration could process as the measure of effectiveness. The throughput value listed in the statistical summary is the throughput achieved by the system only during that period of 10,000 writes. After the printout, the value was reset. At the end of the simulation run, a program scans the file to determine the overall throughput for that trial. This result is then graphed against the throughputs of the other disk update policies, with the number of clients on the x-axis and throughput on the y-axis.

The foreign pages in the cache is only relevant for server configurations that use the cache-splitting algorithm. It shows how many pages in that server's cache are

```
84 events in queue
0 fetch across
        25824 commits in 50047.965 msecs is 516.0 transactions/second
  server 0:
        Log size : 2028, 2 entries on disk locally
        40 segs dirty, 40 writes outstanding
        20 foreign pages in cache
        1 reads outstanding, 0 ireads outstanding, 995 delayed installs
        5000 disk writes, 3.3% absorbed, 99.9 writes/second
        549 disk reads, 11.0 reads/second
        2035 ireads, 2.5% iabsorbed, 40.7 ireads/second
        avg disk costs:   read=16.8ms, iread=4.8ms, write=6.0ms,
                          first iread=8.7ms, first write=17.1ms
        18.30% cache hit ratio
        cpu usage: 67.1%, disk usage: 97.7%
  server 1:
        Log size : 2028, 2 entries on disk locally
        38 segs dirty, 38 writes outstanding
        20 foreign pages in cache
        0 reads outstanding, 0 ireads outstanding, 996 delayed installs
        5000 disk writes, 3.3% absorbed, 99.9 writes/second
        518 disk reads, 10.4 reads/second
        2165 ireads, 2.4% iabsorbed, 43.3 ireads/second
        avg disk costs:   read=16.2ms, iread=4.5ms, write=6.0ms,
                          first iread=6.9ms, first write=15.9ms
       20.06% cache hit ratio
      cpu usage: 10.3%, disk usage: 95.7%
   1320 fetchs, avg 119.943 msec, stddev 87.3 msec, max 441.3 msec
   25824 commits, avg 5.450 msec, c_stddev 2.0 msec
```

Figure 5-1: A sample simulator statistics printout

owned by the other server. In figure 5-1, the servers are using the MBatch policy, so the 20 foreign pages in the cache at each server correspond to the 20 pages received from the other server after an iread batch was complete.

We can calculate the disk queue length by adding the total number of writes, iread, and reads that are outstanding. The first iread and write times are only used with the MBatch policy. They show the cost of the first operation in the iread and write batches. The time for the first write is significantly higher than the average opportunistic access time, since the server can process fetch reads between the completion of the iread batch and the initialization of the write batch. This moves the disk head position while the server installs the modifications. Therefore, the first write becomes a random disk access, just like a fetch.

Another important statistic is the write and iread absorption percentages. They tell us the percentage of writes and ireads that were saved because we delayed them, i.e. the percentage of modifications that went to pages already in the iread or write set. These modifications did not require a separate iread or write. The simulator calculated the percentage by the equation:

$$\frac{modifications\_written - total\_writes}{modifications\_written} \tag{5.1}$$

For example, if we state that 150 modified objects were written to disk in only 100 writes. This would imply 1.5 modifications/write, and, by our formula, an absorption of 33%; one third of the objects that were written to disk did not require a separate write.

Also included in the simulation statistics is the utilization percentage for both the cpu and the disk. One way to compare the policies is to check the disk utilization. If it is near 100%, then the policy would not be able to handle a faster transaction arrival rate. A lower disk utilization shows that the system has not reached its maximum throughput yet. If two policies had the same throughput, but one had 100% disk utilization and the other didn't, then the one with lower utilization would be better for that configuration, since the disk could handle more clients before becoming

saturated.

5.4 Parameter Derivation

Now that we have presented the overall system parameters, we can derive some of the others based on the equations from chapter 4. We need to determine the amount of log space required by the servers, which in turn will tell us how much memory we will have for the cache. From this, we can predict the server cache hit rate that we can compare with the one observed in the simulator. We also need to compute the write absorption for the Opportunistic policy, so that we can then determine the value of the iread trigger in the Read-Modify-Write and MBatch polices. We will select the trigger value so that the write absorption is equal for all three policies.

5.4.1 Memory Partition

We will determine the log size by using the iread and write trigger of the Opportunistic policy. However, we need to determine the write absorption that we will see before we compute the log size, since we need to determine the expected number of modifications that will be serviced by each write. However, we need to determine the cache size before we can calculate the write absorption, since it is affected by the server cache hit rate. Therefore, we will assume that there will be a maximum of 1.5 modifications per write, which, as we will show, is a fair maximum value. This will give us:

$$\begin{aligned} num\_dirty\_modifications &= 2 \cdot write\_trigger \cdot 1.5 \\ &= 2 \cdot 4500 \cdot 1.5 \\ &= 13,500 \end{aligned} \tag{5.2}$$

modifications that have been installed but are waiting for writes. Including the modifications that are waiting for ireads, this gives us a log size of:

$$log\_size > 2 \cdot iread\_trigger + num\_dirty\_modifications$$

$$\begin{aligned} log\_size &> 2 \cdot 4500 \cdot 13,500 \\ log\_size &> 22,500 objects \end{aligned} \tag{5.3}$$

To account for any objects involved in outstanding pages writes and ireads, we will set the log size at:

$$log\_size = 25,000 objects = 2,500 pages \tag{5.4}$$

Using the large memory size, this makes the cache size:

$$cache\_size = 27,500 pages \tag{5.5}$$

while with the small memory size we get:

$$cache\_size = 7,500 pages \tag{5.6}$$

Memory Size	Cache Splitting?	Cache Hit Rate
Large	No	8.33%
Large	Yes	15.30% / 16.67%
Small	No	2.27%
Small	Yes	3.18% / 4.55%

Table 5.6: Server Cache Hit Rate - Theoretical Values

Table 5.6 shows the server cache hit rate that we should expect to see with these cache sizes under each fetch policy. With the dual-fetch case, two values are listed for the cache hit rate. The first value is the cache hit rate for the Opportunistic policy, which, because of the dirty pages being cached at both servers, has a lower cache hit rate in this case. The second value is the server cache hit rate for the MBatch and

Read-Modify-Write policies.

5.4.2 Write Absorption - Opportunistic Policy

Now that we know the server cache hit rate, we can compute the probability that an incoming modification needs an iread. With this and the iread trigger value, we can compute the expected number of modifications that will occur before the iread is issued. We can then determine the expected iread absorption.

For write absorption, we can now compute the probability that a modification will be absorbed in the dirty page set, since we now know the cache size and the write_trigger. With the probability that a modification hits in the log, we can compute the expected number of modifications that will occur before the write for a page is issued. Factoring in the number of modifications that will occur before an iread for a page is issued, we can compute the expected number of modifications that will be written out by each disk write. The results for each cache size, fetch policy, and fetch probability are shown in table 5.7.

Memory Size	Cache Splitting?	Fetch Prob.	Iread Absorption	Write Absorption
Large	No	.05	1.0157	1.0276
Large	No	.20	1.0186	1.0279
Large	Yes	.05	1.0340	1.0415
Large	Yes	.20	1.0404	1.0418
Small	No	.05	1.0170	1.0279
Small	No	.20	1.0175	1.0293
Small	Yes	.05	1.0297	1.0417
Small	Yes	.20	1.0353	1.0432

Table 5.7: Write and Iread Absorption for the Opportunistic policy

5.4.3 Iread Trigger Value, MBatch and Read-Modify-Write

Now that we have computed the write absorption for the Opportunistic policy with the system parameters, we can compute the iread trigger value for the MBatch and Read-Modify-Write policies. In section 4.1.3, we presented the method for computing the write absorption given the iread trigger value. Now that we have the absorption for each configuration, we can backsolve the equations, as shown in section 4.1.6, to compute the iread trigger value for that configuration. These results are presented in table 5.8.

Memory Size	Cache Splitting?	Fetch Prob.	Iread Trigger Value
Large	No	.05	9085
Large	No	.20	9185
Large	Yes	.05	6820
Large	Yes	.20	6870
Small	No	.05	9185
Small	No	.20	9640
Small	Yes	.05	6855
Small	Yes	.20	7100

Table 5.8: Iread Trigger Value - MBatch and Read-Modify-Write

5.5 Maximum Throughput

Now that we have computed the expected server cache hit rate and write absorption for each configuration, we can now determine the expected maximum throughput for each policy. We will use the equations presented in section 4.2 with the parameters calculated above in order to determine the throughput for each policy in each server configuration and parameter set.

5.5.1 Basic Assertions

This section will present the basic assertions that we will use for all of the calculations we will be doing in this section. For simplicity, we will use the same number of transactions for every situation. We will set this at:

$$T = 100,000 transactions \tag{5.7}$$

for our tests.

We know that $P_w = 20\%$ for every case, so we will also be holding the number of write-commits the same. This value will be:

$$write\_commits = T \cdot P_w = 100,000 \cdot .20 = 20,000 \tag{5.8}$$

There are only two different values for P_f, so we can compute the number of fetch requests in advance as well. With the light fetch load, $P_f = 5\%$, so the total number of fetch requests is:

$$fetch\_requests = T \cdot P_f = 100,000 \cdot .05 = 5,000 \tag{5.9}$$

In the heavy fetch load, $P_f = 20\%$, giving us:

$$fetch\_requests = T \cdot P_f = 100,000 \cdot .20 = 20,000 \tag{5.10}$$

fetch requests. Note that this is not the number of fetch reads that will occur. A fetch read only results from a cache miss on a fetch request, so we can represent the number of fetch reads as:

$$fetch\_reads = (1 - \lambda) \cdot fetch\_requets \tag{5.11}$$

where λ is determined for each configuration.

5.5.2 Basic Fetch Policy

This section will present all of the throughput calculation for the basic fetch configuration. We will begin each analysis by presenting the parameter values that are specific to each test, i.e. cache hit rate and absorption values. Then we will proceed to calculate the maximum steady state throughput for each policy.

Large Cache, Heavy Fetch Load

For this test: $\lambda = 8.33\%$, $A_i = A_w = .973$, and for the Opportunistic policy, $A_i = .982$. Since we are using the heavy fetch load, we will have 20,000 fetch requests for this configuration.

The number of fetch reads we should see is going to be the same for all policies. Each policy has the same cache hit rate, so we there should be:

$$fetch\_reads = (1 - \lambda) \cdot 20,000 = 18,334 \tag{5.12}$$

fetch reads in the system.

The number of writes will also be the same, since that is based on write absorption, which is the same for each policy. Inserting the expected writes per modification gives us:

$$num\_writes = 20,000 \cdot A_w = 19,460 \tag{5.13}$$

disk writes for this configuration.

However, the number of ireads will be different for the Opportunistic policy because of the difference in iread absorption. For the MBatch and Read-Modify-Write policies, we use the write absorption value, giving us:

$$num\_ireads = (1 - \lambda) \cdot 20,000 \cdot (1 - P_f) \cdot A_w = 14,272 \tag{5.14}$$

ireads. For the Opportunistic policy, we substitute the value for A_i, giving us:

$$num\_ireads = (1 - \lambda) \cdot 20,000 \cdot (1 - P_f) \cdot A_i = 14,404 \tag{5.15}$$

ireads.

Now we will compute the actual throughput for each update policy, beginning with the Opportunistic policy. For the Opportunistic policy, the average iread and write will take 3.9msec. This should give us a total disk time of:

$$disk\_time = 3.9 \cdot (14,404 + 19,460) + 16.73 \cdot 18,334 = 438.8 seconds \quad (5.16)$$

This leads us to an expected throughput of:

$$throughput = \frac{100,000}{438.8} = 227.9 transactions/second \quad (5.17)$$

With the future parameters, we should see:

$$disk\_time = 3.4 \cdot (14,404 + 19,460) + 11.54 \cdot 18,304 = 326.7 seconds \quad (5.18)$$

$$throughput = \frac{100,000}{326.7} = 306.1 transactions/second \quad (5.19)$$

For the MBatch policy, we must determine how many writes will require a random access time and how many will be opportunistic accesses. The number of random writes will be:

$$first\_writes = \frac{19,460}{20} = 973 \quad (5.20)$$

This leaves us with:

$$regular\_writes = 19,460 - 973 = 18,487 \quad (5.21)$$

regular writes. For our computations, we will use an Opportunistic access time of 3.1msec. This gives us a disk time of:

$$disk\_time = 3.1 \cdot (14,272 + 18,847) + 16.73 \cdot (18,334 + 973) = 425.7 seconds \quad (5.22)$$

and an expected throughput of:

$$throughput = \frac{100,000}{425.7} = 234.9 transactions/second \tag{5.23}$$

With the future parameters, we should see:

$$\begin{aligned} disk\_time &= 2.5 \cdot (14,272 + 18,847) + 11.54 \cdot (18,334 + 973) \\ &= 305.6 seconds \qquad (5.24) \\ throughput &= \frac{100,000}{305.6} = 327.2 transactions/second \qquad (5.25) \end{aligned}$$

Similarly, for the Read-Modify-Write policy, we must divide the writes into the different categories presented in section 4.2. The number of rmw_writes that take the time required to spin the disk head is equal to the number of ireads, which was already computed above. Given this, the number of writes that will require a random access charge will be:

$$short\_writes = 19,460 - 14,272 = 5,188 \tag{5.26}$$

This gives a total disk time of:

$$\begin{aligned} disk\_time &= 9.9 \cdot 14,272 + 16.73 \cdot (18,304 + 14,272 + 5,188) \\ &= 773.1 seconds \qquad (5.27) \end{aligned}$$

which gives us an expected throughput of:

$$throughput = \frac{100,000}{773.1} = 129.3 transactions/second \tag{5.28}$$

With the future parameters, we should see:

$$\begin{aligned} disk\_time &= 5.7 \cdot 14,272 + 11.54 \cdot (18,304 + 14,272 + 5,188) \\ &= 517.1 seconds \qquad (5.29) \end{aligned}$$

$$throughput = \frac{100,000}{517.1} = 193.3 transactions/second \quad (5.30)$$

Other Configurations

Memory Size	p f	Opportunistic Tput	MBatch Tput	R-M-W Tput
Large	.05	455.6 / 563.7	493.1 / 654.8	175.6 / 267.5
Large	.20	227.9 / 306.1	234.9 / 327.2	129.3 / 193.3
Small	.05	437.4 / 542.2	473.1 / 628.9	170.7 / 260.6
Small	.20	216.3 / 291.1	227.1 / 318.7	124.7 / 186.5

Table 5.9: Expected Throughput Values for Different Configurations

Now that we have given a detailed example of computing the expected throughput, we will just give the results for the other parameter combinations in table 5.9. The computation process will be the same for each case, only the parameter values will be different. In the table, the values are listed present throughput / future throughput. These are only the cases for the basic fetch policy; the dual-fetch case will be handled below.

For these tests, we simply took values from the tables printed earlier in the chapter. The values for λ were obtained from table 5.6 for the appropriate cache size and access. The values for A_i and A_w were computed by using the absorption figures from table 5.7.

5.5.3 Dual Fetch Policy

This section will mirror the previous one. It will begin by presenting a detailed example of how we computed the expected throughput for one configuration operating under the dual-fetch scheme. Then, we will present the expected throughput for the other parameter sets in a table.

Large Cache, Heavy Fetch Load

We will use the same configuration here that we used in the previous section as our example. First, we will compute the throughput for the Opportunistic policy, and then we will update the parameters for computing the throughput for the MBatch and Read-Modify-Write policies.

For the Opportunistic policy, we should see $\lambda = 15.30\%$, $A_w = .960$, and $A_i = .967$. We need to have a different cache_hit_rate for the Opportunistic because of the effects of the dirty pages being at both servers will have on the cache hit rate.

Since we are testing the heavy fetch load, we will see 20,000 fetch requests to the servers. This will yield:

$$fetch\_reads = \frac{(1-\lambda) \cdot 20,000}{2} = 8,472 \tag{5.31}$$

fetch reads for the server to handle.

The number of ireads will be affected in a similar manner as the fetch reads. Therefore, we should expect:

$$num\_ireads = \frac{20,000 \cdot (1-\lambda) \cdot (1-P_f) \cdot A_i}{2} = 6,519 \tag{5.32}$$

ireads to occur at each server. The total number of writes will not be affected by the fetch policy, so we should have:

$$num\_writes = 20,000 \cdot A_w = 19,200 \tag{5.33}$$

disk writes in this configuration.

This gives us a total disk time of:

$$disk\_time = 3.9 \cdot (19,200 + 6,519) + 16.73 \cdot 8,472 = 242 seconds \tag{5.34}$$

This gives us an expected throughput of:

$$throughput = \frac{100,000}{242} = 413.2 transactions/second \tag{5.35}$$

Using the future parameters we get:

$$\begin{aligned} disk\_time &= 3.4 \cdot (19,200 + 6,519) + 11.54 \cdot 8,472 = 185.2 seconds & (5.36) \\ throughput &= \frac{100,000}{185.2} = 539.9 transactions/second & (5.37) \end{aligned}$$

For the MBatch and Read-Modify-Write policies, we need to adjust the parameters. The value for A_w will remain the same, but we will now use $\lambda = 16.67$ and $A_i = .960$ for our calculations.

First, we shall compute the number of fetch reads we expect to see. Given that we have 20,000 fetch requests, we should have:

$$fetch\_reads = \frac{(1-\lambda) \cdot 20,000}{2} = 8,333 \tag{5.38}$$

fetch reads in this configuration. We will also need to divide the ireads between the servers. This gives us:

$$num\_ireads = \frac{(1-\lambda) \cdot 20,000 \cdot (1-P_f) \cdot A_i}{2} = 6,400 \tag{5.39}$$

ireads for the system. The total number of writes will be the same as in the Opportunistic case, since all policies have the same write absorption. Now we need to divide the writes into the proper type for each update policy.

For the MBatch policy, we need to divide the writes into first writes and regular writes. Given the total number of writes, we should have:

$$\begin{aligned} first\_writes &= \frac{19,200}{20} = 960 & (5.40) \\ regular\_writes &= 19,200 - 960 = 18,240 & (5.41) \end{aligned}$$

writes of each type. This gives us a total disk time of:

$$disk\_time = 3.4 \cdot (18,240 + 6,400) + 16.73 \cdot (960 + 8,333) = 239.2seconds \quad (5.42)$$

giving us an expected throughput of:

$$throughput = \frac{100,000}{239.2} = 418transactions/second \quad (5.43)$$

Using the future parameters gives us:

$$\begin{aligned} disk\_time &= 3.0 \cdot (18,240 + 6,400) + 11.54 \cdot (960 + 8,333) \\ &= 181.2seconds \quad (5.44) \\ throughput &= \frac{100,000}{181.2} = 552transactions/second \quad (5.45) \end{aligned}$$

For the Read-Modify-Write, we need to divide the writes into the random writes and the rmw_writes. The number of rmw_writes that occur will be equal to the number of ireads, which was computed above. Therefore, the number of random_writes is simply the difference from the total number of writes.

$$random\_writes = 19,200 - 6,400 = 12,800 \quad (5.46)$$

These values give us a total disk time of:

$$\begin{aligned} disk\_time &= 9.9 \cdot 6,400 + 16.73 \cdot (12,800 + 6,400 + 8,333) \\ &= 523.7seconds \quad (5.47) \end{aligned}$$

With this amount of time, we should see an expected throughput of:

$$throughput = \frac{100,000}{523.7} = 190.9transactions/second \quad (5.48)$$

With the future parameters, we should see:

$$\begin{aligned} disk\_time &= 5.7 \cdot 6,400 + 11.54 \cdot (12,800 + 6,400 + 8,333) \\ &= 354.2 seconds \qquad (5.49) \\ throughput &= \frac{100,000}{354.2} = 282.3 transactions/second \qquad (5.50) \end{aligned}$$

Other Configurations

Memory Size	p f	Opportunistic Tput	MBatch Tput	R-M-W Tput
Large	.05	710.9 / 860.8	720.6 / 887.7	231.8 / 346.1
Large	.20	413.2 / 539.9	418.0 / 552.0	190.9 / 282.3
Small	.05	665.7 / 809.1	677.5 / 837.1	223.5 / 334.7
Small	.20	376.0 / 494.1	380.8 / 505.4	180.8 / 267.9

Table 5.10: Expected Throughput for several dual-fetch configurations

Table 5.10 shows the throughput values for all configurations when the cache-splitting algorithm is used for fetches and ireads. We list only the final throughput result for each configuration, as the method by which we obtain the value will be the same as above. The first value listed in each slot is the present throughput, while the second one shows the future expected throughput. As in the basic-fetch case, we obtained all of the values for λ from table 5.6, and the values for A_w and A_i from table 5.7.

Chapter 6

Simulation Results

This chapter will present all of the simulation results we obtained. For each configuration, we will include a graph showing the throughput each policy achieves, as well as a table providing further metrics that can be used to judge each policy's performance We will also analyze the results, comparing what we observe with what the mathematical model predicted.

In our simulation results, we will be concerned with the maximum throughput each policy can achieve. However, actual database systems are never configured to operate at maximum throughput constantly. In the saturated state, the response time for fetch and commit requests becomes larger. Therefore, the database systems are configured so that the normal transaction load is only about 60-70% of what it can process. This way, response time is good, and the database can handle a short burst in the load without sacrificing performance. Despite this, studying the policies in the saturated state still provides us with information on when the policies enter that state. In addition, with a smaller number of clients, we can see how each policy performs when the workload is not as high.

We have divided the analysis by server configuration. In chapter 3, we detailed the advantages of each configuration when we presented them. Therefore, we will not address those issues in our analysis. Since the mathematical model predicted that the client cache hit rate would affect the throughput more than the server cache size, we divided each configuration's analysis into the normal and heavy fetch loads. With

each client cache hit rate, we will provide the simulation results of the large and small cache sizes.

In order to facilitate the presentation of the results, we will define several different operation states that the server could be in with each policy. These states are:

Sub-Saturation State In this state, the disk is not operating at full utilization, so the policy can process a higher transaction arrival rate.

Saturated State This state is when the policy has achieved its maximum throughput, since the disk is fully utilized.

Full-Log State In this state, the log is full. Since the log is full, there will be higher absorption than in the steady state, since modifications will use a larger portion of the log than they would in the steady state. For the MBatch policy, ireads and writes will take less time, since there will be more pages to choose from. However, since the log is full, clients must wait until log space is available before their commit requests can be processed. This results in the policy achieving less than maximum throughput.

We will also define a state that the simulation can be in for a trial.

Non-Steady State A policy is in this state when modifications are arriving at a faster rate than they are being cleaned from the log. We will see the same benefits we did from the full log state, but the log is not full, so commit requests would be processed normally. Therefore, the policy will achieve higher throughput while in this state when compared with the steady state.

We will ignore the results in which the policy is in the full-log state. Since clients will have to wait a long time for their requests to commit, the throughput will be lower and the policy would be undesirable in that state. We will also ignore the results from the non-steady state, since the throughput we observe will be an inflated value of what the policy could actually achieve.

6.1 Basic Configuration

6.1.1 Normal Fetch Load

Large Cache

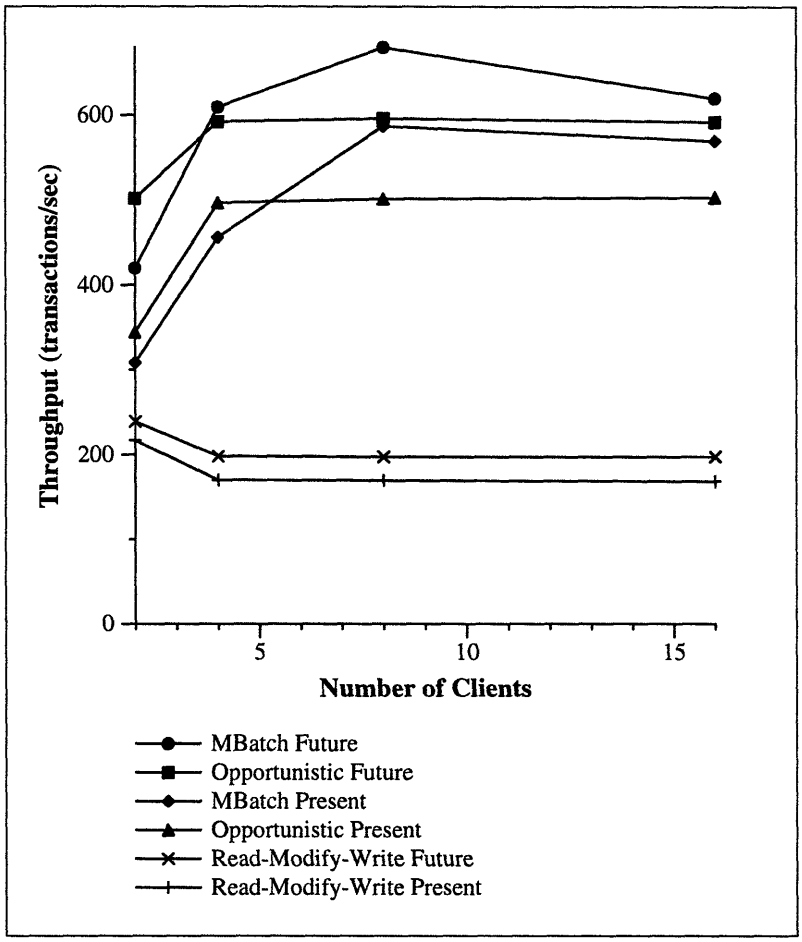

Figure 6-1: Large Cache, Basic Configuration, Normal Fetch Load

Figure 6-1 presents the graph of the results for this configuration. Table 6.1 provides further information with which to study the results. The graph shows that the MBatch policy achieves the highest throughput of the policies in this configuration. However, with eight clients, the MBatch policy is in the non-steady state for most of the trial and finally reaches the full-log state at the end. With sixteen clients, the MBatch policy again starts in the non-steady state and reaches the full-log state earlier in the trial, explaining the drop in throughput we observe. Therefore, we will

	Opportunistic	MBatch	Read-Modify-Write
Commit Latency (msec)	5.33	4.89	13.4
Fetch Latency (msec)	47.0	66.0	38.2
Log Space Used (objects)	12,998	12,846	25,000

Table 6.1: Further Results. Metrics are for 4 clients, present parameters

ignore the results of the MBatch policy with eight and sixteen clients.

There is also a drop in throughput for the Read-Modify-Write policy. With two clients, the policy is in the non-steady state, raising the observed throughput. With four or more clients, the system remains in the full-log state throughout the trial. Even in the non-steady state, the Read-Modify-Write policy can not process transactions as fast as the other two policies. This shows that we benefit greatly from scheduling the ireads and writes instead of issuing them in a FIFO manner.

In comparing the MBatch and Opportunistic policies, we will only be concerned with two and four clients, since those are the cases when the MBatch policy is in the sub-saturation state. In both cases, the Opportunistic policy achieves slightly higher throughput than the MBatch policy. We can attribute this to the improved fetch response time of the Opportunistic policy. As shown in table 6.1, the fetch latency for the Opportunistic policy is about 30% less than that of the MBatch scheme. Fetch latency includes the time a fetch read waits in the queue as well as the time required to read the page. Since the fetch reads must wait behind a batch of ireads or writes for the MBatch policy, the waiting time in the disk queue will be longer, resulting in higher fetch latency. While the client is waiting for a fetch, it can't generate any commit requests for the system, so the longer fetch latency leads to lower throughput. Since there is a light fetch load in this system, the improved fetch latency of the Opportunistic policy only leads to a slight throughput increase over the MBatch scheme.

At four clients, both policies are in the saturated state. This is seen in the Opportunistic graph by the fact that throughput does not increase when we add more clients. Therefore, by the simulation results, we should conclude that the Opportunistic policy has the best throughput in this configuration when we look at the saturation state of each policy.

When we compare the simulation results with the mathematical predictions from table 5.9, we see very different results from what we predicted. For the Read-Modify-Write policy, this can be explained by the fact that it was operating in the non-steady state throughout. Our model did not cover this state, and the policy achieves higher throughput than expected in this state anyway.

With the MBatch policy, we see slightly lower throughput at four clients, the saturation point, than what we predicted in the model. Part of this can be explained by the longer fetch latency. With the higher fetch latency, clients are spending more time not issuing commit requests. This was not taken into consideration in the mathematical model, but does affect the real system.

In the Opportunistic policy, we see higher throughput than we predicted in the model. Part of this comes from us including the initial state of the simulation trial in our results. In the beginning, there are no writes or ireads on the disk queue, so the system will be able to process transactions at a fast rate, since the fetch latency will be extremely low. The observed throughput was about 10% higher at the beginning than in the steady-state, increasing the observed throughput. We also see that the cost of the ireads and writes is slightly less than we predicted. This factor raised the steady state throughput, since it would take the disk less time to process the workload.

Small Cache

As predicted by the mathematical model, we only see a slight drop in throughput when compared with the large cache version of this configuration. The throughput reduction is mainly due to the higher fetch latency observed in the simulator (see table 6.2). Because of the smaller cache size, fewer fetch requests will hit in the

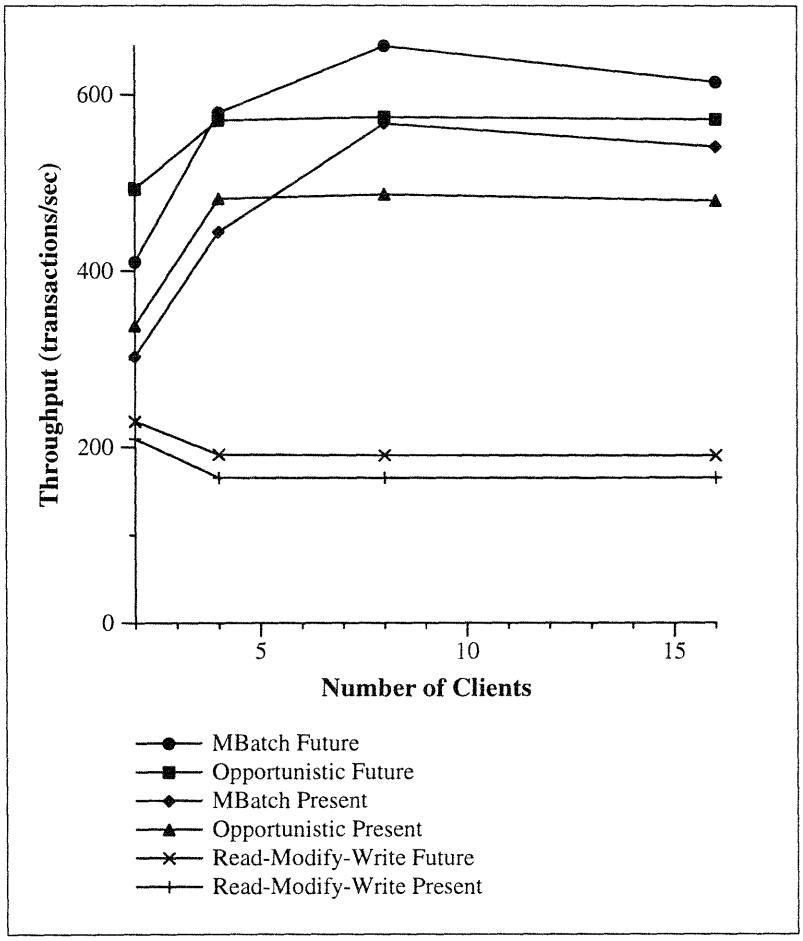

Figure 6-2: Small Cache, Basic Configuration, Normal Fetch Load

	Opportunistic	MBatch	Read-Modify-Write
Commit Latency (msec)	5.28	4.88	13.7
Fetch Latency (msec)	52.7	70.1	41.8
Log Space Used (objects)	13,416	13,372	25,000

Table 6.2: Further Results. Metrics are for 4 clients, present parameters

cache. This increases the average fetch latency since there are fewer fetch requests that will be quickly returned, taking much less time than those that need a disk read. However, the client cache hit rate is high, meaning that a small increase in fetch latency will only slightly decrease the throughput, since there are few fetch requests in this workload.

We also see more ireads than we did with the large cache. Since there are fewer pages in the cache, more of the modifications will require ireads before they can be written to disk. However, since ireads take less time than fetch reads in the Opportunistic and MBatch scheme, the increase in the number of ireads will have minimal effect on the throughput.

When comparing the simulation results with the mathematical model, we see the same occurrences as we did with the large cache version. The Read-Modify-Write policy achieves higher throughput, but it is the non-steady state, so its results are invalid. The Opportunistic policy achieves slightly higher throughput than predicted, from the same reasons we saw earlier. The MBatch policy achieves slightly lower throughput than predicted due to the increased fetch latency.

6.1.2 Heavy Fetch Load

Large Cache

	Opportunistic	MBatch	Read-Modify-Write
Commit Latency (msec)	4.73	4.67	11.8
Fetch Latency (msec)	57.4	64.9	55.4
Log Space Used (objects)	15,744	14,073	25,000

Table 6.3: Further Results. Metrics are for 4 clients, present parameters

Figure 6-3 presents the graph of the results for this configuration. Table 6.3 provides further information with which to study the results. We see in table 6.3 that

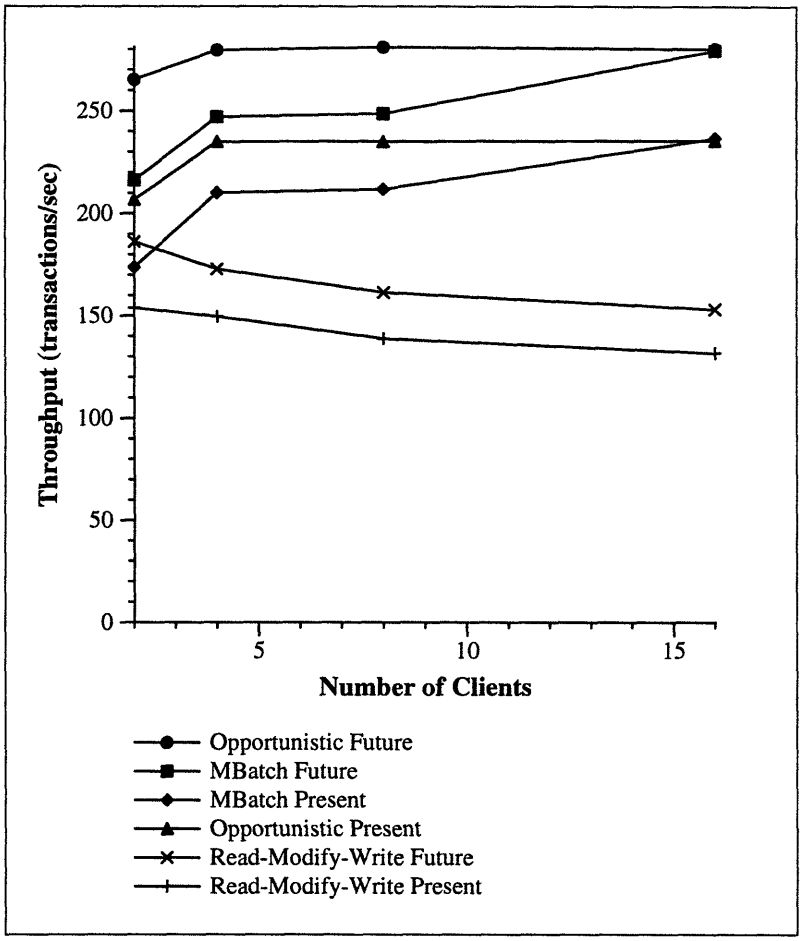

Figure 6-3: Large Cache, Basic Configuration, Heavy Fetch Load

the Read-Modify-Write policy is operating the in full-log state at four clients, just as in the normal fetch load case. At two clients, the system is in the non-steady state, so we see higher throughput than the maximum steady-state throughput for the policy. Therefore, we will not thoroughly study the Read-Modify-Write results. Again, we observe that scheduling the ireads and writes provides higher throughput than the FIFO policy, even when the FIFO policy is running in a state in which it is achieving higher throughput.

The MBatch policy does not enter the non-steady or full-log state until sixteen clients in this configuration. With the increased fetch load, commit requests are arriving at a slower rate. Therefore, more clients are required before modifications enter the log at a faster rate than they can be removed. We will only study the performance of the MBatch policy for less than sixteen clients.

For less than sixteen clients, we observe that the Opportunistic policy has higher throughput than the MBatch policy. This is due to the lower fetch latency, although the difference in latency is lower in this case than in the normal fetch load. We see higher fetch latency for the Opportunistic policy, due to the fact that there are more fetch requests. However, with the MBatch policy, we see approximately the same fetch latency. This can be explained by the fact that the fetch requests will have to wait approximately the same time before they can be processed; one batch length of ireads or writes. Also, since modifications are arriving at a slower rate, batches will be issued at a slower rate. Therefore, more fetch requests will occur when there are no batches on the disk queue, meaning the fetch reads can be processed much quicker. These two factors effectively cancel each other out, so we see approximately the same fetch latency as we did earlier.

The Opportunistic was able to achieve lower fetch latency, so it was able to achieve the highest throughput in this configuration. Since we are in the heavy fetch load, the fetch latency is an important performance metric of each policy. We also see that the difference in throughput between the Opportunistic and MBatch policies is larger here than in the normal fetch load. Since there are more fetch requests, the fetch latency becomes more of a factor in determining the throughput, giving the

Opportunisitc policy a greater advantage.

We will now compare the results with the predicted throughput from table 5.9. For the Read-Modify-Write, our mathematical model does not apply, since the policy is in the non-steady state even at two clients.

For the Opportunistic policy, we see that the observed throughput is slightly higher than the predicted value. Even when we eliminate the initial transactions, which operate at a higher throughput since there are no ireads or writes in the system yet, from the throughput calculations, we still observe slightly higher throughput than predicted. Again, the iread and write cost in this case is lower than we predicted, which leads to the slight improvement in throughput, since the disk can process the workload quicker.

With the MBatch policy, we see lower throughput than predicted. This results from the longer fetch latency, which means clients must wait longer before their fetch requests are returned, so they will not be sending commit requests to the system. Since the clients must wait longer for the fetch requests to be returned, they will not be sending commit requests to the server at as fast a rate, leading to lower throughput than predicted.

Small Cache

	Opportunistic	MBatch	Read-Modify-Write
Commit Latency (msec)	4.70	4.62	12.5
Fetch Latency (msec)	62.5	67.8	61.7
Log Space Used (objects)	14,258	14,921	25,000

Table 6.4: Further Results. Metrics are for 4 clients, present parameters

Again, as predicted in the mathematical model, using the smaller cache in this configuration has very little effect on the outcome. However, we do see that there is a

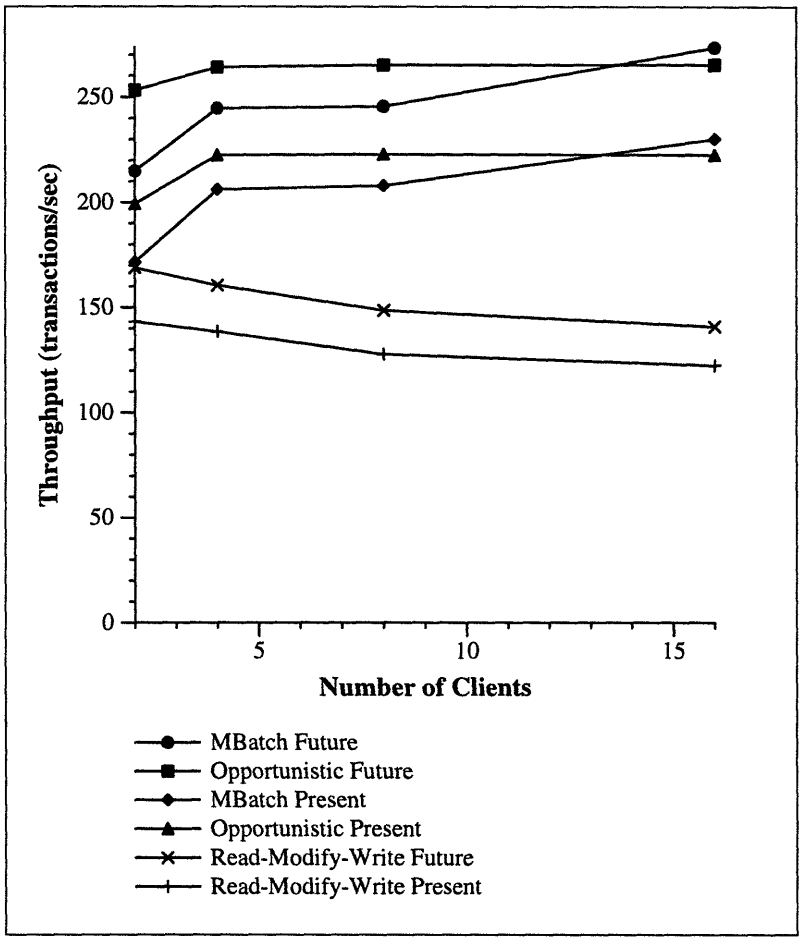

Figure 6-4: Small Cache, Basic Configuration, Heavy Fetch Load

larger throughput difference between the different cache sizes here than in the normal fetch load case. Since we are increasing the fetch load, we are increasing the number of fetch reads in the system by a greater amount than we did with the normal fetch load, leading to a larger difference in throughput between the two cache sizes.

There will also be more ireads in this configuration than in the large cache version. However, since ireads take less time for the MBatch and Opportunistic policy, increasing the number that occur will have less of an effect on the system. The increase in the number of ireads should be smaller here than in the small cache, normal fetch load configuration. An iread will not occur for a modification that is preceeded by a fetch. Since there is a higher percentage of fetch requests, we will not see as many more ireads as we did in the normal fetch case when we switch to the smaller cache size.

6.2 Dual-Fetch Configuration

6.2.1 Normal Fetch Load

Large Cache

	Opportunistic	MBatch	Read-Modify-Write
Commit Latency (msec)	5.42	5.04	9.97
Fetch Latency (msec)	35.7	50.9	41.8
Log Space Used (objects)	14,643	9,168	25,000

Table 6.5: Further Results. Metrics are for 4 clients, present parameters

As with the basic configuration, the Read-Modify-Write policy is operating in the non-steady state even at two clients. Increasing the number of clients will only force the policy into the full-log state throughout the simulation trial. Therefore,

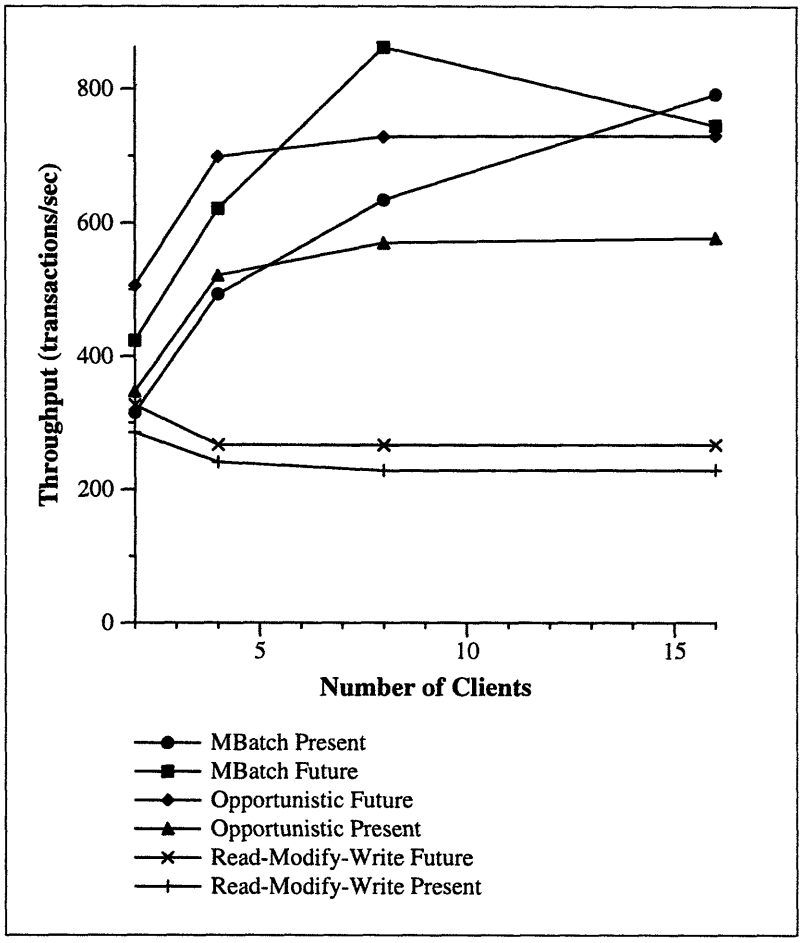

Figure 6-5: Large Cache, Dual-Fetch Configuration, Normal Fetch Load

we will not analyze the results for the Read-Modify-Write policy. At sixteen clients, the present system running the MBatch policy achieves higher throughput than the future system. In this case, the future system is in the full-log state, while the present system is in the non-steady state. Therefore, we will simply ignore the sixteen client data point when we examine the results.

With two and four clients, we see that the Opportunistic policy has slightly higher throughput than the MBatch policy. In both cases, the fetch latency for the Opportunistic policy is less than what we observe in the MBatch case. Even though there will be the fewest fetch reads in this case, improved fetch response time is still a factor. Since we are in the dual-fetch case, the server cache hit rate for the Opportunistic policy is less than that of the MBatch policy. As a result, more fetch reads and ireads will occur with the Opportunistic policy. Despite this, we still see higher throughput in the Opportunistic case, since the fetch reads can be processed faster than with the MBatch policy.

At eight clients, the MBatch policy achieves higher throughput than the Opportunistic policy. Even though the policy was a higher fetch latency, it still achieves higher throughput. However, when we examine the disk at four clients, we notice that the Opportunistic policy is operating near the saturated state, while the MBatch policy is in the sub-saturation state. Therefore, it is not surprising that the MBatch policy would be able to process more transactions at eight clients, since the Opportunistic policy could only handle a small workload increase before entering the saturated state.

However, the saturation state for the Opportunistic policy occurs at a much earlier point than we predict (see table 5.10) The reason for this is the average write cost. For some reason, the average write cost in this case is almost twice what we predicted it should be. This leads to the higher disk utilization that we observed at four clients. Since the writes require more disk time, the system can't process transactions as fast as we predicted, since the disk reaches the saturation point earlier. Unfortunately, we were unable to determine an explanation for this phenomenon; there was not enough time to fully investigate it.

We also notice that the MBatch policy achieves lower throughput than we predicted. Part of this can be attributed to the higher fetch latency, since if clients are waiting longer for their fetch requests to be returned, then it will take longer to send commit requests to the server. Also, the average iread and write cost is slightly higher than we predicted. Part of this is due to the absorption. Since we are in the dual-fetch case, we have higher number of modifications per iread. Because of this, there are fewer pages waiting for ireads. Since the set of pages, not objects, waiting for ireads is smaller than we predicted, the average cost of the operations will be slightly higher. Also, with eight clients, the disk is not quite fully utilized, so the MBatch policy could handle a slightly greater number of clients. Combining these factors, we see lower throughput than expected with the MBatch policy at eight clients.

Small Cache

	Opportunistic	MBatch	Read-Modify-Write
Commit Latency (msec)	5.55	5.01	10.1
Fetch Latency (msec)	26.0	55.5	48.9
Log Space Used (objects)	17,041	9,230	25,000

Table 6.6: Further Results. Metrics are for 4 clients, present parameters

For the small cache case, the MBatch and Read-Modify-Write policies are in the same states as they were in the large cache version. Again, using the small cache made little difference in throughput. However, the Opportunistic policy's throughput increases when compared with the large cache throughput. In this case, the throughput we observe for the Opportunistic policy is what we expect (see table 5.10). Unlike the large cache version, our write cost is what we expected it to be. Therefore, the policy was able to achieve the throughput we predicted.

As predicted in the mathematical model, we see a more significant reduction in

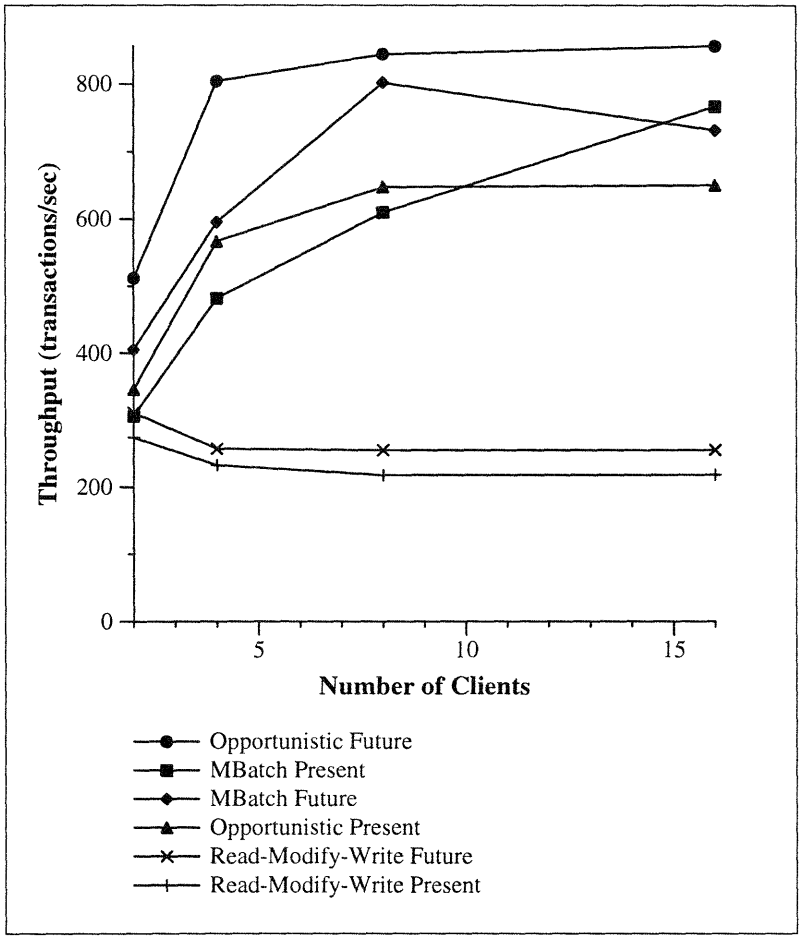

Figure 6-6: Small Cache, Dual-Fetch Configuration, Normal Fetch Load

throughput with the small cache in this configuration than we did in the basic case. With the dual-fetch case, we are essentially doubling the cache size that we used in the basic fetch case for the MBatch and Read-Modify-Write policies. Instead of reducing our cache size from 27,500 pages to 7,500 pages, we are reducing the effective cache size from 55,000 pages to 15,000 pages. This is a much larger reduction in cache size, so we see a larger reduction in throughput when we compare the different cache sizes.

6.2.2 Heavy Fetch Load

Large Cache

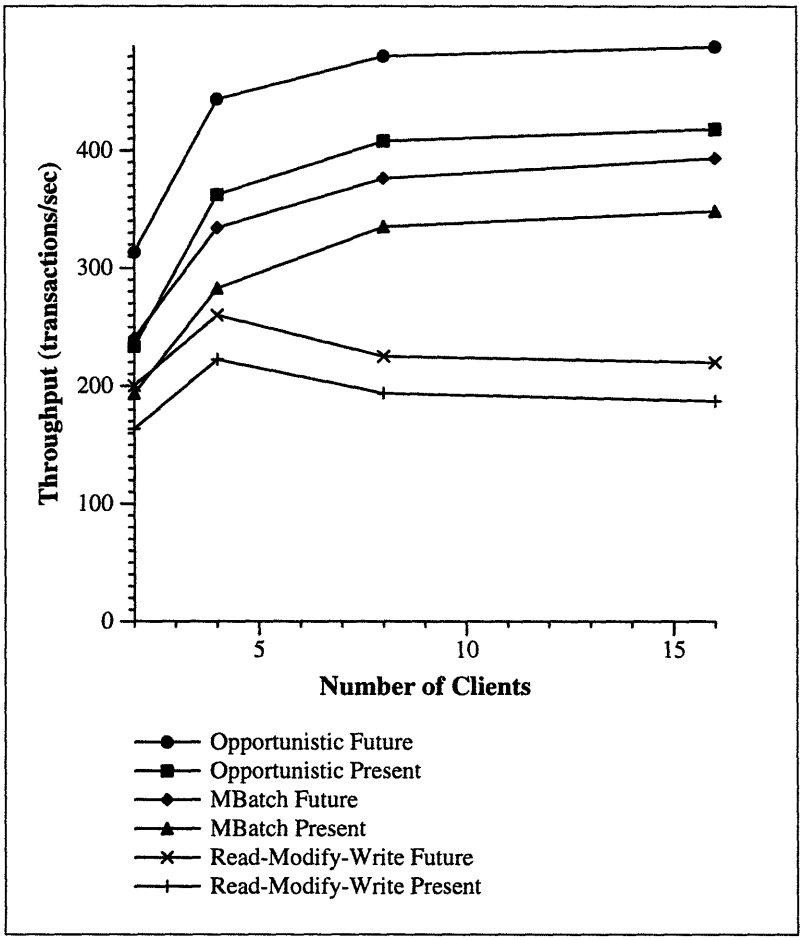

Figure 6-7: Large Cache, Dual-Fetch Configuration, Heavy Fetch Load

For this configuration, the MBatch policy does not enter the non-steady state,

	Opportunistic	MBatch	Read-Modify-Write
Commit Latency (msec)	4.93	4.78	5.04
Fetch Latency (msec)	28.0	41.5	54.3
Log Space Used (objects)	15,819	9,237	25,000

Table 6.7: Further Results. Metrics are for 4 clients, present parameters

so we will consider all of the MBatch results. The higher fetch rate slows down the modification arrival rate, allowing the MBatch policy to remove entries from its log as fast as they enter. For the Read-Modify-Write policy, it does not enter the non-steady state until four clients. Afterwards, it is in the full-log state throughout the trial. Therefore, we will compare all three policies at two clients, and then the MBatch and Opportunistic policies for the rest of the clients.

With only two clients, each policy achieves similar throughput. The Opportunistic has the highest, followed by the MBatch policy and the Read-Modify-Write policy. Related to this, the Opportunistic policy has the lowest fetch latency, followed by the MBatch policy and then the Read-Modify-Write policy. The MBatch policy has higher fetch latency because of the effect of the fetch reads needing to wait behind an entire batch.

The Read-Modify-Write policy has even higher fetch latency because of the high iread and write cost. Since we are in the dual-fetch case, more of the writes will require a random access time, since a page owned by the other server is written to disk as soon as it is received from that server. Since the cost of these operations is higher, the fetch reads must wait longer before they can be processed, leading to the higher latency.

We also notice that the disk is much more utilized in the Read-Modify-Write policy than in the other two. This is expected, since the Read-Modify-Write policy has almost achieved its maximum throughput (see table 5.10), while the MBatch and

Opportunistic policies can still process a much higher rate of transactions per second.

Even with the lower server cache hit rate, the Opportunistic policy still has its throughput advantage over the MBatch policy. This makes sense when we consider that with the smaller cache, the throughput was only slightly less than with the large cache. In the dual-fetch case, the effective cache size, when we consider that each server is only caching its own pages, is 55,000 pages for the MBatch policy, and 51,500 pages for the Opportunistic policy when we account for the dirty pages cached at both servers. This is a small difference, so there should be little effect on the throughput of the Opportunistic policy.

With more than two clients, the Opportunistic policy still has a lower fetch latency than the MBatch policy. Because of this, we see that the Opportunistic policy achieves higher throughput than the MBatch policy. With the heavy fetch load, the policy's ability to process fetch requests quickly is extremely important. Therefore, we observe that the gap between the throughputs of the Opportunistic and MBatch policies is greater than in the normal fetch load. Part of this is due to the Opportunistic write cost being what we expect, but part of it is also due to the improved fetch latency.

When compared with the mathematical model (see table 5.10), we see that the Opportunistic and Read-Modify-Write policies achieve approximately the throughput we predicted. The Read-Modify-Write policy is able to handle two clients in this configuration, which we should have seen given that the observed throughput was less than the maximum predicted throughput. When we increased the workload, the policy entered the non-steady and full-log states, which makes sense because with two clients, its throughput was very close to the maximum value. With the Opportunistic policy, the throughput we observe at sixteen clients, when the policy enters the saturated state, is almost exactly what we predicted in table 5.10. With the MBatch policy, the observed throughput is less than what we predicted. We see the same factors causing this here that we did in the normal fetch load: higher fetch latency and iread cost.

Small Cache

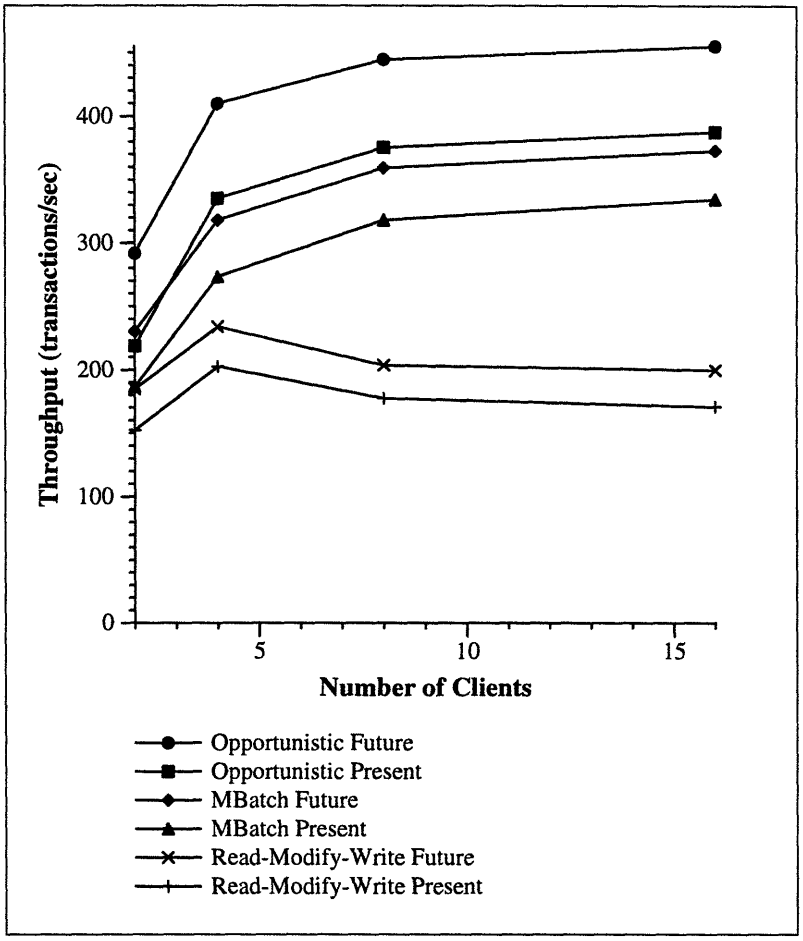

Figure 6-8: Small Cache, Dual-Fetch Configuration, Heavy Fetch Load

	Opportunistic	MBatch	Read-Modify-Write
Commit Latency (msec)	4.89	4.72	4.83
Fetch Latency (msec)	33.1	43.3	63.4
Log Space Used (objects)	17,077	14,048	25,000

Table 6.8: Further Results. Metrics are for 4 clients, present parameters

Again, as predicted in the mathematical model, using the small cache has little effect on the throughput. As with the normal fetch load, we see a larger reduction in throughput than with the basic fetch policy. The cache-splitting algorithm essentially doubles the size of the cache, so there is a larger difference in effective cache sizes between the large and small caches.

We also observe that the Opportunistic policy's throughput is lowered by a larger factor than the MBatch and Read-Modify-Write policies. This is also due to the cache-splitting algorithm, since the dirty pages are stored in the cache at both servers. With the small cache, the dirty pages take up a larger percentage of the cache than they did with the larger cache, so holding the dirty pages in both caches has a more noticeable effect.

Chapter 7

Conclusion

In this chapter, we will attempt to combine all of the results that we have presented so that we can determine which update policy is the best. We have created a mathematical model of Thor to predict how each update policy would perform with certain system parameters. In order to validate the analytical model, we developed a simulator of Thor. With the calculations we have made using the mathematical model and the results we have obtained from the simulator, we will be able to confirm that the Opportunistic policy was the best policy to use with these server parameters.

First, we will examine the policies, summarizing the simulation results that we observed. We will determine how much Thor would benefit from using the scheduling algorithms, and we will compare the two different policies which scheduled ireads and writes. Next, we will make some general observations that we observed from across the simulation results. Afterwards, we will discuss ways in which this thesis could be improved, and future research that could be done based on this work.

7.1 General Policy Comparisons

This section will examine all of the simulation results and mathematical modeling in order to determine which policy should be used. First, we will present the benefits that we saw from using the Opportunistic and MBatch policies as opposed to the Read-Modify-Write. After that, we will examine the Opportunistic and MBatch

policies side-by-side. In chapter 6, we did this for each configuration, but now we will look for generalities in those results and present them in order to determine which one should be used. We will also examine the future systems in this section.

7.1.1 Scheduling vs. Non-scheduling

Both the mathematical model and the simulation results show that scheduling ireads and writes provides much higher throughput than issuing them in a FIFO manner. The cost of these operations becomes lower when we schedule them, allowing the disk to process a higher transaction arrival rate. In the simulator, the Read-Modify-Write policy was in the non-steady or full-log state even with two clients for almost all of the configurations. Therefore, we recommend that a policy that scheduled ireads and writes be used in order to provide higher throughput in the system. Even while the policy was in the non-steady state, where throughput was elevated, it could not process as many transactions per second as the policies which did schedule the ireads and writes.

The only time in which the Read-Modify-Write policy was in the steady state with two clients was in the dual-fetch case with a heavy fetch load. In that case, the Read-Modify-Write achieved only slightly less throughput than the Opportunistic and MBatch policies. However, the disk utilization in the policy was much higher than that of the other two policies, meaning it could not achieve higher throughput. Also, it could not process fetch requests as quickly as the other two policies, since the disk queue was made up of operations with a much higher cost than with the other two policies.

However, with a small number of clients, the effects of the update policy on throughput will be minimal. An update policy is used for increasing the maximum throughput that a server configuration can achieve. If we are operating at a low transaction arrival rate, then there is little need for an update policy, since the disk would not be fully utilized. However, the update policy will affect response time instead of the throughput. With the MBatch and Opportunistic policy, the ireads and writes take less time, so the disk queue can be processed faster, leading to lower

fetch latency.

In [4], a study was done of the MOB under different server configurations. The maximum throughput was achieved when a larger amount of memory was allocated to the MOB. Since the MOB is identical to the Read-Modify-Write policy, we can make a similar conclusion about the Read-Modify-Write policy. However, the MOB used a much larger page size than we used, so the server cache became less efficient. Because of this parameter difference, we can not make a comparison between the MOB results and the MBatch and Opportunistic policy results.

7.1.2 Opportunistic vs. MBatch

Using the mathematical model, we predicted that the MBatch policy would achieve the highest throughput. When we held absorption the same for every policy, the iread trigger value for the MBatch policy became higher than that of the Opportunistic policy. Therefore, ireads and writes took less time under the MBatch policy, offsetting the penalty the policy pays when issuing the first write in a batch. Therefore, under the mathematical model, the disk was able to process the fetch reads, ireads, and writes that resulted from a certain number of commit requests at a faster speed than the Opportunistic policy, leading to higher throughput.

However, we observed that the Opportunistic policy achieved the highest throughput in the simulated system in almost all of the tests. The only case in which the MBatch policy achieved the highest throughput without entered the non-steady state was with the large cache and normal fetch load in the dual-fetch scheme. However, the Opportunistic policy behaved inexplicably in this test, so the accuracy of the results in uncertain. Therefore, we will use the other simulation results as our basis

One thing our mathematical model did not take into consideration was waiting time for fetch reads. In the MBatch policy, an incoming fetch read would be queued behind an entire batch of ireads or writes before it could be processed. In the Opportunistic case, writes and ireads were only scheduled one at a time, so the disk queue length was generally shorter than the MBatch's. Therefore, the fetch reads could be serviced faster once they were placed on the disk queue. While the fetch

read was waiting to be processed, the client was forced to wait until the read was serviced before it could send a commit request to the server. Because of the higher fetch latency in the MBatch policy, there were longer periods of time when a client was simply waiting, not issuing any commit requests to the database. This lowered the throughput of the system, since time was progressing with no commit requests being made. As a result, the Opportunistic policy achieved a higher throughput.

With a higher number of clients under these parameters, the MBatch policy could not clean the log as fast as modifications entered. Therefore, the policy entered the full-log state. In this state, commit requests were forced to wait until log space was available before they could be processed, leading to lower throughput. In the Opportunistic case, the server just kept scheduling ireads and writes as needed. As a result, the fetch latency increased due to longer disk queues, but overall throughput did not decrease.

Considering the simulation results, it seems clear that we should recommend the Opportunistic policy as the one to use. It achieved the highest throughput in almost every situation, and could handle any number of clients in the system at any time.

7.1.3 Future Systems

In the future systems, we noticed the same results as we did for the systems with the present parameters. Therefore, we will draw the same conclusions for the future system that we did for the present one: the Opportunistic policy is the best policy to use under these parameters.

7.2 Specfic Observations

This section will discuss several observations that we made throughout the results.

7.2.1 Effect of the Cache Size

As shown, and predicted in the mathematical model, the cache size had very little impact on the observed throughput. Since our cache sizes were fairly small (less than 10% of the database), most of the fetch requests would miss in the cache and require a disk read. The larger cache size only saved a few random access reads for the fetches.

With the dual-fetch configuration, we saw that the cache size was slightly more of a factor in the final outcome. However, the benefits of using the larger cache were still minor. In the dual-fetch case, we approximately double the size of the cache, except in the Opportunistic case where the dirty pages are present at each server. Because of the cache splitting, the difference in cache sizes between the large and small cache was greater than in the basic fetch policy. In this case, we saw a greater increase in throughput when using the large cache, but the effects of switching to the small cache were still relatively minor.

The reason there is so little difference in performance is that fetch reads and ireads result from cache misses. Under the basic fetch policy, the mathematical model predicted $\lambda = 8.33\%$ for the large cache and $\lambda = 2.27\%$ for the small cache. This makes the cache miss rate 91.67% and 97.73% respectively. There is very little percentage difference between the two figures, which implies there will be a small percentage difference in the number of ireads and fetch reads we see in the large and small cache size systems. Therefore, we should only see a slight improvement in throughput when we use the large cache in this system.

Therefore, in a system that expects a uniform access to its pages, the cache could be made smaller and the log made bigger. With the MBatch and Read-Modify-Write policies, we could keep a small cache and a large enough log that reduce the policy's probability of entering the full-log state. Keeping the small cache would lower the performance slightly, but the benefits of using the larger log would outweigh the detriments of the small cache. This conclusion matches what was determined for the MOB by [4].

With the Opportunistic policy, this would not work, since the cache is used to

hold the dirty page set, from which the policy schedules writes. We would need to keep the cache large enough that the Opportunistic policy could achieve a low enough average write cost.

7.2.2 Importance of Fetch Response Time

One of the most important metrics in determining a policy's throughput is the fetch latency. In all of the trials, the Opportunistic policy had the lowest fetch latency and the highest throughput, even though the MBatch policy had lower iread cost. Since the iread_trigger value for the MBatch policy was higher than that of the Opportunistic, there were more pages awaiting ireads for the MBatch policy. Therefore, the average iread cost was lower because there were more pages from which to choose.

Generally, the fetch reads used a significant portion of the disk time. If the system was operating at a throughput of T transactions/second, then there would be:

$$T \cdot P_f \cdot (1 - \lambda) \text{ fetch\_reads/second} \tag{7.1}$$

$$T \cdot P_w \text{ writes/second} \tag{7.2}$$

$$T \cdot P_w \cdot (1 - P_f) \cdot (1 - \lambda) \text{ ireads/second} \tag{7.3}$$

For our choices of parameters, $P_f = P_w$ in the heavy fetch load. For the Opportunistic policy, the cost of a random fetch read was approximately 4.25 times greater than than the cost of a write or iread. Therefore, the fetch reads will use approximately 70% of the disk time. Therefore, the policy with the lowest fetch latency should provide the highest throughput, since fetch requests will wait less for that policy. The simulation results support this claim, since the policy with the best fetch response time, the Opportunistic, has the highest throughput of the three policies.

With our parameters in the normal fetch load, $4 \cdot P_f = P_w$. Even with the lower number of fetch reads, they will use approximately 35-40% of the disk time. This makes fetch latency less important than in the heavy fetch load, but it will still be an important factor in the final outcome. This is also shown in the simulation results. When we examine the cases with the normal fetch load, the Opportunistic policy

still has the highest throughput. However, the margin by which it is higher than the MBatch policy is less than it was in the case with the heavy fetch load. With the normal fetch load, the write and iread cost become more important since they are taking up more of the disk time than previously. However, the improvement in write and iread cost is not enough for the MBatch to achieve higher throughput than the Opportunistic policy.

The above analysis is for the basic fetch policy cases. In the dual-fetch case, we divide the fetch load between the two servers, so the disk load of the fetch requests will be divided in half for this situation. For the heavy fetch load, fetch requests will have similar utilization as the normal fetch load with the basic fetch policy. Therefore, fetch latency will still be an important factor in the final outcome. With the normal fetch load, fetch latency will be less significant a factor, as only about 20% of the disk usage will be for fetches. This is seen in the simulation results, as the Opportunistic has a very slight edge in throughput over the MBatch policy.

We also divide the ireads in the dual-fetch configuration, which increases the importance of the write cost. Therefore, the Opportunistic policy has another advantage over the MBatch policy, since the first write in the batch will pay a random cost. The other writes in the batch will help to amortize this cost, but the Opportunistic policy still had a lower write cost than the MBatch policy.

7.3 Further Experimentation

The section will discuss future work that we think could be done as a next step from what we have presented here. We will also discuss ways in which we think that the current thesis could be improved.

7.3.1 Problems with the Tests

One thing that could be done to improve the results would be to run the trials for longer periods of time. In our simulator, the test run lasted until 70,000 writes had been issued. However, with the current run length some policies did not reach the

steady state. By extending the trial length, we would get a more accurate view of the steady state, and we would allow some configurations to reach the steady state throughput. Longer tests runs would eliminate the non-steady state from the results, as each policy in that state would reach the full-log state, so we would see a lower throughput for that case.

We would have liked to have used longer run lengths, but there are some problems associated with this. We needed to run the simulator on several shared machines. The simulator consumed a lot of memory and CPU time, so the trials had to be run in the middle of the night in order not to disturb the work of others. Even with a run length of 70,000 writes, the test runs took several hours. By lengthening them it would take much longer to gather the results.

7.3.2 Skewed Access Mathematical Model

Pages were selected in a uniform access pattern in our tests. We would have also liked to have tested a skewed access pattern, where a higher percentage of transactions would be to a small percentage of the database. However, we ran into problems with this because we were unable to develop a mathematical model of the server cache hit rate or write absorption with a skewed access pattern.

For the cache hit rate, unless the set of hot pages fits inside the cache, we would need to determine the percentage of hot and cold pages in the cache. However pages are entering the cache from two sources: fetch requests and ireads. For the fetch requests, we would know what percentage of those requests went to hot pages. However, for the ireads, we would need to determine the percentage of hot and cold objects in the log. From this we could develop a system of equations that would allow us to determine the percentage of hot and cold pages in the cache in the steady state. However, with the ireads, we would need some way of modeling the selection of pages from the log, since the MBatch and Opportunistic policies use a scheduling algorithm to issue the ireads. This became too complicated, and we were unable to create the model. Therefore, we decided to just use the uniform access model.

7.3.3 MBatch and Opportunistic Comparison

We compared the MBatch and Opportunistic policies with a certain set of server parameters. However, are these the best parameters for each policy to operate under? Probably not, since the MBatch policy entered the full-log state during our tests. We also believe that we could manipulate the parameters for the Opportunistic policy in order to extract a higher throughput from the system.

The mathematical model provides a good tool for experimenting with the server configurations. The equations will provide the expected throughput for each policy based on the system parameters. From these equations, the optimal configuration could be determined for both policies. The simulator could then be used to test each configuration in order to determine which policy was truly the best. Future work could be based on trying to different configurations with each policy, and then comparing the observed throughput under different configurations.

Bibliography

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control for distributed database systems. In *Proceedings of 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA., May 1995.

[2] E. Coffman, L. Klimko, and B. Ryan. Analysis of scanning policies for reducing disk seek time. *SIAM Journal of Computing*, 1(3), September 1972.

[3] H. Garcia-Molina and C.A. Polyzois. Processing of read-only queries at a remote backup. Technical Report CS-TR-354-91, Department of Computer Science, Princton University, December 1991.

[4] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases.* PhD thesis, Massachusetts Institute of Technology, 1995.

[5] S. Ghemawat, R. Gruber, J. O'Toole, and L. Shrira. Coordinated resource management in a replicated object server. Technical Report TM505, MIT/LCS, February 1994.

[6] R. Gruber. Notes for upcoming thesis.

[7] M. Hofri. Disk scheduling: FCFS vs SSTF revisited. *Communications of the ACM*, 2(3), August 1984.

[8] R. Katz, G. Gibson, and D. Patterson. Disk system architecture for high performance computing. *Proceedings of the IEEE*, 77(12):1842–1857, 1989.