# Replication Control in Distributed B-Trees

by

Paul Richard Cosway

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

Bachelor of Science

and

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Signature of Author ........ [illegible] ........
Department of Electrical Engineering and Computer Science
September 3, 1994

Certified by ........ [illegible] ........
William E. Weihl
Associate Professor of Computer Science
Thesis Supervisor

Accepted by ........ [illegible] ........
Frederic R. Morgenthaler
Chair, Department Committee on Graduate Students

# Replication Control in Distributed B-Trees

by

Paul R. Cosway

## Abstract

*B-trees* are a common data structure used to associate symbols with related information, as in a symbol table or file index. The behavior and performance of B-tree algorithms are well understood for sequential processing and even concurrent processing on small-scale shared-memory multiprocessors. Few algorithms, however, have been proposed or carefully studied for the implementation of concurrent B-trees on networks of *message-passing multicomputers*. The distribution of memory across the several processors of such networks creates a challenge for building an efficient B-tree that does not exist when all memory is centralized – distributing the pieces of the B-tree data structure. In this work we explore the use and control of replication of parts of a distributed data structure to create efficient distributed B-trees.

Prior work has shown that replicating parts of the B-tree structure on more than one processor does increase throughput. But while the one original copy of each tree node may be too few, copying the entire B-tree wastes space and requires work to keep the copies consistent. In this work we develop answers to questions not faced by the centralized shared-memory model: which B-tree nodes should be copied, and how many copies of each node should be made. The answer for a particular tree can change over time. We explore the characteristics of optimal replication for a tree given a static pattern of accesses and techniques for dynamically creating near-optimal replication from observed access patterns.

Our work makes three significant extensions to prior knowledge:

- It introduces an analytic model of distributed B-tree performance to describe the tradeoff between replication and performance.
- It develops, through analysis and simulation, rules for the use of replication that maximize performance for a fixed amount of space, updating the intuitive rules of prior work.
- It presents a description and analysis of an algorithm for dynamic control of replication in response to changing access patterns.

Thesis Supervisor: William E. Weihl
Title: Associate Professor of Computer Science

This work was supported indirectly by the Advanced Research Projects Agency under Contract N00014-91-J-1698, by grants from IBM and AT&T, and by an equipment grant from DEC.

# Acknowledgements

I owe a tremendous debt of gratitude to all of my family for their patience, support, and occasional prodding during the long-delayed completion of this work. Their waiting and wondering may have been harder work than my thinking and writing. I owe much more than gratitude to Tanya for all of the above and her (partial) acceptance of the complications this has added to our already complicated life. It's time to move on to new adventures.

Several members of the Laboratory for Computer Science have contributed to this work. Bill Weihl's willingness to include me in his research group and supervise this work made it possible for me to undertake this effort. His insights and suggestions along the way were invaluable. John Keen helped me structure the original ideas that led to the selection of this topic, and Paul Wang contributed his knowledge and experience from his prior work on B-trees. Eric Brewer's simulation engine, Proteus, made it possible to obtain simulation results and was a pleasure to use. In addition to his always available ear to help me test ideas, Brad Spiers was (and is) a great friend.

Finally, for their help in times of crisis near the end, I am eternally grateful to Anthony Joseph for jiggling the network connection to reconnect my workstation to the network and to Yonald Chery for correctly suggesting that power-cycling the printer might make it work again.

# Contents

# List of Figures

# Chapter 1

# Introduction

*B-trees* are a common data structure used to associate symbols with related information, as in a symbol table or file index. The behavior and performance of B-tree algorithms are well understood for sequential processing and even concurrent processing on small-scale shared-memory multiprocessors. Few algorithms, however, have been proposed or carefully studied for the implementation of concurrent B-trees on networks of *message-passing multicomputers*. The distribution of memory across the several processors of such networks creates a challenge for building an efficient B-tree that does not exist when all memory is centralized – distributing the pieces of the B-tree data structure. In this thesis we explore the use and control of replication of parts of a distributed data structure to create efficient distributed B-trees.

The reader unfamiliar with the basics of B-trees is referred to Comer's excellent summary [Com79]. In brief, the B-tree formalizes in a data structure and algorithm the technique one might use in looking up a telephone number in a telephone directory, shown graphically in figure 1-1. Begin at a page somewhere near the middle of the directory; if the sought after name is alphabetically earlier than the names on that page, look somewhere between the beginning of the directory and the current page. If the name is now alphabetically later than the names on the new page, look somewhere between this page and the page just previously examined. If this process is continued, it will quickly reach the page that should hold the desired name and number – if the name is not found on that page, it is not in the directory.

The problems encountered when using the conventional B-tree structure on a message-passing multicomputer are similar to those of a large city with only one copy of its telephone

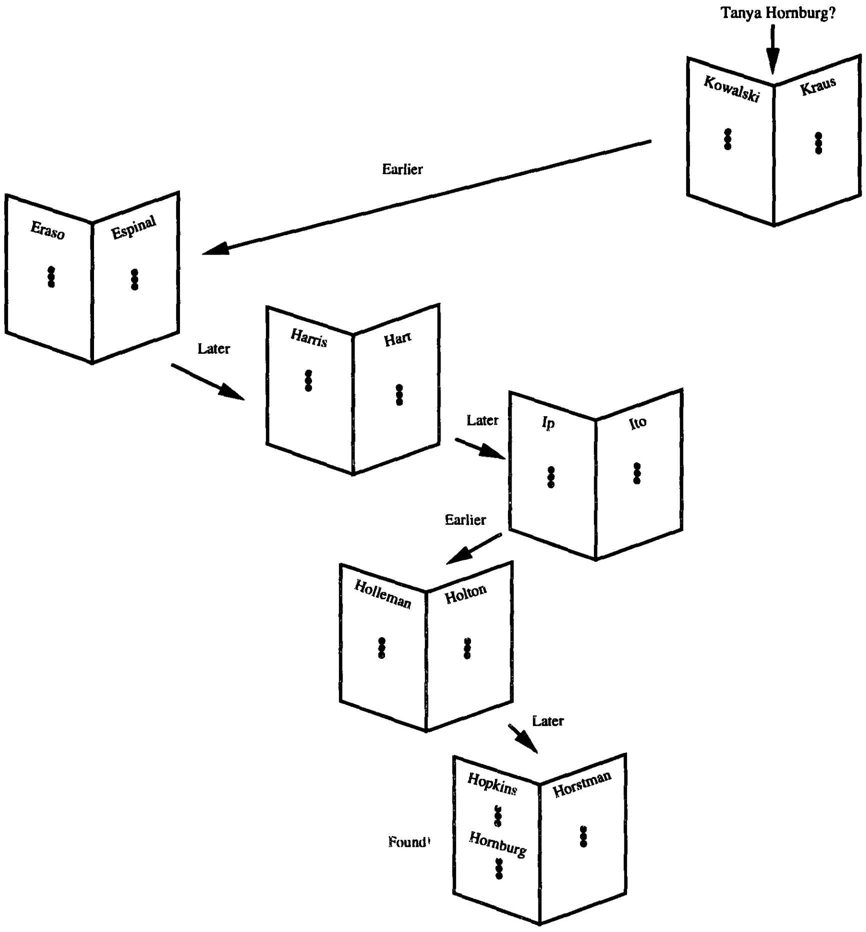


Figure 1-1: Telephone Directory Lookup

directory – only one person can use the directory at a time and to use it each person must travel to the location of the directory. If the single copy of the directory is divided up with pieces placed in a number of locations, more people may be able to use the directory at a time, but the number of people able to use the directory at any one time would still be limited and each person might have to visit several locations to find the piece of the directory holding his or her sought after entry. The telephone company solves these problems by giving a copy of the directory to every household, but this solution has weaknesses that we do not wish to introduce to the B-tree data structure. First, printing all those copies uses up a great deal of paper, or memory in the B-tree version. Second, the directory is wrong almost as soon as it is printed – telephone numbers are added, removed and changed every day. Fortunately for the Postal

Service, the telephone company does not send out daily updates to all its customers. While users of the telephone directory can tolerate the directory growing out of date, the users of a B-tree demand that it always accurately reflect all prior additions and deletions. Wouldn't it be nice if we could all look up telephone numbers nearly as quickly as we can each using our own directory, but using only a fraction of the paper and always guaranteed to be accurate! That is analogous to our objective in controlling replication in distributed B-trees.

The B-tree algorithm was developed and is used extensively on traditional, single processor computers and is also used on multiprocessors with a shared central memory. Recent trends in computer architecture suggest the B-tree should be studied on a different architectural model. A number of new multiprocessor architectures are moving away from the model of a small number of processors sharing a centralized memory to that of a large number of independent processors, each with its own local memory, and linked by passing messages between them [Dal90, ACJ+91]. The aggregate computing power of the tens, hundreds, or even thousands of processors hooked together is substantial – if they can be made to work together. However, the physical and logical limitations of sharing information across such a network of processors create difficulties in making the processors work together. For example, while each processor can indirectly read or write memory on another processor, it is much faster to directly access local memory than to exchange messages to access memory on a remote processor. And if every processor needs to read or write from the same remote processor, the read and write request messages must each wait their turn to be handled, one at a time, at that remote processor. To most effectively take advantage of the potential computing power offered by these new architectures, the computation and data for a problem must be *distributed* so that each of the many processors can productively participate in the computation while the number of messages between processors is minimized.

If the nodes of a B-tree are distributed across the $n$ processors of a message-passing multicomputer instead of residing on only one processor, we would like to see an $n$ times increase in B-tree operation throughput (or an $n$ times reduction in single operation latency). Unfortunately, there cannot be an immediate $n$ times increase in throughput, for the B-tree structure itself limits the throughput that can be achieved. Since all operations must pass through the single root node of the B-tree, the processor that holds the root must be involved in every B-tree operation. The throughput of that single processor presents a *bottleneck* that limits the overall throughput. As for single operation latency, it will increase, not decrease. Once past the root, a B-tree search will almost always have to visit more than one processor to find all the nodes on the path to the destination leaf. Since each inter-processor message increases operation latency, simply distributing the B-tree nodes across many processors guarantees that latency of a single operation will increase.

The obvious solution is to create replicas or copies of selected B-tree nodes on other processors to reduce or eliminate the root bottleneck and reduce the volume of inter-processor messages. Wang [Wan91] has shown that replicating parts of the B-tree structure on more than one processor does increase throughput. But while the one original copy of each tree node may be too few, copying the entire B-tree wastes space and requires work to keep the copies consistent. Thus, in building a B-tree on a distributed-memory message-passing architecture we must address problems not faced by the centralized shared-memory model: we must determine which B-tree nodes should be copied, and how many copies of each node should be made. The answer for a particular tree can change over time. If the B-tree and the pattern of access to the tree remain static, the replication decision should also remain static. But if the pattern of accesses to the B-tree changes over time in such a way that an initial decision on replication is no longer suited to the current access pattern, we would also like to dynamically control the replication to optimize B-tree performance.

To date little work has been done on the static or dynamic problem. Lehman and Yao [LY81] developed a B-tree structure that allows concurrent access, but has been historically applied to single processors and shared-memory multiprocessors. Of the work done with distributed B-trees, Wang [Wan91] showed that increased throughput can be obtained through replicating parts of the B-tree structure, but did not directly address how much replication is necessary or how it can be controlled. Johnson and Colbrook [JC92] have suggested an approach to controlling replication that we label "path-to-root", but it has not yet been tested. This work is being extended by Johnson and Krishna [JK93]. Both pieces of prior work suggest using replication in patterns that make intuitive sense, but both produce replication patterns that are independent of actual access pattern and do not allow changes in the tradeoff between replication and performance.

We start from this prior work and use a combination of simulation and analytic modeling to study in detail the relationship between replication and performance on distributed B-trees. In this work we do not study the related decision of where to place the nodes and copies. We place nodes and copies randomly because it is simple and produces relatively good balancing without requiring any knowledge of other placement decisions. Our work makes three significant extensions to prior knowledge:

- It introduces an analytic model of distributed B-tree performance to describe the tradeoff between replication and performance.
- It develops, through analysis and simulation, rules for the use of replication that maximize performance for a fixed amount of space, updating the intuitive rules of prior work.
- It presents a description and analysis of an algorithm for dynamic control of replication in response to changing access patterns.

In the body of this thesis we expand on the challenges of creating replicated, distributed B-trees, our approach to addressing the challenges, and the results of our simulation and modeling. The key results are developed in chapters 5, 6, and 7.

- Chapter 2 presents relevant prior work on concurrent and distributed B-tree algorithms;
- Chapter 3 describes key characteristics of the system we used for simulation experiments;
- Chapter 4 presents a queueing network model for the performance of replicated B-trees;
- Chapter 5 presents a validation of the queueing network model against simulation experiments;
- Chapter 6 uses the results of simulation and modeling of static replication patterns to develop replication rules to optimize performance;
- Chapter 7 describes an approach to the dynamic control of replication and analyzes the results of simulations;
- Chapter 8 summarizes the conclusions of our work and indicates avenues for further investigation.

# Chapter 2

# Related Work

The original B-tree algorithm introduced by Bayer and McCreight [BM72] was designed for execution on a single processor by a single process. Our current problem is the extension of the algorithm to run on multiple processors, each with its own local memory, and each with one or more processes using and modifying the data structure. The goal of such an extension is to produce a speedup in the processing of B-tree operations. In this work we seek a speedup through the concurrent execution of many requests, not through parallel execution of a single request. Kruskal [Kru83] showed that the reduction in latency from parallel execution of a single search is at best logarithmic with the number of processors. In contrast, Wang's study of concurrent, distributed B-trees with partial node replication [Wan91], showed near linear increases in lookup throughput with increasing processors.

To efficiently utilize many processors concurrently participating in B-tree operations, we must extend the B-tree algorithm to control concurrent access and modification of the B-tree, and to efficiently distribute the B-tree data structure and processing across the several processors. In this section we look at prior work that has addressed these two extensions.

## 2.1 Concurrent B-tree Algorithms

The basic B-tree algorithm assumes a single process will be creating and using the B-tree structure. As a result, each operation that is started will be completed before a subsequent operation is started. When more than one process can read and modify the B-tree data structure simultaneously (or apparently simultaneously via multi-processing on a single processor) the data structure and algorithm must be updated to support concurrent operations.

A change to the basic algorithm is required because modifications to a B-tree have the potential to interfere with other concurrent operations. Modifications to a B-tree result from an *insert* or *delete* operation, where a key and associated value are added to or deleted from the tree. In most cases, it is sufficient to obtain a write lock on the leaf node to be changed, make the change, and release the lock without any interference with other operations. However, the insert or delete can cause a ripple of modifications up the tree if the insert causes the leaf node to split or the delete initiates a merge. As a split or merge ripples up the tree, restructuring the tree, it may cross paths with another operation descending the tree. This descending operation is encountering the B-tree in an inconsistent state and, as a result, may finish incorrectly. For example, just after a node is split but before a pointer to the new sibling is added in the parent node, any other B-tree operation has no method of finding the newly created node and its descendants. Two methods have been proposed to avoid this situation, *lock coupling* and *B-link trees.*

Bayer and Schkolnick [BS77] proposed lock coupling for controlling concurrent access. To prevent a reader from "overtaking" an update by reading a to-be-modified node before the tree has been made fully consistent, they require that a reader obtain a lock on a child node before releasing the lock it holds on the current node. A writer, for its part, must obtain an exclusive lock on every node it intends to change prior to making any changes. Thus, a reading process at a B-tree node is guaranteed to see only children that are consistent with that current node. Lock coupling prevents a B-tree operation from ever seeing an inconsistent tree, but at the expense of temporarily locking out all access to the part of the tree being modified. The costs of lock coupling increase when the B-tree is distributed across several processors and some nodes are replicated – locks must then be held across several processors at the same time.

Lehman and Yao [LY81] suggested the alternative of B-link trees, a variant of the B-tree in which every node is augmented with a link pointer directed to its sibling on the right. The B-link tree also requires that a split always copy into the new node the higher values found in the node being split, thus placing the new node always logically to the right of the original node. This invariant removes the need for lock coupling by allowing operations to correct themselves when they encounter an inconsistency. An operation incorrectly reaching a node that cannot possibly contain the key it is seeking (due to one or more "concurrent" splits moving its target to the right) can follow the link pointer to the right until it finds the new correct node. Of course, writers must still obtain exclusive locks on individual nodes to prevent them from interfering with each other and to prevent readers from seeing an inconsistent single node, but only one lock must be held at a time.

The right-link structure only supports concurrent splits of B-tree nodes. The original proposal did not support the merging of nodes. Lanin and Shasha [LS86] proposed a variant with "backlinks" or left-links to support merging. Wang [Wan91] added a slight correction to this algorithm.

Other algorithms have been proposed, as well as variants of these [KW82, MR85, Sag85], but lock coupling and B-link remain the dominant options. All proposals introduce some temporary limit on throughput when performing a restructuring modification, either by locking out access to a sub-tree or lengthening the chain of pointers that must be followed to reach the correct leaf. Analysis of the various approaches has shown that the B-link algorithm can provide the greatest increases in throughput [LS86, JS90, Wan91, SC91].

We use the B-link algorithm and perform only splits in our simulations. The B-link algorithm is particularly well suited for use with replicated B-tree nodes because it allows tree operations to continue around inconsistencies, and inconsistencies may last longer than with a shared memory architecture. B-tree nodes will be temporarily inconsistent both while changes ripple up the tree and while the changes further ripple out to all copies of the changed nodes. When one copy of a node is modified, the others are all incorrect. The updates to copies of nodes cannot be distributed instantaneously and during the delay we would like other operations to be allowed to use the temporarily out-of-date copies. As Wang [WW90] noted in his work on multi-version memory, the B-link structure allows operations to correct themselves by following the right link from an up-to-date copy if they happen to use out-of-date information and reach an incorrect tree node. Of course, when an operation starts a right-ward traversal, it must follow up-to-date pointers to be sure of finding the correct node.

## 2.2 Distributed B-tree Algorithms

The B-link algorithm provides control for concurrent access to a B-tree that may be distributed and replicated, but does not provide a solution to two additional problems a distributed and replicated B-tree presents: distributing the B-tree nodes and copies, and keeping copies up to date.

Before examining those problems, it should be noted that there have been proposals for concurrent, distributed B-trees that do not replicate nodes. Carey and Thompson [CT84] suggested a pipeline of processors to support a B-tree. This work has been extended by Colbrook, et al. [CS90, CBDW91]. In these models, each processor is responsible for one level of the B-tree. This limits the amount of parallelism that can be achieved to the depth of the tree. While trees can be made deeper by reducing the branch factor at each level, more levels means more messages between processors, possibly increasing the latency of a search. But the most significant problem with the pipeline model is data balancing. A processor must hold every node of its assigned tree level. Thus, the first processor holds only the root node, while the last processor in the pipeline holds all of the leaves.

Our focus in this work is on more general networks of processors and on algorithms that can more evenly distribute and balance the data storage load while also trying to distribute and balance the processing load.

### 2.2.1 B-tree Node Replication

Whenever a B-tree node is split, a new node must be created on a processor. When the tree is partially replicated, the decision may be larger than selecting a single processor. If the new node is to be replicated, we must decide how many copies of the node should be created, where *each* copy should be located, and which processors that hold a copy of the parent node should route descending B-tree operations to each copy of the new node. These decisions have a dramatic impact on the balance of both the data storage load and the operation processing load, and thus on the performance of the system. If there are not enough copies of a node, that node will be a bottleneck to overall throughput. If the total set of accesses to nodes or copies is not evenly distributed across the processors, one or more of the processors will become a bottleneck. And if too many copies are created, not only is space wasted, processing time may also be wasted in keeping the copies consistent.

Since the size and shape of a B-tree and the volume and pattern of B-tree operations are dynamic, replication and placement decisions should also be dynamic. When the root is split, for example, the old root now has a sibling. Copies of the new root and new node must be created, and some copies of the old root might be eliminated. Thus, even under a static B-tree operation load, dynamic replication control is required because the tree itself is changing. When the operation load and access patterns are also changing, it is even more desirable to dynamically manage replication to try to increase throughput and reduce the use of memory.

To date there has been little or no work studying dynamic replication for B-trees or even the relationship between replication and performance under static load patterns. However, we take as starting points the replication models used in previous work on distributed B-trees.

Wang's [Wan91] work on concurrent B-trees was instrumental in showing the possibilities of replication to improve distributed B-tree performance. This work did not explicitly address the issue of node and copy placement because of constraints of the tools being used. In essence, the underlying system placed nodes and copies randomly. Wang's algorithm for determining the number of copies of a node is based on its height above the leaf level nodes. Leaf nodes themselves are defined to have only one copy. The number of copies of a node is the *replication factor* (RF), a constant, times the number of copies of a node at the next lower level, but never more than the number of participating processors. For a replication factor of 7, for example, leaves would have one copy, nodes one level above the leaves would have 7 copies, and nodes two levels above the leaves would have 49 copies. The determination of the key parameter, the replication factor, was suggested to be the average branch factor of the B-tree nodes.

Using this rule and assuming that the B-tree has a uniform branch factor, BF, and a uniform access pattern, the replicated tree will have the same total number of nodes and copies at each level. The exception is when a tree layer can be fully replicated using fewer copies. The number of copies per node, therefore, is proportional to the relative frequency of access. This distribution of copies makes intuitive sense, since more copies are made of the more frequently accessed B-tree nodes. Figure 2-1 shows the calculation of relative access frequency and copies per level, where the root is defined to have relative access frequency of 1.0.

Johnson and Colbrook [JC92] suggested a method for determining where to place the copies

| Level | Relative Frequency | Copies |
|---|---|---|
| h | 1 | $min(P, RF^h)$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| 3 | $1/BF^{(h-3)}$ | $min(P, RF^3)$ |
| 2 | $1/BF^{(h-2)}$ | $min(P, RF^2)$ |
| 1 | $1/BF^{(h-1)}$ | $min(P, RF)$ |
| 0 | $1/BF^{(h)}$ | 1 |

Figure 2-1: Copies per level – Wang's Rule

of a node that also determines the number of copies that must be created. Their copy placement scheme is "path-to-root", i.e., for every leaf node on a processor, the processor has a copy of every node on the path from the leaf to the root, including a copy of the root node itself. Thus, once a search gets to the right processor, it does not have to leave. Without the path-to-root requirement, a search may reach its eventual destination processor, but not know that until it has visited a node on another processor. The path-to-root method requires no explicit decision on how many copies of a node to create. Instead, the number is determined by the locations of descendant leaf nodes. The placement of leaf nodes becomes the critical decision that shapes the amount of replication in this method.

For leaf node placement, Johnson and Colbrook suggest keeping neighboring leaf nodes on the same processor as much as possible. This minimizes the number of copies of upper-level nodes that must exist and may reduce the number of inter-processor messages required. They are developing a placement algorithm to do this. To do so they introduce the concept of *extents*, defined as a sequence of neighboring leaves stored on the same processor. They also introduce the dE-Tree (distributed extent tree) to keep track of the size and location of extents. When a leaf node must be created, they first find the extent it should belong to, and then try to add the node on the associated processor. If adding to an extent will make a processor more loaded than is acceptable, they suggest shuffling part of an extent to a processor with a neighboring extent, or if that fails, creating a new extent on a lightly loaded processor. This proposal has not been fully analyzed and tested, so it is not known whether the overhead of balancing is overcome by the potential benefits for storage space and B-tree operation time.

In our work we identify the path-to-root approach when using random placement of leaf nodes as "random path-to-root" and when using the copy minimizing placement as "ideal path-

| Level | Relative Frequency | Copies |
|---|---|---|
| h | 1 | $place(BF^h, P)$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| 3 | $1/BF^{(h-3)}$ | $place(BF^3, P)$ |
| 2 | $1/BF^{(h-2)}$ | $place(BF^2, P)$ |
| 1 | $1/BF^{(h-1)}$ | $place(BF, P)$ |
| 0 | $1/BF^{(h)}$ | 1 |

Figure 2-2: Copies per level – Random Path-To-Root Rule

to-root". The random path-to-root method uses a similar amount of space to Wang's method. It might be expected to use exactly the same amount, for each intermediate level node must be on enough processors to cover all of its leaf children, of which there are $BF^n$ for a node $n$ levels above the leaves. The actual number of copies is slightly less because the number of copies is not based solely on the branch factor and height above the leaves, but on the actual number of processors that the leaf children of a node are found on, typically less than $BF^n$. When a single object is placed randomly on one of $P$ processors, the odds of it being placed on any one processor are $1/P$, the odds of it not being on a specific processor $(1 - 1/P)$. When $m$ objects are independently randomly placed, the odds that none of them are placed on a specific processor are $(1 - 1/P)^m$, thus the odds that a processor holds one or more of the $m$ objects is $1 - (1 - 1/P)^m$. Probabilistically then, the number of processors covered when placing $m$ objects on $P$ processors is:

$$place(m, P) = P * (1 - (1 - \frac{1}{P})^m)$$

Figure 2-2 shows the calculations for the number of copies under random path-to-root.

When using ideal path-to-root, the minimum number of copies required at a level $n$ above the leaves is the number of leaves below each node of the level, $BF^n$, divided by the number of leaves per processor, $BF^h/P$, or $P * BF^{n-h}$. This minimum is obtainable, however, only when the number of leaves below each node is an even multiple of the number of leaves per processor. In general, the average number of copies required is $P * BF^{n-h} + 1 - \frac{P}{BF^h}$, but never more than $P$ copies. (We explain the development of this equation in appendix A.) This rule also results in the number of copies per level being roughly proportional to the relative frequency of access. Figure 2-3 shows the calculations for the number of copies under ideal path-to-root.

Figure 2-4 shows, for these three rules, the calculation of space usage for an example with

| Level | Relative Frequency | Copies |
|---|---|---|
| h | 1 | $P$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| 3 | $1/BF^{(h-3)}$ | $min(P, P * BF^{3-h} + 1 - \frac{P}{BF^h})$ |
| 2 | $1/BF^{(h-2)}$ | $min(P, P * BF^{2-h} + 1 - \frac{P}{BF^h})$ |
| 1 | $1/BF^{(h-1)}$ | $min(P, P * BF^{1-h} + 1 - \frac{P}{BF^h})$ |
| 0 | $1/BF^{(h)}$ | 1 |

Figure 2-3: Copies per level - Ideal Path-To-Root Rule

| | | | Wang | | Random P-T-R | | Ideal P-T-R | |
|---|---|---|---|---|---|---|---|---|
| Level | Nodes | Rel. Freq. | Copies | Total Nodes | Copies | Total Nodes | Copies | Total Nodes |
| 3 | 1 | 1 | 100 | 100 | 97 | 97 | 100 | 100 |
| 2 | 7 | 1/7 | 49 | 343 | 39 | 273 | 15 | 105 |
| 1 | 49 | 1/49 | 7 | 343 | 7 | 343 | 2.75 | 135 |
| 0 | 343 | 1/343 | 1 | 343 | 1 | 343 | 1 | 343 |
| | Total Nodes: | | | 1129 | | 1056 | | 683 |
| | Copies: | | | 729 | | 656 | | 283 |

Figure 2-4: Comparison of Copying Rules
Branch Factor = Replication Factor = 7,Processors = 100, Levels = 4

tree height above leaves, $h = 3$, branch factor and replication factor, $BF = RF = 7$, and number of processors, $P = 100$. In addition to the 400 nodes that form the unreplicated B-tree, the ideal path-to-root rule creates 283 more total copies, random path-to-root 656 more, and Wang's rule 729 more.

Neither the algorithm implemented by Wang nor that proposed by Johnson and Colbrook links placement and replication decisions to a detailed understanding of the relationship between replication and performance or to the actual operation load experienced by a B-tree. Both algorithms can produce balanced data storage and processing loads under a uniform distribution of search keys, but neither body of work is instructive about how replication decisions can be changed to improve or reduce performance, use more or less space, or respond to a non-uniform access pattern.

The work in this thesis is closest to an extension of Wang's work. The copy placement and routing decisions are similar to those of his work, but we eliminate the constant *replication factor* and explore in detail the relationship between the number of copies of B-tree nodes and performance, including the possibility of dynamically changing the number of copies. In chapter 6 we discuss experiments that compare our approach to replication with the possibilities presented by Johnson and Colbrook's path-to-root algorithm.

### 2.2.2 Copy Update Strategy

If there are a number of copies of a B-tree node, there must be a method for updating all of the copies when a change is made to any one of them. However, they do not all have to be updated instantaneously to achieve good B-tree performance. Wang's work [Wan91] showed that B-link algorithms do not require strict coherence of the copies of a node. Instead of an atomic update of all copies, he used a weaker version of coherence called *multi-version memory* [WW90]. Wang demonstrated this approach to coherence dramatically improves concurrent B-tree performance.

Multi-version memory still leaves a choice for how updates are distributed and old versions brought up to date. Two methods have been proposed. Wang required that all modifications are made to a "master" copy of a node, and then sent out the complete new version of the node to update copies. (The original copy of the node is usually identified as the "master".) Johnson and Colbrook [JC92] have proposed sending out just the update transactions to all copies of a node and are exploring an approach to allow modifications to originate at any copy of a node. Of course, if updates are restricted to originate from one "master" copy of a node and ordered delivery of the update transactions is guaranteed, transaction update will produce the same results as sending complete copies.

A major motivation for distributing updates by sending a small update transactions and not the full node contents was to drop the requirement that modifications originate at the "master" copy. To coordinate updates from different processors Johnson and Colbrook introduced the distinction between *lazy* and *synchronizing* updates. Most updates to a B-tree node (leaf or non-leaf) do not propagate restructuring up the tree and, unless they affect the same entry, are commutative. Non-restructuring updates are termed *lazy* and can be done in any order, as long as they are completed before the node must split or merge. Johnson and Colbrook guarantee that concurrent lazy updates will not affect the same entry by limiting replication to non-leaf nodes and requiring all splits and merges to be synchronized by the "master" copy of a node. Thus, the leaf level presents no possibility for a simultaneous insert or delete of the same key because a definite sequence is determined on a single processor. And for all non-leaf nodes, since the insert or delete can come from only the one "master" copy of a child node, all updates to an entry will be made on the one processor holding the "master" of the child, also assuring a definite sequence of updates.

Any tree restructuring operation is called *synchronizing*, and these do not commute. Johnson and Colbrook suggest an algorithm that allows lazy updates to be initiated on any processor, but still requires synchronizing actions to be started on the processor holding the "master" copy. This algorithm has not yet been implemented and requires minor extensions to handle "simultaneous" independent splits correctly, so it will not be fully described here. Johnson and Krishna [JK93] are extending this work.

While the copy update issue is critical to an actual implementation, it is not critical to our study. Therefore we use the simplest method of updating copies and restrict all updates to originate on the processor where the original, or "master", copy of a node was created. Other copies are updated by sending the complete new version of the node after every change.

# Chapter 3

# System Setup

We implemented a distributed B-tree using Proteus, a high-performance MIMD multiprocessor simulator [BDCW91, Del91]. Proteus provided us with a basic multiprocessor architecture – independent processors, each with local memory, that communicate with messages. It also provided exceptionally valuable tools for monitoring and measuring program behavior. On top of Proteus we created a simple structure for distributed, replicated objects, and on top of that, a distributed B-tree. In this chapter we briefly describe those three elements of our simulation system.

## 3.1 Proteus

The Proteus simulation tool provides high-performance MIMD multiprocessor simulation on a single processor workstation. It provides users with a basic operating system kernel for thread scheduling, memory management, and inter-processor messaging. It was designed with a modular structure so that elements of a multiprocessor, the interconnection network for example, can easily be changed to allow simulation of a different architecture. User programs to run on Proteus are written in a superset of C. The resulting executable program provides a deterministic and repeatable simulation that, through selection of a random number seed, also simulates the non-determinism of simultaneous events on a physical multiprocessor.

In addition to its simulation capabilities, Proteus also provides a rich set of measurement and visualization tools that facilitate debugging and monitoring. Most of the graphs included in this thesis were produced directly by Proteus.

Proteus has been shown to accurately model a variety of multiprocessors [Bre92], but the purpose of our simulations was not to model a specific multiprocessor architecture. Rather, it was to adjust key parameters of multiprocessors such as messaging overhead and network transmission delay to allow us to develop an analytic model that could be applied to many architectures.

## 3.2 Distributed, Replicated Objects

The construction of an application using a distributed and replicated data structure required a facility for processing inter-processor messages and an object identification and referencing structure on top of Proteus. The model for both elements was the runtime system of Prelude, a programming language being developed on top of Proteus for writing portable, MIMD parallel programs [WBC$^+$91]. Prelude provided a model for message dispatching and a mechanism for referencing objects across processors [HBDW91]. To the Prelude mechanism for distributed object references we added a simple structure for creating and managing copies of objects.

### 3.2.1 Interprocessor Messages

In our simulations each processor is executing one thread (one of the processors actually has a second thread, usually inactive, to control the simulation). Each processor has a work queue to hold messages to be processed. The single thread executes a loop, pulling a message off the head of the work queue, dispatching it appropriately to a processing routine, and, when finished processing the message, returning to look for the next message. The finishing of a received message typically involves sending a message to another processor, either as a forwarded operation or a returned result.

Messages are added to the work queue by an interrupt handler that takes messages off of the network.

### 3.2.2 Distributed Objects and References

Every object created in our system has an address on a processor. This address, unique for each object on a specific processor, is used only for local references to the object. For interprocessor references, an object is referred to by an object identifier (OID), that can be translated through

```
typedef struct {
    short status;          /* Object type flags */
    ObjectLock lock;
    Oid  oid;              /* System-wide unique identifier */
    struct locmap *locmap  /* Map of object copy locations */
} ObjectHeader;
```

Figure 3-1: Object Header Data Structure

| Status bit | Name | Values |
|---|---|---|
| 0 | exported | 0 on creation, 1 when exported |
| 1 | surrogate | 0 if the original or copy, 1 if surrogate |
| 2 | master | 1 if original, 0 otherwise |

Figure 3-2: Object Status Bits

an *OID table* to a local address on a processor (if the object exists on that processor). The use of OIDs for interprocessor references allows processors to remap objects in local memory (e.g., for garbage collection) and allows copies of objects to be referenced on different processors.

Every object has an object header, shown in figure 3-1. When a new object is created the object status in the header is initialized to indicate the object has not been exported, is not a surrogate, and is the master, using status bits described in figure 3-2. As long as all references to the object are local, the object header remains as initialized. When a reference to the object is exported to another processor, an object identifier (OID) is created to uniquely identify the object for inter-processor reference. In our implementation the OID is a concatenation of the processor ID and an object serial number. The OID is added to the object's header and the OID/address pair is added to the local OID table. A processor receiving a reference to a remote object will create a surrogate for the object, if one does not already exist, and add an entry to its local OID table. The location map will be described in the next section.

When accessing an object, a remote reference on a processor is initially identical to a local reference – both are addresses of objects. If the object is local the address will be the address of the object itself. If the address is remote, the address is that of a special type of object called a *surrogate*, shown in figure 3-3. The surrogate contains the OID in its header. If an object existed always and only on the processor where it was created, the OID would be enough to find the object. To support replication we use additional fields that are described in the next

```
typedef struct {
    ObjectHeader obj;
    Node location_hint;
    ObjectHeader *local_copy;
} Surrogate;
```

Figure 3-3: Surrogate Data Structure

section.

### 3.2.3 Copies of Objects

The addition of copies of objects requires extension of the object header and surrogate structures. To the object header we expand the status field to include identification of a copy of an object – status neither a surrogate or the master; and we add a location map. A location map will be created only with the master of an object and contains a record of all processors that hold a copy of the object. Only the master copy of an object knows the location of all copies. The copies know only of themselves and, via the OID, the master. We implemented the location map as a bitmap.

Two changes are made to the surrogate structure. First, we add a location hint to indicate where the processor holding a particular surrogate should forward messages for the object, i.e., which copy it should use. Second, we add a pointer to a local copy of the object, if one exists. Since copies are created and deleted over time, a local reference to a copy always passes through a surrogate to assure dangling references will not be left behind. Likewise, as copies are created and deleted, a surrogate may be left on a processor that no longer holds any references to the object. Although it would be possible to garbage collect surrogates, we did not do so in our implementation.

### 3.2.4 Mapping Surrogates to Copies

The purpose of creating copies of an object is to spread the accesses to an object across more than one processor in order to eliminate object and processor bottlenecks. To accomplish this spread, remote accesses to an object must be distributed via its surrogates across its copies, not only to the master copy of the object. As indicated in the previous section, we give each surrogate a single location hint of where a copy might be found (*might*, because the copy may have been deleted since the hint was given).

We do not give each surrogate the same hint, however. To distribute location hints, we first identify all processors that need location hints and all processors that have copies. The set of processors needing hints is divided evenly across the set of processors holding copies, each processor needing a hint being given the location of one copy. In this description we have consciously used the phrase "processor needing a hint" instead of "processor holding a surrogate". In our implementation we did not map all surrogates to the copies, but rather only the surrogates on processors holding copies of the parent B-tree node. It is the downward references from those nodes that we are trying to distribute and balance in the B-tree implementation. Of course, as copies are added or deleted, the mapping of surrogates to copies must be updated. For our implementation, we placed the initiation of remapping under the control of the B-tree algorithm rather than the object management layer.

There are other options for the mapping of surrogates to copies. Each surrogate, for example, could be kept informed of more than one copy location, from two up to all the locations, and be given an algorithm for selecting which location to use on an individual access. In section 7.4 in the chapter on dynamic control of replication, we explore a modification to our approach to mapping that gives each surrogate knowledge of the location of all of its copies.

## 3.3 Additional Extensions to the B-tree Algorithm

On top of these layers we implemented a B-link tree which, because it is distributed, has two features that deserve explanation. First, we defined a B-tree operation to always return its result to the processor that originated the operation, to model the return to the requesting thread. There is relatively little state that must be forwarded with an operation to perform the operation itself; we assume that an application that initiates a B-tree operation has significantly more state and should not be migrated with the operation.

Second, the split of a tree node must be done in stages because the new sibling (and possibly a new parent) will likely be on another processor. We start a split by sending the entries to be moved to the new node along with the request to create the new node. We do not remove those entries from the node being split until a pointer to the sibling has been received back. During the intervening time, lookups may continue to use the node being split, but any modifications must be deferred. We created a deferred task list to hold such requests separately from the work queue.

After a new node is created, the children it inherited are notified of their new parent and the insertion of the new node into its parent is started. A modification to the node that has been deferred may then be restarted.

# Chapter 4

# Queueing Network Model

In this chapter we present a queueing network model to describe and predict the performance of distributed B-trees with replicated tree nodes. A queueing network model will not be as flexible or provide as much detail as the actual execution of B-tree code on our Proteus simulator, but it has two distinct advantages over simulation. First, it provides an understanding of the observed system performance based on the established techniques of queueing network theory. This strengthens our faith in the accuracy and consistency of our simulations[1] and provides us with an analytic tool for understanding the key factors affecting system performance. Second, our analytic model requires significantly less memory and processing time than execution of a simulation. As a result, we can study more systems and larger systems than would be practical using only the parallel processor simulator. We can also study the affects of more efficient implementations without actually building the system.

The queueing network technique we use is Mean Value Analysis (MVA), developed by Reiser and Lavenberg [Rei79b, RL80]. We use variations of this technique to construct two different models for distributed B-tree performance. When there is little or no replication of B-tree nodes, a small number of B-tree nodes (and therefore processors) will be a bottleneck for system throughput. The bottleneck processors must be treated differently than non-bottleneck processors. When there is a large amount of replication, no individual B-tree node or processor will be a bottleneck, and all processors can be treated equivalently. We label the models for these two situations "bottleneck" and "high replication", respectively.

[1]Use of the model actually pointed out a small error in the measurements of some simulations.

In this chapter, we will:

- Introduce the terminology of queueing network theory;
- Review our assumptions about the behavior of B-trees and replication;
- Describe the Mean Value Analysis algorithm and relevant variations; and
- Define our two models of B-tree behavior and operation costs.

In the next chapter we will validate the model by comparing the predictions of the queueing network model with the results of simulation.

## 4.1 Queueing Network Terminology

A queueing network is, not surprisingly, a network of queues. At the heart of a single queue is a *server* or *service center* that can perform a task, for example a bank teller who can complete customer transactions, or more relevant to us, a processor that can execute a program. In a bank and in most computer systems many *customers* are requesting service from a server. They request service at a frequency called the *arrival rate.* It is not uncommon for there to be more than one customer requesting service from a single server at the same time. When this situation occurs, some of the customers must wait in line, *queue*, until the server can turn his, her, or its attention to the customer's request. A server with no customers is called *idle.* The percentage of time that a server is serving customers is its *utilization* ($U$). When working, a server will always work at the same rate, but the demands of customer requests are not always constant, so the *service time* ($S$) required to perform the tasks requested by the customers will vary. Much of *queueing theory* studies the behavior of a single queue given probability distributions for the arrival rates and service times of customers and their tasks.

*Queueing network theory* studies the behavior of collections of queues linked together such that the output of one service center may be directed to the input of one or more other service centers. Customers enter the system, are routed from service center to service center (the path described by *routing probabilities*) and later leave the system. At each center, the customers receive service, possibly after waiting in a queue for other customers to be served ahead of them.

In our case, the service centers are the processors and the communication network connecting them. The communication network that physically connects processors is itself a service center in the model's logical network of service centers. Our customers are B-tree operations. At each step of the descent from B-tree root to leaf, a B-tree operation may need to be forwarded, via the communication network, to the processor holding the next B-tree node. The operation physically moves from service center to service center, requiring service time at each service center it visits. The average number of visits to an individual service center in the course of a single operation is the *visit count* ($V$) and the product of the average service time per visit and the visit count is the *service demand* ($D$) for the center. The sum of the service demands that a single B-tree operation presents to each service center is the *total service demand* for the operation.

In our model the two types of service center, processors and communication network, have different behaviors. The processors are modeled as *queueing* service centers, in which customers are served one at a time on a first-come-first-served basis. A customer arriving at a processor must wait in a queue for the processor to complete servicing any customer that has arrived before it, then spend time being serviced itself. The network is modeled as a *delay* service center: a customer does not queue, but is delayed only for its own service time before reaching its destination. The total time (queued and being served) that a customer waits at a server each visit is the *residence time* ($R$). The total of the residence times for a single B-tree operation is the *response time*. The rate at which operations complete is the throughput ($X$).

In our queueing network model and in our simulations we use a *closed* system model: our system always contains a fixed number of customers and there is no external arrival rate. As soon as one B-tree operation completes, another is started. The alternative model is an *open* system, where the number of customers in the system depends on an external arrival rate of customers.

Within a closed queueing system, there can be a number of *classes*[2] of customers. Each customer class can have its own fixed number of customers and its own service time and visit count requirement for each service center. If each service center has the same service demand requirement for all customers, the customers can placed in a single class. If, however, the service

[2]The term *chain* is also used in some of the literature.

| | |
|---|---|
| Service centers | $K$, the number of service centers. |
| | For each center, $k$, the type, queueing or delay |
| Customers | $C$, the number of classes |
| | $N_c$, the number of customers in each class |
| Service demands | For each class $c$ and center $k$, service demand given by $D_{c,k} \equiv V_{c,k}S_{c,k}$, the average number of visits per operation $*$ the average service time per visit. |

Figure 4-1: Queueing Network Model Inputs

demand requirement for an individual service center varies by customer, multiple customer classes must be used. We will use both single-class and multiple-class models; single-class to model systems with low replication, and multiple-class to model systems with high replication. The necessity for using both types of models is described in the next section.

Queueing network theory focuses primarily on networks that have a *product-form solution*; such networks have a tractable analytic solution. In short, a closed, multi-class queueing network with first-come-first-served queues has a product-form solution if the routing between service centers is Markovian (i.e., depends only on transition probabilities, not any past history) and all classes have the same exponential service time distribution. Most real-world systems to be modeled, including ours, do not meet product-form requirements exactly. However, the techniques for solving product-form networks, with appropriate extensions, have been shown to give accurate results even when product-form requirements are not met [LZGS84, Bar79, HL84, dSeSM89]. Our results indicate the extensions are sufficiently accurate to be useful in understanding our problem.

To use a queueing network model, we must provide the model with a description of the service centers, customer classes, and class service demand requirements. The inputs for the multi-class MVA algorithm are shown in figure 4-1. When solved, the queueing network model produces results for the system and each service center, for the aggregate of all customers and for each class. MVA outputs are shown in figure 4-2. We use these results, particularly throughput and response time, to characterize the performance of a particular configuration and compare performance changes as we change parameters of our model or simulation.

It is important to note that throughput and response time can change significantly when the system workload changes. With a closed system, the workload is determined by the number of

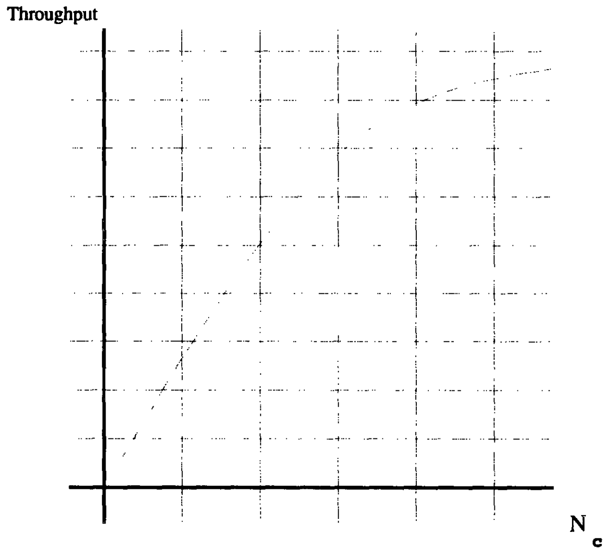
| | |
|---|---|
| Response/Residence time | $R$ for system average, |
| | $R_c$ for class average, |
| | $R_k$ for center residence time, |
| | $R_{c,k}$ for class $c$ residence time at center $k$. |
| Throughput | $X$ for system average, |
| | $X_c$ for class average, |
| | $X_k$ for center average, |
| | $X_{c,k}$ for class $c$ at center $k$. |
| Queue length | $Q$ for system, |
| | $Q_c$ for class, |
| | $Q_k$ for center, |
| | $Q_{c,k}$ for class $c$ at center $k$. |
| Utilization | $U_k$ for centers, |
| | $U_{c,k}$ for class $c$ at center $k$. |

Figure 4-2: Queueing Network Model Outputs

customers in the system, specified by the number of classes, $C$, and the number of customers per class, $N_c$. High throughput can often be bought at the cost of high response time by increasing $N_c$. For some systems, as $N_c$ rises, throughput initially increases with only minor increases in response time. As additional customers are added, the utilization of service centers increases, and the time a customer spends waiting in a queue increases. Eventually throughput levels off while latency increases almost linearly with $N_c$. Figure 4-3 shows this relationship graphically. Thus, while we will primarily study different configurations using their respective throughputs, as we compare across different configurations and as workload changes, we will also compare latencies to make the performance characterization complete.

## 4.2 Modeling B-Trees and Replication

In our use of queueing network theory we make one important assumption: that B-tree nodes and copies are distributed randomly across processors. This means the probability of finding a node on a given processor is $\frac{\#copies}{\#processors}$. Of course, a tree node will actually be on $\#copies$ processors with probability 1.0, and on $(\#processors - \#copies)$ processors with probability 0.0. But the selection of which processors to give copies is random, without any tie to the tree structure as, for example, Johnson and Colbrook [JC92] use in their path-to-root scheme. In our modeling, we assume that all nodes at the same tree level have the same number of copies,



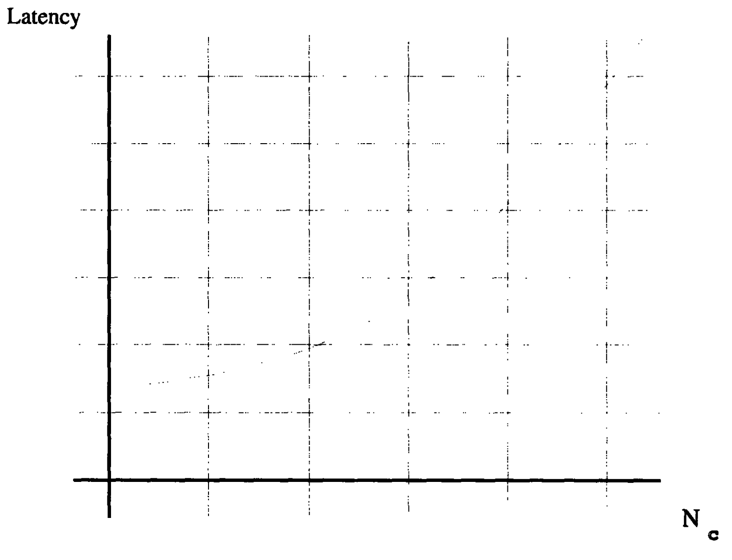


Figure 4-3: Throughput and Latency vs Number of Customers ($N_c$)

and the nodes at a level in a tree are copied to all processors before *any* copies are made at a lower level. In the simulations described in Chapter 6 we will remove this level-at-a-time copying rule and develop rules that, given a fixed, known access pattern, can determine the optimum number of copies to be made for each B-tree node. We will also compare our random placement method with the path-to-root scheme.

In our simulations and in our model, we also assume:

- The distribution of search keys for B-tree operations is uniform and random,
- Processors serve B-tree operations on a first-come-first-served basis,
- The result of an operation is sent back to the originating processor. Even if an operation completes on the originating processor, the result message is still added to the end of the local work queue.

As mentioned in the previous section, we use two different queueing models, one multi-class and one single class. When replication is extensive and there are no bottlenecks, all routing decisions during tree descent are modeled as giving each processor equal probability. The return of a result, however, is always to the processor that originated the operation. Because of this return, each operation has service demands on its "home" processor for operation startup and result handling that it does not have on other processors. If, in the extreme, a B-tree is fully replicated on all processors, a B-tree lookup never has to leave its "home" processor. Because processor service time requirements for an operation depend on which processor originates the operation, we must use a multiple-class model. All operations that originate on a specific processor are in the same class.

When there is little or no replication and one or more processors presents a bottleneck, we will use a single class queueing network model. All operations will be in the same class, but we have three types of service centers, bottleneck processors, non-bottleneck processors and the network. The routing of operations from processor to processor is still modeled as giving each processor equal probability, except that every operation is routed to one of the bottleneck processors for processing of the bottleneck level. We do not explicitly model the return of an operation to its home processor, but this has little impact on the results because overall performance is dominated by the throughput limits of the bottleneck level.

For a given input to the model, we always apply the "high replication" model and only if we see that a level of the tree does not have enough copies to assure participation of all processors do we apply the "bottleneck" model. The lower throughput result of the two models is used as the composite result.

## 4.3 Mean Value Analysis

Reiser and Lavenberg [Rei79b, RL80] have shown that it is possible to compute mean values for queueing network statistics such as queue sizes, waiting times, utilizations, throughputs and latencies for closed, multi-class queueing networks with product-form solution, given the inputs introduced in the previous section. Reiser and Lavenberg originally presented the Mean Value Analysis (MVA) algorithm for computing the exact solutions for product-form networks. ("Exact" refers to the mathematical solution of the equations, not the model's fit to the "real world".) However, because of the time and space required when solving for large networks, they and others [Bar79, CN82] have presented algorithms to approximate the solutions. Of critical importance to our use, the MVA technique has also been extended and shown to provide adequate solutions for some non-product-form networks, using both the exact and approximate algorithms.

In this section we will:

- Describe the single class MVA algorithm;
- Describe MVA extensions required for multi-class systems with non-exponential service times;
- Introduce a simplification of the exact algorithm that is computationally feasible for a class of large, multi-class systems; and
- Describe the approximate MVA algorithm.

The notation used in this section is described as introduced and summarized in appendix B.

### 4.3.1 Single-class Mean Value Analysis

MVA, in its simplest form (single class), relies on three equations, in which $N$ is the number of customers and $K$ is the number of service centers:

1. Little's Law applied to the network to calculate system throughput, $X(N)$, from the mean residence time at each server, $R_k(N)$.

$$X(N) = \frac{N}{\sum_{k=1}^{K} R_k(N)}$$

2. Little's Law applied to each service center to calculate the mean queue length at each server, $Q_k(N)$, from system throughput and mean residence times.

$$Q_k(N) = X(N)R_k(N)$$

3. Service center residence equations to calculate the mean residence time from the mean service demand, $D_k \equiv V_k S_k$ (where $V_k$ is the visit count and $S_K$ is the mean visit service time), and mean queue length at customer arrival, $A_k(N)$.

$$R_k(N) = \begin{cases} D_k & \text{delay center} \\ D_k * (1 + A_k(N)) & \text{queueing center} \end{cases}$$

The innovation of the MVA algorithm was the method for computing $A_k(N)$, the mean queue length at customer arrival. Reiser and Lavenberg [RL80] and Sevcik and Mitrani [SM81] independently proved the arrival theorem that states $A_k(N) = Q_k(N-1)$; that is, the average number of customers seen at a service center on arrival is equal to the steady state queue length with one customer removed from the system. Using this theorem, the exact solution to the queueing network equations starts with a network with no customers (queue lengths are zero) and iteratively applies the three MVA equations, calculating residence times, throughput, and queue lengths, for the system with one task, then two tasks, up to $N$ tasks.

Approximate solutions to the equations use a heuristic to estimate $A_k(N)$ from $Q_k(N)$, rather than compute it exactly from $Q_k(N-1)$. They start with a estimate for $Q_k(N)$ and repeatedly apply the MVA equations until the change in $Q_k(N)$ between iterations is small. The approximate algorithm is described in more detail in section 4.3.4.

### 4.3.2 Extensions to Basic MVA

We must modify the three basic MVA equations to account for three differences between our B-tree model and the basic MVA model (only the third difference applies to our use of the single-class model):

1. Multi-class — Since the result of each B-tree operation will be returned to the originating processor, we define each processor to have its own class. $C$ represents the number of classes; the customer population, $N$, becomes a vector $\overline{N} = (N_1, N_2, ..., N_C)$; and the visit count, $V_k$, becomes $V_{c,k}$. We use the notation $\overline{N - 1_c}$ to indicate the population $\overline{N}$ with one customer of class $c$ removed.

2. Different mean service times per class — Since B-tree operations have a higher service demand on the processor where they originate than on other processors, the mean service time per visit may be different. Mean service time, $S_k$, becomes $S_{c,k}$.

3. Non-exponential distribution of service times — The service demand for each step of a B-tree operation (e.g., checking a B-tree node, preparing the message to forward the operation to another processor, receiving a message from another processor) is modeled as a constant, not exponentially distributed, function. The service time per visit is the

combination of these constant steps. While this will result in a non-constant distribution for the service time per visit, the distribution will not be exponential. This change affects the amount of service time remaining for a customer being served at arrival. We describe the required equation change below.

In response to these three differences, the MVA equations become [Rei79a, LZGS84]:

1. Little's Law applied to the network to calculate system throughput per class from the mean residence time per class at each server.

$$X_c(\overline{N}) = \frac{N_c}{\sum_{k=1}^{K} R_{c,k}(\overline{N})}$$

2. Little's Law applied to each service center to calculate the mean queue length per class at each server from system throughput per class and mean residence times per class.

$$Q_{c,k}(\overline{N}) = X_c(\overline{N}) R_{c,k}(\overline{N})$$

And, summed over all classes, the mean queue length per server:

$$Q_k(\overline{N}) = \sum_{c=1}^{C} Q_{c,k}(\overline{N}) = \sum_{c=1}^{C} X_c(\overline{N}) R_{c,k}(\overline{N})$$

3. Service center residence equations to calculate the mean residence time per class from the mean service requirement and mean queue length at arrival.

For delay centers, becomes:

$$R_{c,k}(\overline{N}) = \begin{cases} V_{c,k} S_{c,k} & \text{if } N_c > 0 \\ 0 & \text{if } N_c = 0 \end{cases}$$

and for queueing centers, becomes:

$$R_{c,k}(\overline{N}) \approx \begin{cases} V_{c,k} * \left( \begin{array}{l} S_{c,k} + \sum_{i=1}^{C} S_{i,k} * \left( Q_{i,k}(\overline{N - 1_c}) - U_{i,k}(\overline{N - 1_c}) \right) + \\ \sum_{j=1}^{C} r_{j,k} U_{j,k}(\overline{N - 1_c}) \end{array} \right) & \text{if } N_c > 0 \\ 0 & \text{if } N_c = 0 \end{cases}$$

where $r_{j,k}$ is the residual service time of the task being served at time of arrival, given by:

$$r_{j,k} = \frac{S_{j,k}}{2} + \frac{\sigma_{j,k}^2}{2S_{j,k}}$$

and $\sigma^2_{j,k}$ is the variance in the service times per visit of class $j$ tasks at processor $k$. Again, $\overline{N - 1_c}$ is the population $\overline{N}$ with one customer from class $c$ removed, and $U_{i,k}(\overline{N})$ is the utilization of processor $k$ by tasks of class $i$, given by $U_{i,k}(\overline{N}) = X_i(\overline{N}) * D_{i,k}$.

For the single-class model with non-exponential distribution of service times, the first two equations remain unchanged from the basic MVA equations, while the service center residence equation for queueing centers becomes:

$$R_k(N) \approx D_k * (1 + Q_k(N-1) - U_k(N-1)) + V_k r_k U_k(N-1)$$

where

$$r_k = \frac{S_k}{2} + \frac{\sigma_k^2}{2S_k}$$

$\sigma_k^2$ is the variance in the service times per visit at processor $k$, and processor utilization $U_k(N) = X(N) * D_k$.

The major change in the appearance of the third equation results from the change to non-exponential service time distributions. In general, residence time has three components: 1) the service time of the arriving customer, 2) the sum of the mean service times of all customers waiting ahead in the queue, and 3) the mean residual service time of the customer being served at arrival. When service times are exponentially distributed, the mean residual service time is the same as the mean service time, $S_{i,k}$. The residence time is then given by $R_{c,k}(\overline{N}) = V_{c,k} * (S_{c,k} + \sum_{i=1}^{C} S_{i,k} Q_{i,k}(\overline{N - 1_c}))$, the number of visits to the center times the sum of the mean service time for the arriving customer and the mean service time for each customer in the queue.

For non-exponential service time we must adjust the MVA treatment of residual service time. We first remove the customer being served from the queue by subtracting the probability that center $k$ is serving a customer of class $i$, $U_{i,k}(\overline{N - 1_c})$, from the arrival queue length. We must then add another term for the time spent waiting on the customer in service, given by the probability that a customer of class $j$ is being served, $U_{j,k}(\overline{N - 1_c})$, times the mean residual service time. The formula for the mean residual service time, $r_{j,k}$, comes from renewal theory (see [Cox62, Kle75]). Note that when the service time distribution is exponential, $\sigma^2_{j,k} = S^2_{j,k}$, so $r_{j,k} = S_{j,k}$, as expected. We delay further discussion of our use of service time variance until section 4.4.4.

One additional comment on residual service time is required. When an operation ends up on a leaf node on its "home" processor, it is "returned" by being added to the local work queue. Since this arrival is not the least bit independent of the current state of the server, it is not a random arrival that will see the average residual service time, and therefore the residual service time must be adjusted accordingly. In this case the residual service time of the task that is just beginning to be served at time of arrival is the same as its mean service time. We calculate a "blended" residual service time based on the probability that an addition to the work queue is the local return of a result.

### 4.3.3 Exact MVA Algorithm and Simplifications

The exact multi-class MVA algorithm, from Lazowska [LZGS84], is shown in Figure 4-4. This algorithm is not generally useful for large systems, as the time and space requirements are proportional to $\mathrm{KC}\prod_{c=1}^{C}(N_c + 1)$. In our work, we use the exact algorithm only when there is one customer per class. Even though this reduces the space requirements of the algorithm to $KC \begin{pmatrix} C \\ C/2 \end{pmatrix}$, it is still not computationally feasible for large systems. When the number of processors in the system is 100 ($C = 100$), for example, the space requirement is still very large, about $10^{33}$. Fortunately, our use of the MVA algorithm does not require its full flexibility and we can simplify the computation, with no change in results, to require a constant amount of space and time proportional to $C$, the number of processors.

Our simplification stems from the recognition that all our customer classes are identical, except for the service center they consider "home". The general MVA algorithm allows visit counts and service times to be specified separately for each class/center pair and, as a result, must calculate and store values for residence time and queue length for each class/center pair, for every possible distribution of $n$ customers. By specifying that all classes (except the "home" class) present the same load to a given processor, we need only calculate residence time and queue length for the "home" class and a representative "other" class. Further, when we restrict ourselves to one customer per class ($N_c = 1$, for all $c$), then from the perspective of each processor there are only two possible distributions of $n$ customers: either all $n$ must be of an "other" class, or one of the "home" class and $n - 1$ "other". All possible arrangements of the "other" customers across processors are identical, so we need calculate only one representative

```
for $k = 1$ to $K$ do $Q_k(\overline{0}) = 0$
for $n = 1$ to $\sum_{c=1}^{C} N_c$ do
    for each feasible population $\overline{n} \equiv (n_1, ..., n_C)$ with $n$ total customers do
    begin
        for $c = 1$ to $C$ do
            for $k = 1$ to $K$ do
                calculate $R_{c,k}(\overline{n})$
        for $c = 1$ to $C$ do
            calculate $X_c(\overline{n})$
        for $k = 1$ to $K$ do
            calculate $Q_{c,k}(\overline{n})$ and $Q_k(\overline{n})$
    end
```

Figure 4-4: Exact MVA Algorithm

| | | Replaced by | |
|---|---|---|---|
| Symbol format | | $c = k$ (Home) | $c \neq k$ (Other) |
| $G_{c,k}$ | | $G_{home}$ | $G_{other}$ |
| $G_{c,k}(\overline{N - 1_c})$ | $N_k = 1$ | $G_{home}$ | $G_{other,yes}$ |
| | $N_k = 0$ | Not Applicable | $G_{other,no}$ |

Figure 4-5: Replacement Rules for MVA Simplification ($k \neq net$)

arrangement. The computation can now be done in time proportional to $C$, using constant space.

The general structure of the algorithm will remain the same, but the simplification changes the intermediate values that must be computed and the equations used to compute them. Because all vectors $\overline{N}$ of $N$ customers are equivalent, we simplify by replacing $\overline{N}$ with $N$. We use $N_k$ to indicate whether or not the customer associated with processor $k$ has been added to the system. Symbols of the form $G_{c,k}$ and $G_{c,k}(\overline{N - 1_c})$ are simplified by explicitly stating the possible relationships between $c$, $k$, and $N_k$. Figure 4-5 shows the replacements used when the server, $k$, is a processor. $G_{other,yes}$ indicates that the customer is at an "other" processor that has had its own customer added to the system. $G_{other,no}$ indicates that the local "home" customer has not yet been added. Since the communication network ($k = net$) is modeled as a single delay server, service time and visit count are always the same for all classes.

Specifically, we can replace throughput, $X_c(\overline{N})$, with $X(N)$, replace mean service time, $S_{c,k}$, with:

$$S_{c,k} = \begin{cases} S_{home} & \text{if } c = k \\ S_{other} & \text{if } c \neq k \text{ and } k \neq net \\ S_{net} & \text{if } k = net \end{cases}$$

replace mean visit count, $V_{c,k}$, with:

$$V_{c,k} = \begin{cases} V_{home} & \text{if } c = k \\ V_{other} & \text{if } c \neq k \text{ and } k \neq net \\ V_{net} & \text{if } k = net \end{cases}$$

replace service time variance, $\sigma^2_{c,k}$, with:

$$\sigma^2_{c,k} = \begin{cases} \sigma^2_{home} & \text{if } c = k \\ \sigma^2_{other} & \text{if } c \neq k \text{ and } k \neq net \\ NA & \text{if } k = net \end{cases}$$

replace mean queue length, $Q_{c,k}(\overline{N})$, with:

$$Q_{c,k}(\overline{N}) = \begin{cases} Q_{home}(N) & \text{if } c = k \text{ and } N_c = 1 \\ Q_{other,yes}(N) & \text{if } c \neq k,\ N_c = 1,\ N_k = 1 \text{ and } k \neq net \\ Q_{other,no}(N) & \text{if } c \neq k,\ N_c = 1,\ N_k = 0 \text{ and } k \neq net \\ Q_{net}(N) & \text{if } k = net \text{ and } N_c = 1 \\ 0 & \text{if } N_c = 0 \end{cases}$$

replace mean residence time, $R_{c,k}(\overline{N})$, with:

$$R_{c,k}(\overline{N}) = \begin{cases} R_{home}(N) & \text{if } c = k \text{ and } N_c = 1 \\ R_{other,yes}(N) & \text{if } c \neq k,\ N_c = 1,\ N_k = 1 \text{ and } k \neq net \\ R_{other,no}(N) & \text{if } c \neq k,\ N_c = 1,\ N_k = 0 \text{ and } k \neq net \\ R_{net}(N) & \text{if } N_c = 1 \text{ and } k = net \\ 0 & \text{if } N_c = 0 \end{cases}$$

and replace mean utilization, $U_{c,k}(\overline{N})$, with:

$$U_{c,k}(\overline{N}) = \begin{cases} U_{home}(N) & \text{if } c = k \text{ and } N_c = 1 \\ U_{other}(N) & \text{if } c \neq k \text{ and } N_c = 1 \text{ and } k \neq net \\ NA & \text{if } k = net \text{ and } N_c = 1 \\ 0 & \text{if } N_c = 0 \end{cases}$$

$U_{c,k}(\overline{N})$ does not expand into $U_{other,yes}$ and $U_{other,no}$ because it is defined as $U_{c,k}(\overline{N}) = X_c(\overline{N}) * D_{c,k}$ and none of the factors on the right side of this equation depend on whether the operation from processor $k$ has been added to the system.

Now, instead of providing $S_{c,k}$, $V_{c,k}$ and $\sigma^2_{c,k}$ for every class/center pair, we need provide only eight values:

| | |
|---|---|
| Service times | $S_{home}$, $S_{other}$, and $S_{net}$ |
| Visit counts | $V_{home}$,$V_{other}$,and $V_{net}$ |
| Service time variances | $\sigma^2_{home}$, and $\sigma^2_{other}$ |

The MVA equations can then be rewritten by substituting these symbols and replacing the summation forms with explicit multiplications and additions.

## Updating Throughput

The throughput of a given class $c$ with $N_c = 1$ and $N$ total operations in the system, is given by:

$$X(N) = \frac{1}{R_{home} + (N-1) * R_{other,yes} + (C-N) * R_{other,no} + R_{net}}$$

Total system throughput with $N$ total operations in the system is:

$$X_{system}(N) = \frac{N}{R_{home} + (N-1) * R_{other,yes} + (C-N) * R_{other,no} + R_{net}}$$

## Updating Mean Queue Lengths

The mean queue lengths, $Q_{c,k}(\overline{N})$, must be specified for four cases, $Q_{home}$, $Q_{other,yes}$, $Q_{other,no}$ and $Q_{net}$. Since the communication network is a single delay server, $Q_{net}$ does not have quite the same interpretation as the queue length for a processor. $Q_{net}$ will give the mean number of tasks in the entire communication network at an instance.

$$Q_{home} = X(N)R_{home}(N)$$

$$Q_{other,yes} = X(N)R_{other,yes}(N)$$

$$Q_{other,no} = X(N)R_{other,no}(N)$$

$$Q_{net} = X(N)R_{net}(N)$$

### Updating Residence Time

With our simplification, $R_{c,k}(\overline{N})$ must be specified for three cases, $R_{home}$, $R_{other,yes}$, and $R_{other,no}$.

For an operation arriving at its home processor, there are $N-1$ operations from other processors in the system, so $R_{home}$ is:

$$\begin{aligned}
&R_{home}(N) = V_{home} * & \\
&\quad (\ S_{home} + & \text{Own service time} \\
&\quad\ S_{other} * (Q_{other,yes}(N-1) - U_{other}(N-1)) * (N-1) + & \text{Service of waiting customers} \\
&\quad\ r_{other} * U_{other}(N-1) * (N-1)) & \text{Residual service time}
\end{aligned}$$

When an operation arrives at an "other" processor that has a customer in the system ($N_k = 1$), the $N$ total operations in the system are the one just arriving, the one whose home is the current processor and $N-2$ that are from other processors. Thus:

$$\begin{aligned}
&R_{other,yes}(N) = V_{other} * & \\
&\quad (\ S_{other} + & \text{Own service time} \\
&\quad\ S_{home} * (Q_{home}(N-1) - U_{home}(N-1)) + & \text{Service of “home” class} \\
&\quad\ r_{home} * U_{home}(N-1) + & \text{Residual time of “home” class} \\
&\quad\ S_{other} * (Q_{other,yes}(N-1) - U_{other}(N-1)) * (N-2) + & \text{Service for other classes} \\
&\quad\ r_{other} * U_{other}(N-1) * (N-2)) & \text{Residual time of other classes}
\end{aligned}$$

Finally, when an operation arrives at an "other" processor that does not have a customer in the system ($N_k \neq 1$), the $N-1$ other operations in the system are all from other processors:

$$\begin{aligned}
&R_{other,no}(N) = V_{other} * & \\
&\quad (\ S_{other} + & \text{Own service time} \\
&\quad\ S_{other} * (Q_{other,no}(N-1) - U_{other}(N-1)) * (N-1) + & \text{Service of other classes} \\
&\quad\ r_{other} * U_{other}(N-1) * (N-1)) & \text{Residual time of other classes}
\end{aligned}$$

## 4.3.4 Approximate MVA Algorithm

When we allow more than one customer per class ($N_c > 1$), the simplification described in the previous section no longer holds. During the iteration up from zero customers, the possible distribution of $n$ customers across the classes becomes more complicated than "$n$ have 1 customer

```
for $k = 1$ to $K$ do
    for $c = 1$ to $C$ do
        $Q_{c,k}(\overline{N}) = N_c/K$
while (TRUE)
    Approximate $Q_{c,k}(\overline{N})$ and $U_{c,k}(\overline{N})$
    Apply MVA equations using approximations
    Compare calculated $Q_{c,k}(\overline{N})$ with previous value, break if within 0.1%
```

Figure 4-6: Approximate MVA Algorithm

each in the system, the rest have no customers." Thus, not all feasible populations $\overline{n}$ of $n$ total customers are equivalent.

Rather than develop a more complicated "simplification" for the equations, we use a simplified algorithm, the approximate MVA algorithm (from Lazowska [LZGS84]) shown in Figure 4-6, and use Schweitzer's method for our approximations.

The algorithm, proposed by Schweitzer and described by Bard [Bar79, Bar80], uses the extended MVA equations described in section 4.3.2, but proceeds by refining an estimate of $Q_{c,k}(\overline{N})$ until successive values are within a specified tolerance. The critical step in this algorithm is the approximation of $Q_{i,k}(\overline{N - 1_c})$ from $Q_{i,k}(\overline{N})$.

The Schweitzer approximation assumes the removal of one customer from the full population affects only the queue lengths of that customer's class, and that it reduces those queue lengths in proportion to the original size:

$$Q_{i,k}(\overline{N - 1_c}) = \begin{cases} \frac{(N_c - 1)}{N_c} Q_{c,k}(\overline{N}) & \text{if } i = c \\ Q_{i,k}(\overline{N}) & \text{if } i \neq c \end{cases}$$

When the service time distribution is non-exponential, we also need an approximation for $U_{i,k}(\overline{N - 1_c})$, the mean utilization of a server, $k$, by a customer class, $c$. It is more difficult to develop a good intuitive approximation for the utilization. When there is only one task per class, the removal of the task will drop utilization to zero. When there are so many tasks per class that a single class has 100% utilization of a processor, the removal of a single task has no effect on utilization. Fortunately, the approximation of utilization has a minor affect on our results. Following Schweitzer's lead, we assume that the removal of a customer from a class affects only the utilization of that customer's class and that it reduces the class utilization in proportion to the original utilization.

$$U_{i,k}(\overline{N - 1_c}) = \begin{cases} \frac{(N_c-1)}{N_c} U_{c,k}(\overline{N}) & \text{if } i = c \\ U_{i,k}(\overline{N}) & \text{if } i \neq c \end{cases}$$

We have also used a more complicated approximation algorithm due to Chandy and Neuse [CN82] and found its results on our application not significantly different from Schweitzer's algorithm.

## 4.4 B-Tree Cost Model – High Replication

To use the MVA algorithms just described to model a distributed B-tree with replicated nodes we must provide the eight parameters mentioned in section 4.3.3: three service times, $S_{home}$, $S_{other}$, and $S_{net}$; three visit counts, $V_{home}$,$V_{other}$, and $V_{net}$; and two service time variances, $\sigma^2_{home}$ and $\sigma^2_{other}$. We calculate these values using knowledge of the configuration of the parallel processor, the shape of the B-tree, and the costs of individual steps of the B-tree algorithm.

From the configuration of the parallel processor we take two values: the number of processors used by the B-tree ($C$) and the average network delay for messages sent between processors (*net_delay*).

We also need to know the shape and size of the B-tree data structure to model its distribution. We use the number of levels in the B-tree (*num_levels*) and the number of B-tree nodes per level (*nodes*[*l*], where $0 \leq l <$ *num_levels* and the leaves are level 0). We model the replication of B-tree nodes by specifying a value, *stay_level*, that indicates the number of levels a B-tree operation can proceed before it may need to move to another processor. The value 0 indicates that no B-tree nodes are replicated, the value 1.75 indicates that the root level is fully replicated and each node on the next level has, in addition to its original, copies on 75% of the remaining processors. If *stay_level* = *num_levels*, the B-tree is fully replicated on all processors. Figure 4-7 depicts these measures.

The basic steps of the B-tree algorithm and their respective cost measures are shown in Figure 4-8. The general behavior of a B-tree is very simple: look at the current B-tree node to find the correct entry and act, forwarding the B-tree operation to a child if at an upper level node, or completing the B-tree operation if at a leaf node. Before any B-tree operation can

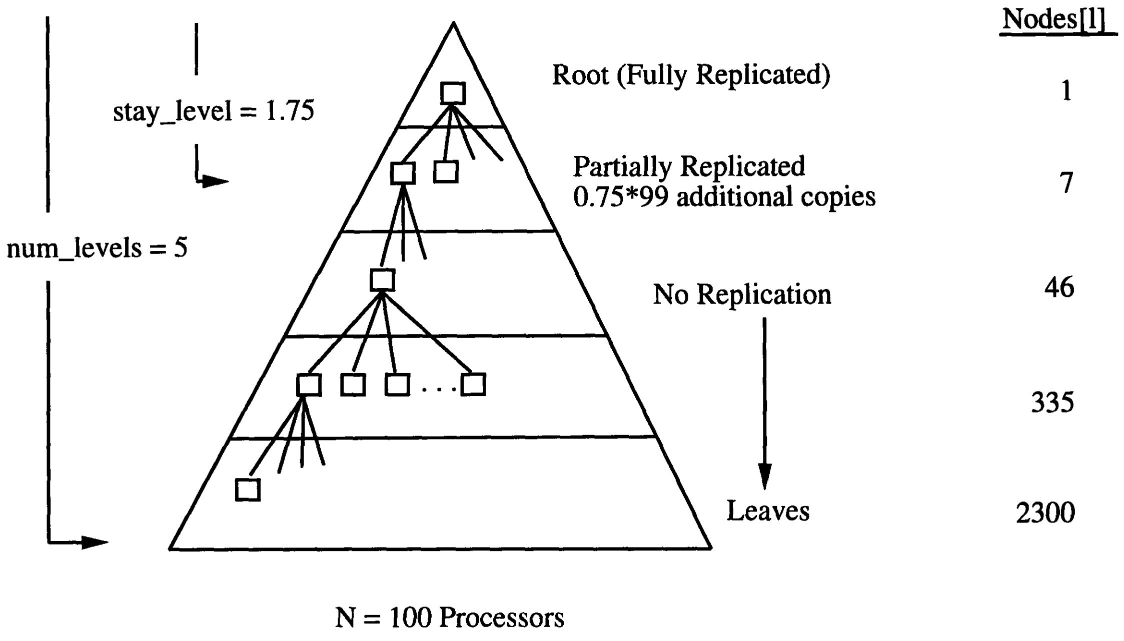


Figure 4-7: B-tree Shape and Replication Parameters

start, however, the "application" thread that will generate the operation must be scheduled and remove a prior result from the work queue. We model this with cost *start_ovhd*. The requesting thread requires time *start_cost* to process the prior result and initiate a new B-tree operation. After starting, the work required at a single upper level node is *node_cost* and the cost of sending a message to a node on another processor is *mesg_ovhd* (this includes all overhead costs, sending and receiving the message, work queue addition and removal, and scheduling overhead). At a leaf node, an operation has cost *leaf_cost*, and, if necessary, sends its result to another processor at cost *result_ovhd*. In section 4.4.3 we will discuss the costs of splitting B-tree nodes and propagating node changes to other copies.

Whenever a message must be sent between processors it is delayed *net_delay* by the communication network. If all work for an operation were done on a single processor, the service demand on that processor would be:

$$\begin{aligned}\text{Service demand} = \;& \mathit{start\_ovhd} + \mathit{start\_cost} + \\ & \mathit{node\_cost} * (\mathit{num\_levels} - 1) + \\ & \mathit{leaf\_cost}\end{aligned}$$

| B-Tree Step | Cost Measure |
|---|---|
| 1. Source thread executes, initiating B-tree operation | *start_cost* |
| 2. If B-tree root is not local, forward operation to root | *mesg_ovhd* |
| While not at leaf: | |
| 3. Find child possibly containing search key | *node_cost* |
| 4. If child is not local, forward operation to child | *mesg_ovhd* |
| When at leaf (lookup operation): | |
| 5. Find entry matching key (if any) | *leaf_cost* |
| 6. If requesting thread is not local, send result to source processor | *result_ovhd* |
| When at leaf (insert operation): | |
| 5. Find correct entry and insert key, splitting node if necessary | see section 4.4.3 |
| 6. If requesting thread is not local, send result to source processor | *result_ovhd* |
| When at leaf (delete operation): | |
| 5. Find entry matching key (if any) and remove entry | see section 4.4.3 |
| 6. If requesting thread is not local, send result to source processor | *result_ovhd* |
| 7. Restart source thread to read and process result | *start_ovhd* |
| For any message sent between processors | *net_delay* |

Figure 4-8: B-Tree Steps and Cost Measures

For all other processors, the service demand would be zero.

In general, however, B-tree operations will require messages between processors and encounter queueing delays. The service demands ($D_{c,k}$) and visit counts ($V_{c,k}$) an operation presents to each processor can be calculated using the probability of finding a copy of the next desired B-tree node on a specific processor. We then use the formula $S_{c,k} = D_{c,k}/V_{c,k}$ to yield mean service times. In the following sections we describe the computation of visit counts and service demands for B-trees with only lookup operations, then discuss the implications of adding insert and delete operations, and, finally, the computation of service time variances.

### 4.4.1 Calculating Visit Counts

We define a B-tree operation to have visited a processor whenever it is added to the work queue of the processor. This includes the arrival of an operation forwarded from another processor while descending the B-tree, and the addition of a result to the work queue of the processor that originated the operation, regardless of whether the operation reached a leaf on the "home" processor or another processor. An operation visits the network when the operation must be sent to another processor while descending or returning. The visit counts are calculated from the probabilities of these events.

### Processor Visit Count

In this section, we use $C$ to denote the number of processors and $\alpha$ to denote the fractional part of *stay_level*, the percentage of copies made on the partially replicated level.

An operation always visits its home processor at least once, at the return of a result/start of the next operation. For every level of the tree the probability of visiting a processor is:

$$P_{visit} = P_{away} * P_{move} * P_{here}$$

When there is no replication of B-tree nodes, $P_{away}$, the probability of having been on any other processor before this B-tree level, is:

$$P_{away} = 1 - \frac{1}{C} = \frac{C-1}{C}$$

and $P_{move}$, the probability that the operation must leave the processor where it is currently located, is:

$$P_{move} = 1 - \frac{1}{C} = \frac{C-1}{C}$$

and $P_{here}$, the probability of moving to a particular processor given the operation will leave its current processor, is:

$$P_{here} = \frac{1}{C-1}$$

As the B-tree is partially replicated these probabilities change. To calculate the new probabilities, we divide the B-tree levels into four sections: the fully replicated levels, the partially replicated level, the first level below partial replication, and the remaining levels below the partial replication. Figures 4-9 and 4-10 show the calculations of the probability of visiting "home" and "other" processors in each sections. These calculations also make use of:

- When an operation reaches the partially replicated level, it will stay on the current processor with probability $P_{stay} = \frac{\#ofcopies}{C}$, where $\#ofcopies = 1 + (C-1) * \alpha$. It will move to a non-home processor with probability:

$$\begin{aligned} P_{move} = 1 - P_{stay} &= 1 - \frac{1 + (C-1) * \alpha}{C} \\ &= \frac{C - 1 - (C-1) * \alpha}{C} \\ &= \frac{(C-1)(1-\alpha)}{C} \end{aligned}$$

| | $P_{away}$ | $*$ | $P_{move}$ | $*$ | $P_{here}$ | $=$ | $P_{visit}$ |
|---|---|---|---|---|---|---|---|
| Start/Fully Replicated | $1$ | | $1$ | | $1$ | $=$ | $1$ |
| Partially Replicated | $0$ | | $\frac{(C-1)(1-\alpha)}{C}$ | | $0$ | $=$ | $0$ |
| First Non-replicated | $\frac{(C-1)(1-\alpha)}{C}$ | | $\frac{C-1}{C}$ | | $\frac{1}{C-1}$ | $=$ | $\frac{(C-1)(1-\alpha)}{C^2}$ |
| Remainder | $\frac{C-1}{C}$ | | $\frac{C-1}{C}$ | | $\frac{1}{C-1}$ | $=$ | $\frac{C-1}{C^2}$ |

Figure 4-9: Probability of Visit ($P_{visit}$) – Home Processor

| | $P_{away}$ | $*$ | $P_{move}$ | $*$ | $P_{here}$ | $=$ | $P_{visit}$ |
|---|---|---|---|---|---|---|---|
| Start/Fully Replicated | $1$ | $*$ | $0$ | $*$ | $0$ | $=$ | $0$ |
| Partially Replicated | $1$ | $*$ | $\frac{(C-1)(1-\alpha)}{C}$ | $*$ | $\frac{1}{C-1}$ | $=$ | $\frac{1-\alpha}{C}$ |
| First Non-replicated | $1-\frac{1-\alpha}{C}$ | $*$ | $\frac{C-1}{C}$ | $*$ | $\frac{1}{C-1}$ | $=$ | $\frac{C-1+\alpha}{C^2}$ |
| Remainder | $\frac{C-1}{C}$ | $*$ | $\frac{C-1}{C}$ | $*$ | $\frac{1}{C-1}$ | $=$ | $\frac{C-1}{C^2}$ |

Figure 4-10: Probability of Visit ($P_{visit}$) – Other Processors

- This same value, $\frac{(C-1)(1-\alpha)}{C}$, is also the probability that the operation will be away from the home processor ($P_{away}$) when it reaches the first tree level after partial replication.
- The probability $P_{away}$ that the operation will not be on a specific "other" processor when the operation encounters the first level of no replication is derived from the probability that the operation visited the processor for the prior level:

$$P_{away} = 1 - P_{visit} = 1 - \frac{1-\alpha}{C}$$

To combine these pieces, we note that for a B-tree with *num_levels* and *stay_level*, there will be $num\_levels - 2 - \lfloor stay\_level \rfloor$ levels below the first non-replicated level and that $stay\_level \equiv \lfloor stay\_level \rfloor + \alpha$.

For the home processor, the result is:

$$\begin{aligned} V_{home} &= 1 + \frac{(C-1)(1-\alpha)}{C^2} + \frac{C-1}{C^2} * (num\_levels - 2 - \lfloor stay\_level \rfloor) \\ &= 1 + \frac{C-1}{C^2} * (1 - \alpha + num\_levels - 2 - \lfloor stay\_level \rfloor) \\ &= 1 + \frac{C-1}{C^2} * (num\_levels - 1 - stay\_level) \end{aligned}$$

When $num\_levels - stay\_level \leq 1$, an operation starts on the home processor and does not visit again until the next start, so the visit count is:

$$V_{home} = 1$$

For the "other" processors, the result is:

$$\begin{aligned}
V_{other} &= \frac{1-\alpha}{C} + \frac{C-1+\alpha}{C^2} + \frac{C-1}{C^2} * (num\_levels - 2 - \lfloor stay\_level \rfloor) \\
&= \frac{1}{C} - \frac{\alpha}{C} + \frac{C-1+\alpha}{C^2} + \frac{C-1}{C^2} * (num\_levels - 2 - \lfloor stay\_level \rfloor) \\
&= \frac{1}{C} + \frac{C-1+\alpha-C*\alpha}{C^2} + \frac{C-1}{C^2} * (num\_levels - 2 - \lfloor stay\_level \rfloor) \\
&= \frac{1}{C} + \frac{(C-1)(1-\alpha)}{C^2} + \frac{C-1}{C^2} * (num\_levels - 2 - \lfloor stay\_level \rfloor) \\
&= \frac{1}{C} + \frac{C-1}{C^2} * (1 - \alpha + num\_levels - 2 - \lfloor stay\_level \rfloor) \\
&= \frac{1}{C} + \frac{C-1}{C^2} * (num\_levels - 1 - stay\_level)
\end{aligned}$$

Similarly, when $num\_levels - stay\_level \leq 1$, the visit ratio for an "other" processor is:

$$V_{other} = \frac{1}{C} * (num\_levels - stay\_level)$$

Note that for all values of $num\_levels - stay\_level$, $V_{home} + (C-1)V_{other}$ does equal the expected number of visits, $1 + \frac{C-1}{C} * (num\_levels - stay\_level)$.

### Network Visit Count

When $num\_levels - stay\_level \geq 1$, the operation may possibly visit the network $num\_levels - stay\_level$ times while descending the tree and once when being sent back to the requesting processor. For each possible visit, the probability of visiting the network is $P_{away} = \frac{C-1}{C}$. Thus, the total network visit count is:

$$V_{net} = \frac{C-1}{C} * (num\_levels - stay\_level + 1)$$

When $num\_levels - stay\_level \leq 1$, an operation can be sent to another processor for at most one step and then is immediately sent back to the "home" processor, so the total network visit count is:

$$V_{net} = 2 * \frac{C-1}{C} * (num\_levels - stay\_level)$$

Note that when $num\_levels - stay\_level = 1$, both equations show the total network visit count is:

$$V_{net} = 2 * \frac{C-1}{C}$$

### 4.4.2 Calculating Service Demand

The service demand on a processor has three components: time spent directly on the B-tree operations (productive work); time spent forwarding and receiving B-tree operations, maintaining the work queues, and scheduling threads (overhead); and time spent propagating B-tree modifications to copies of B-tree nodes (update overhead).

In this section we will calculate the service demand from productive work and message overhead. In section 4.4.3 we will calculate the service demand due to updates.

#### Productive Work on Home Processor

The calculation of productive work itself has three components:

- Operation startup, only on "home" processor,
- Intermediate node processing (all non-leaf nodes),
- Leaf node processing.

If $num\_levels - stay\_level \geq 1$, work done on "home" processor is:

$D_{work,home} =$

| | |
|---|---|
| $start\_cost+$ | Start of B-tree operation |
| $node\_cost * \lfloor stay\_level \rfloor +$ | Intermediate nodes above *stay_level* |
| $node\_cost * \frac{1+\alpha*(C-1)}{C} +$ | Intermediate nodes at *stay_level* |
| $node\_cost * (num\_levels - \lceil stay\_level \rceil - 1) * \frac{1}{C}+$ | Intermediate nodes below *stay_level* |
| $leaf\_cost/C$ | Leaf node |

If $num\_levels - stay\_level \leq 1$ Work done on "home" processor is:

| | | |
|---|---|---|
| $D_{work,home} =$ | $start\_cost+$ | Start of B-tree operation |
| | $node\_cost * (num\_levels - 1)+$ | Intermediate nodes |
| | $leaf\_cost * \alpha * \frac{C-1}{C}+$ | Leaf node (copies on "home") |
| | $leaf\_cost * \frac{1}{C}$ | Leaf node (original on "home") |

Note that if $num\_levels - stay\_level = 1$, both equations evaluate to:

$$\begin{aligned} D_{work,home} = \; & start\_cost + && \text{Start of B-tree operation} \\ & (num\_levels - 1) * node\_cost + && \text{Intermediate nodes} \\ & leaf\_cost / C && \text{Leaf node} \end{aligned}$$

### Productive Work on Other Processors

If $num\_levels - stay\_level \geq 1$, work done on an "other" processor is:

$$\begin{aligned} D_{work,other} = \; & node\_cost * (num\_levels - stay\_level - 1) * \tfrac{1}{C} + && \text{Intermediate nodes} \\ & leaf\_cost * \tfrac{1}{C} && \text{Leaf node} \end{aligned}$$

If $num\_levels - stay\_level \leq 1$, work done on an "other" processor can only be at a leaf node:

$$D_{work,other} = \; leaf\_cost * (num\_levels - stay\_level) * \tfrac{1}{C}$$

Note that if $num\_levels - stay\_level = 1$, both equations evaluate to:

$$D_{work,other} = \; leaf\_cost / C$$

In addition, note that for all values of $stay\_level$,

$$\begin{aligned} D_{work} &= D_{work,home} + (C - 1) * D_{work,other} \\ &= start\_cost + node\_cost * (num\_levels - 1) + leaf\_cost \end{aligned}$$

### Message Overhead on Home Processors

The calculation of message overhead also has three components:

1. Start up overhead from adding a return result to the queue and re-scheduling the requesting thread.
2. Forwarding a B-tree operation to a different processor
3. Returning a B-tree operation result to a different processor

If $num\_levels - stay\_level \geq 1$, overhead on the "home" processor is:

$$\begin{array}{ll}
D_{overhead,home} = & \\
\quad start\_ovhd + & \text{Start up} \\
\quad mesg\_ovhd * \frac{(C-1)(1-\alpha)}{C} + & \text{Forwarding at } stay\_level \\
\quad mesg\_ovhd * \frac{1+\alpha*(C-1)}{C} * P_{away} + & \text{Below } stay\_level \\
\quad mesg\_ovhd * P_{away} * (num\_levels - \lceil stay\_level \rceil - 1) * \frac{1}{C} & \text{Rest of tree}
\end{array}$$

If $num\_levels - stay\_level \leq 1$, overhead on the "home" processor is:

$$\begin{array}{rll}
D_{overhead,home} = & start\_ovhd + & \text{Start up} \\
& mesg\_ovhd * \frac{(C-1)(1-\alpha)}{C} & \text{Forwarding}
\end{array}$$

When $num\_levels - stay\_level = 1$, overhead on the "home" processor is:

$$\begin{array}{rll}
D_{overhead,home} = & start\_ovhd + & \text{Start up} \\
& mesg\_ovhd * \frac{C-1}{C} & \text{Forwarding}
\end{array}$$

**Message Overhead on Other Processors**

If $num\_levels - stay\_level \geq 1$, overhead on an "other" processor is:

$$\begin{array}{rll}
D_{overhead,other} = & mesg\_ovhd * P_{away} * (num\_levels - stay\_level - 1) * \frac{1}{C} + & \text{Forwarding} \\
& result\_ovhd * \frac{1}{C} & \text{Return}
\end{array}$$

If $num\_levels - stay\_level \leq 1$, overhead on an "other" processor can only be associated with returning a result, and is:

$$D_{overhead,other} = result\_ovhd * (num\_levels - stay\_level) * \frac{1}{C}$$

When $num\_levels - stay\_level = 1$, overhead on an "other" processor is:

$$D_{overhead,other} = result\_ovhd * \frac{1}{C}$$

### 4.4.3 Calculating Insert and Delete Costs

The model we have presented so far considers only the costs of B-tree lookup operations. In this section we consider the additional costs due to inserts and deletes. These operations begin just like a lookup, descending the B-tree from the anchor on the "home" processor to find the leaf node that may contain the key. An insert operation adds an entry into a leaf node (if the key is not already there) and may cause a leaf node to split. When a node splits, a new entry is inserted in the parent node, which itself may have to split. For an unreplicated B-tree node, we must model the cost of a node insertion and a node split. When a B-tree node is replicated, we must also model the cost of updating each of the copies of the node. If a delete operation finds the key in the tree, it removes the entry. Since our implementation does not eliminate empty leaf nodes, a delete has no effect on any node but the leaves. Thus, we need only model the cost of the delete itself and the cost of updating other copies of the node.

We assign the direct modification of a leaf node the same cost, whether the modification is an insertion or a deletion, indicated by $modify\_cost$. For splits and updates we introduce two new cost measures each, one for leaf nodes and one for intermediate nodes. These costs are indicated by, $leaf\_split\_cost$,$int\_split\_cost$, $leaf\_update\_cost$ and $int\_update\_cost$. The split cost measures represent the splitting of the original node, creation of a new node, and the message send and receive overhead to accomplish the creation. The update cost measures represent the update of a single copy, including the message costs.

When a node splits, there is only work on the birth processors of the node being split and the new node. When a copy is updated, there is work on all processors that hold copies. Despite the fact that split and update costs are spread across many processors, we allocate the full cost of updates to the "home" processor of an operation. The split and update costs of a single operation are not solely borne by its "home" processor, but since all processors are "home" for some operations, this method does distribute costs evenly. We do not change visit counts to reflect any messages that are sent to accomplish the update as these are all done in the background. As a result, we cannot compute the latency of the background update, only the effect it has on system throughput and response time of the lookup operations.

In our queueing theory model and in our simulations, we control the mix of operations with two variables: $mod\_pct$, the percentage of operations that are modifications, and $del\_pct$, the percentage of modify operations that are deletes. Thus, the probability of each type of operation is:

- $P_{lookup} = 1 - mod\_pct$
- $P_{insert} = mod\_pct * (1 - del\_pct)$

- $P_{delete} = mod\_pct * del\_pct$

We use the work of Johnson and Shasha [JS89] to convert the operation mix into the expected number of node modifications and node splits per operation. Johnson and Shasha suggest that the probability of splitting a leaf node on an insert, given only inserts and deletes, is:

$$P_{split,leaf} = (1 - 2 * del\_pct)/((1 - del\_pct) * branch * space\_util)$$

where *branch* is the branch factor of the B-tree and *space_util* is the average utilization of the B-tree nodes. This equation has intuitive justification. When $del\_pct = .5$, $P_{split,leaf} = 0$, suggesting that when we insert and delete at the same rate, no leaf should ever split. When $del\_pct = 0$, $P_{split,leaf} = 1/(branch * space\_util)$, which suggests that a node must split every $branch * space\_util$ inserts to keep the space utilization constant. Johnson and Shasha, as well as others, have shown that space utilization will be roughly constant at approximately 70%, dropping to approximately 40% when inserts and deletes are equally likely.

Since we do not merge empty nodes, the probability of a deletion in an upper level node is zero, and thus the probability of a split on an insertion in an upper level node is:

$$P_{split,upper} = 1/(branch * space\_util)$$

We define $P_{mod}(i)$ to be the probability of an operation modifying a node at level $i$ (leaves are level 0) and $P_{split}(i)$ to be the probability of an operation causing a split at level $i$. Since all insert and deletes modify a leaf node, we know that $P_{mod}(0) = mod\_pct$. Above the leaf, the rate of inserts is the rate of splits at the next lower level, $P_{mod}(i) = P_{split}(i-1)$. So,

$$P_{mod}(i) = \begin{cases} mod\_pct & \text{for } i = 0 \\ mod\_pct * (1 - del\_pct) * P_{split,leaf} * P^{i-1}_{split,upper} & \text{for } i > 0 \end{cases}$$

$$P_{split}(i) = mod\_pct * (1 - del\_pct) * P_{split,leaf} * P^{i}_{split,upper}$$

The cost of updating a node is proportional to the number of copies of the node (given by $Copies(i)$), so the average cost per operation due to updates is:

$$D_{mod} = \sum_{i=0}^{levels-1} Update\_cost(i) * P_{mod}(i) Copies(i)$$

where

$$Copies(i) = \begin{cases} C-1 & \text{if } i > \lceil (num\_levels - stay\_level) \rceil \\ 0 & \text{if } i < \lfloor (num\_levels - stay\_level) \rfloor \\ (C-1) * frac(stay\_level) & \text{otherwise} \end{cases}$$

$$Update\_cost(i) = \begin{cases} leaf\_update\_cost & \text{if } i = 0 \\ int\_update\_cost & \text{if } i > 0 \end{cases}$$

and where $frac(x)$ is the fractional part of $x$, e.g., $frac(1.75) = .75$.

Similarly, the cost of splitting a node is:

$$\begin{aligned} D_{split} = & \textstyle\sum_{i=0}^{levels-1} Split\_cost(i) * P_{split}(i) + & \text{Split} \\ & \textstyle\sum_{i=0}^{levels-1} Update\_cost * P_{split}(i) Copies(i) & \text{Make new copies} \end{aligned}$$

where

$$Split\_cost(i) = \begin{cases} leaf\_split\_cost & \text{if } i = 0 \\ int\_split\_cost & \text{if } i > 0 \end{cases}$$

The total cost associated with insert and delete operations is:

$$D_{update} = D_{mod} + D_{split}$$

### 4.4.4 Calculating Service Time Variances

We do not actually calculate variance, but run the model assuming both fixed and exponential distributions of service time. When service time is fixed, the variance is zero. When service time is exponentially distributed, the variance is the square of the mean service time, $S^2$.

## 4.5 B-Tree Cost Model – Bottleneck

When replication is relatively low, we use a single class model to study the behavior of the bottleneck. For the single class model we need only calculate the total service demands for each of the three types of service centers, $D_{bottleneck}$, $D_{other}$ and $D_{net}$. Earlier, we described the MVA formula for residence time as:

$$R_k(N) \approx D_k * (1 + Q_k(N-1) - U_k(N-1)) + V_k r_k U_k(N-1)$$

where

$$r_k = \frac{S_k}{2} + \frac{\sigma_k^2}{2S_k}$$

$\sigma_k^2$ is the variance in the service times per visit at processor $k$, and processor utilization $U_k(N) = X(N) * D_k$. This suggests that we need visit counts, service times, and variances for the three service centers. However, since we restrict ourselves to fixed and exponential service time distributions, we need only $D_k$ for the term $V_k r_k U_k(N-1)$ reduces to:

$$V_k r_k U_k(N-1) = \begin{cases} V_k \frac{S_k}{2} U_k(N-1) = \frac{D_k}{2} U_k(N-1) & \text{Fixed Distribution} \\ V_k * (\frac{S_k}{2} + \frac{S_k^2}{2S_k}) * U_k(N-1) = D_k U_k(N-1) & \text{Exponential Distribution} \end{cases}$$

The first step to calculating the performance of a distributed B-tree experiencing a bottleneck is identification of the position and size of the bottleneck. When increasing replication from the top down, completely replicating one level of the B-tree before creating any copies at any lower level, a bottleneck will occur either at the level being replicated or at the level just below it. For example, when no nodes are replicated, the root node is a bottleneck. But once the number of copies of the root exceeds the number of copies of nodes at the next level, that next level becomes the bottleneck. The bottleneck level is the tree level that has the smallest total number of original nodes and copies. If the size of the level with the smallest total number of nodes and copies is greater than the number of processors, the system is assumed not to have a bottleneck.

For simplicity, we assume that all tree nodes and copies of tree nodes at the bottleneck level have been placed on different processors and that each is experiencing an equal load. Thus, if the bottleneck level has 7 B-tree nodes and has 3 additional copies of each node, 28 processors form the bottleneck. Of course, using random placement of copies there is nothing to guarantee that the 28 total copies will all be on different processors. If they are not, the bottleneck will be formed on fewer processors and the load might not be evenly distributed across the processors. As a result, our bottleneck estimates will generally appear slightly higher than simulation results.

When the location and size of the bottleneck has been determined, the service demands can be calculated. As is shown in Figure 4-11, the total operation service demand on the processors can be broken into three components: service demand before the bottleneck level, service time at the bottleneck level, and service time below the bottleneck level. The first

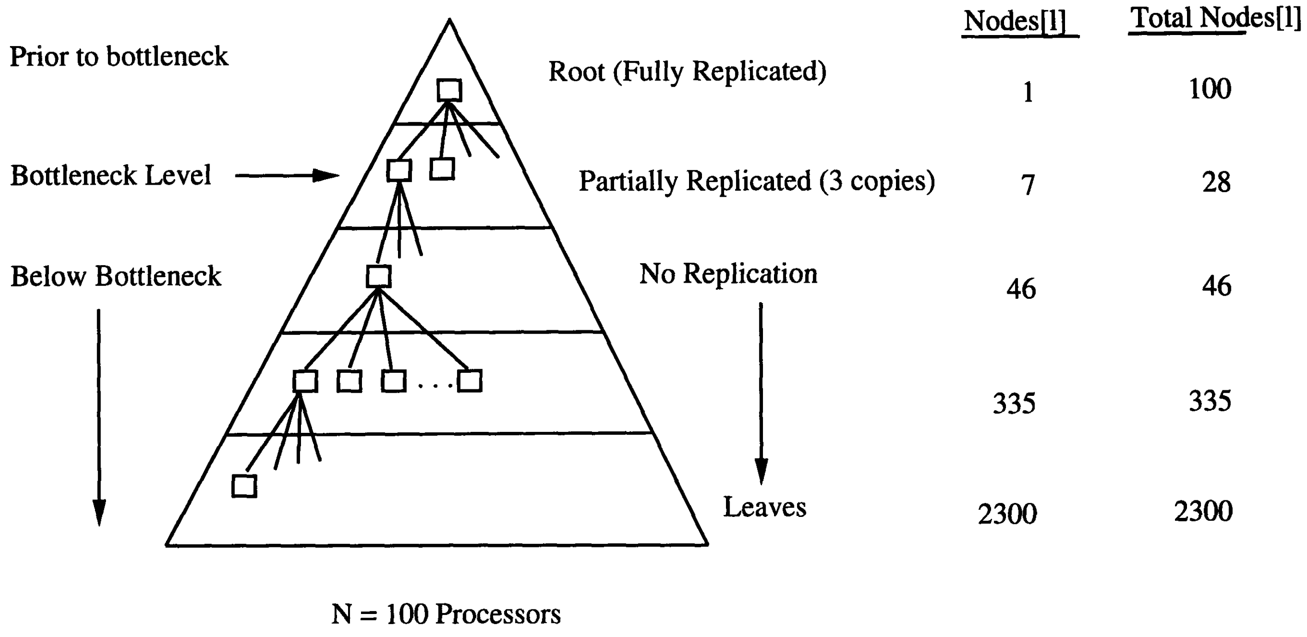


Figure 4-11: Partially Replicated B-tree with Bottleneck

and third components are divided equally across all processors, but the second component is allocated only to the bottleneck processors. With total number of processors $C$ and the size of the bottleneck $B$:

$$D_{bottleneck} = (startup + prior\_cost + lower\_cost)/C + bottle\_cost/B$$

$$D_{other} = (startup + prior\_cost + lower\_cost)/C$$

$$D_{net} = net\_cost$$

Two elements of cost have multiple components:

$$net\_cost = net\_cost_1 + net\_cost_2 + net\_cost_3 + net\_cost_4$$

$$prior\_cost = prior\_cost_1 + prior\_cost_2$$

The service demands prior to the bottleneck level are:

$$startup = start\_cost + start\_ovhd$$

$$prior\_cost_1 = proc\_time * (prior\_levels) + ovhd\_time * \frac{(C-1)(1-\alpha)}{C}$$

$$net\_cost_1 = net\_time * \frac{(C-1)(1-\alpha)}{C}$$

If the bottleneck is below the level being replicated, there is an additional prior cost for the possible forward to the bottleneck level:

$$prior\_cost_2 = ovhd\_time * \frac{C-1}{C}$$

$$net\_cost_2 = net\_time * \frac{C-1}{C}$$

At the bottleneck level itself, there are two possibilities:

$$bottle\_cost = \begin{cases} proc\_time + ovhd\_time * (C-1)/C & \text{Bottleneck at intermediate level} \\ last\_time + return\_ovhd * \frac{(C-1)(1-\alpha)}{C} & \text{Bottleneck at leaves} \end{cases}$$

The network cost that is associated with leaving the bottleneck level is:

$$net\_cost_3 = \begin{cases} net\_time * (C-1)/C & \text{Bottleneck at intermediate level} \\ net\_time * \frac{(C-1)(1-\alpha)}{C} & \text{Bottleneck at leaves} \end{cases}$$

The service demand below the bottleneck is:

$$\begin{aligned} lower\_cost &= (proc\_time + ovhd\_time * \frac{C-1}{C}) * (levels\_below - 1) \\ &\quad + last\_time + return\_ovhd * \frac{C-1}{C} \end{aligned}$$

$$net\_cost_4 = net\_time * \frac{C-1}{C} * (levels\_below)$$

## 4.6 Summary

In this chapter we have presented:

- An explanation of the queueing network theory technique, mean value analysis, we use in this thesis,
- A modification of the MVA algorithms to apply them efficiently to our problem,

- Two models of B-tree behavior and operation cost to predict B-tree performance using MVA techniques.

In the next chapter we compare the results of our queueing network model with the results of simulation and begin to discuss the implications for using replication with distributed B-trees.

# Chapter 5

# Queueing Model Validation

We validate our queueing network model by comparing the results of B-tree simulations with the predictions of the queueing theory model. The Proteus simulator and B-tree code were modified to measure and record the time spent in the phases of a B-tree operation that were described in Chapter 4. Every simulation produces the average and standard deviation for each phase, as well as the latency of individual B-tree operations and the total system throughput. The measured costs of operation phases can be fed into our queueing theory model to produce estimates of latency and throughput for comparison.

In this section we will demonstrate the accuracy of the queueing network model by comparing simulation and queueing model results for a variety of system configurations. We first present a "base case", then variations on the base by changing seven characteristics of the system:

1. The tree size, given by the number of entries and the branching factor,
2. The number of processors,
3. Message sending/receiving overhead,
4. Network transmission delay,
5. Application turn-around time between operations,
6. The number of operations active in the system,

7. The operation mix (lookups/inserts/deletes).

The simulations in this section are run in two parts. First, a B-tree is constructed with no replication. Then replication is gradually increased, fully replicating a level of the tree before doing any partial replication of a lower level. Each time replication is increased, the processors are instructed to initiate a series of B-tree operations, measuring system performance (average system throughput and average operation latency) and the cost of the elements of processing along the way. Each experiment is repeated 5 times, with a different seed for the random number generator, to produce variations in tree layout and access pattern. (For example, in most of our simulations the level below the root consists of seven nodes, but in some simulations this level consists of six or eight nodes.)

After simulations are completed, the observed costs for the elements of B-tree processing are fed into the queueing theory model to produce predictions for performance. We run the model assuming both fixed service time per visit and an exponential distribution. For each test case we provide a graph of throughput vs. replication for the experimental mean of the simulations and the results of the fixed and exponential distribution models. Because the trees created by the 5 test runs vary slightly in size and shape, we plot the mean of the experimental throughputs versus the mean of the replications for the series of operations performed using the same number of copies per node. For example, we group together the 5 results made with 100 copies of the root, 6 copies of each node one level below the root, and 1 copy (the original) of all other nodes. For the test cases where comparison with the base case is relevant, we also include on the graph the results of the queueing network model for the base case using a fixed service time distribution.

## 5.1 Base Case Simulation

In the base case we use a Proteus model of a 128 processor N-ary K-cube. We use 100 of the processors for the simulation. The B-tree structure is defined to have a branch factor of 10, and we perform 2400 insert operations. This produces a tree of depth 4, with each B-tree node approximately 70% full. During simulation, the replication at each tree level increases the total copies of a node (original plus copies) in the following progression: 1, 2, 3, 5, 10, 25, 50, 75, 90, 95, 100. The measured costs of each phase of B-tree lookups are shown in figure 5-1.

| Phase | Cost |
|---|---|
| start_cost | 273 |
| start_ovhd | 47 |
| node_cost | 89 |
| leaf_ cost | 89 |
| result_ovhd | 506 |
| mesg_ovhd | 508 |
| net_delay | 17 |
| modify_cost | 0 |
| leaf_split_cost | 0 |
| int_split_cost | 0 |
| leaf_update_cost | 0 |
| int_update_cost | 0 |

Figure 5-1: Baseline Costs

Figure 5-2 shows the throughput predicted by the queueing model results and observed in the simulations. The queueing model results closely match the simulated results. The shape of the graphs for the base case is common to other test cases. The sharp transitions that occur at about 1,000 and 6,000 copies mark the full replication of one level and start of replication of the next lower level. At such transitions the slope of the curve decreases, indicating that the marginal increase in throughput per copy added is decreasing. As replication is increased within each of the lowest two levels of the tree, figure 5-2 shows that the marginal value of an additional copy is increasing, a phenomenon that will be discussed in the next chapter.

Figure 5-3 shows the base case results, expanding the area of low replication. The left side clearly shows the existence of a bottleneck as the root is replicated. Throughput increases only marginally as replication increases from less than 10 to just below 100 additional nodes. When replication is started at the second level of the B-tree, over 100 copies, throughput increases rapidly. Figure 5-3 also shows that throughput from the simulations remains close to the throughput predicted by the model. The greatest discrepancy arises at low replication, between 10 and 100 additional nodes. Our model assumes that the bottleneck B-tree nodes are each on a separate processor. In the simulations that differ from the model, two or more nodes end up being placed on the same processor. The size of the bottleneck then decreases (and the demand on each bottleneck processor increases), yielding the lower throughput shown by the simulation results.

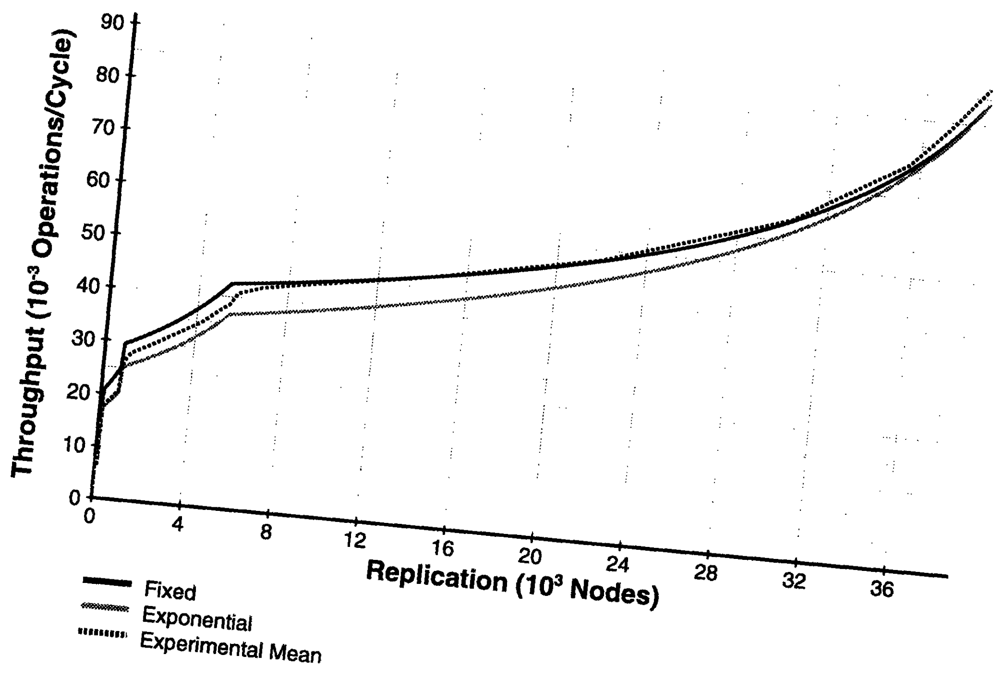


Figure 5-2: Throughput vs. Replication - Base Case

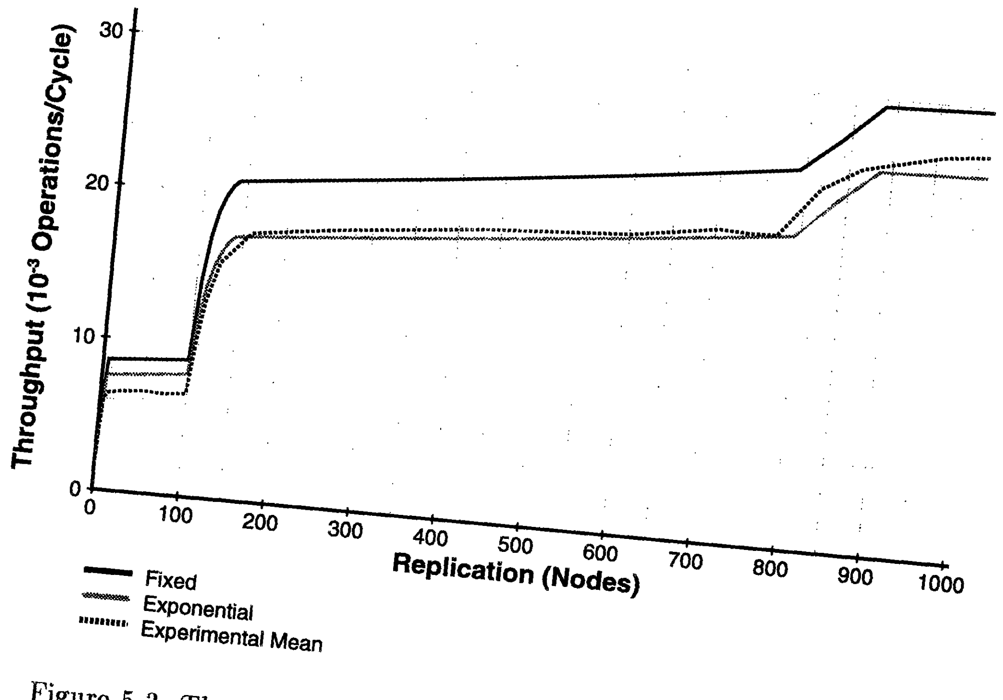


Figure 5-3: Throughput vs. Replication - Base Case, Low Replication

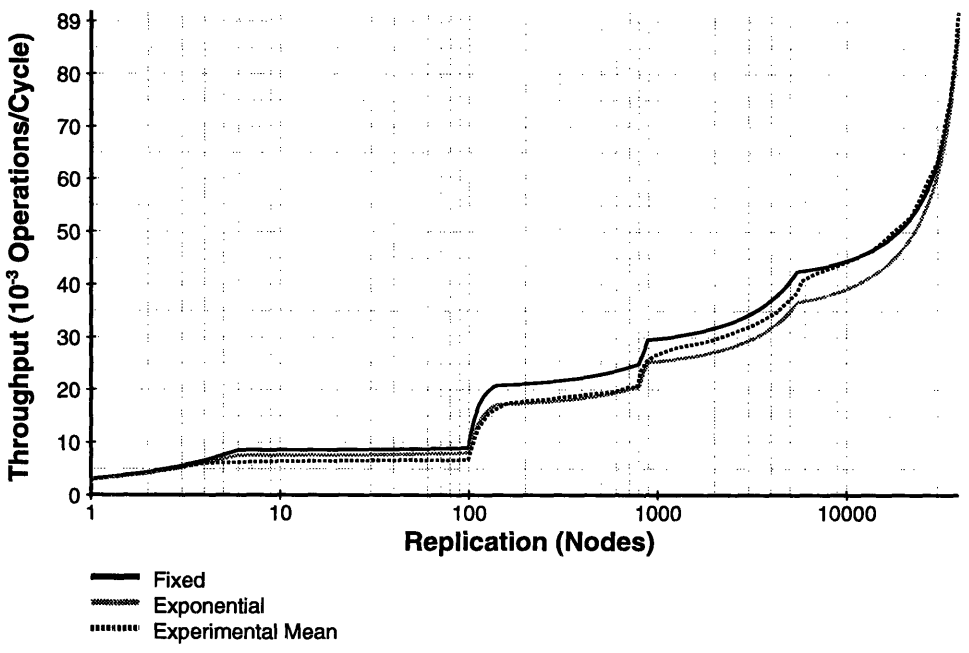


Figure 5-4: Throughput vs. Replication - Base Case, Log Scale

Figure 5-4 shows the messages of both previous graphs on one graph by using a log scale on the x-axis. The transitions between tree levels are clearly visible at around 100, 800, and 6,000 copies. We will use a log scale to present most of the results in this chapter.

Figure 5-5 shows the size of the 95% confidence interval for the mean (given by $1.96 * \frac{\sigma}{\sqrt{5}}$), as a percentage of the experimental mean. This is almost 18% when only 4 copies of the root have been made. At this low level of replication, the results are dependent on the exact layout of nodes that form the bottleneck. If two or more nodes from the level below the root happen to be on the same processor, throughput will be much lower than if they are all on different processors. As replication is increased, the confidence interval becomes much smaller as a percent of the experimental mean - around 6% during replication of the level below the root (where we are still somewhat dependent on the exact tree layout), and less than 1% for the lower two levels of the tree. This pattern is characteristic of all simulations in this chapter.

## 5.2 Changing Tree Size

We next compare simulation and queueing model results for:

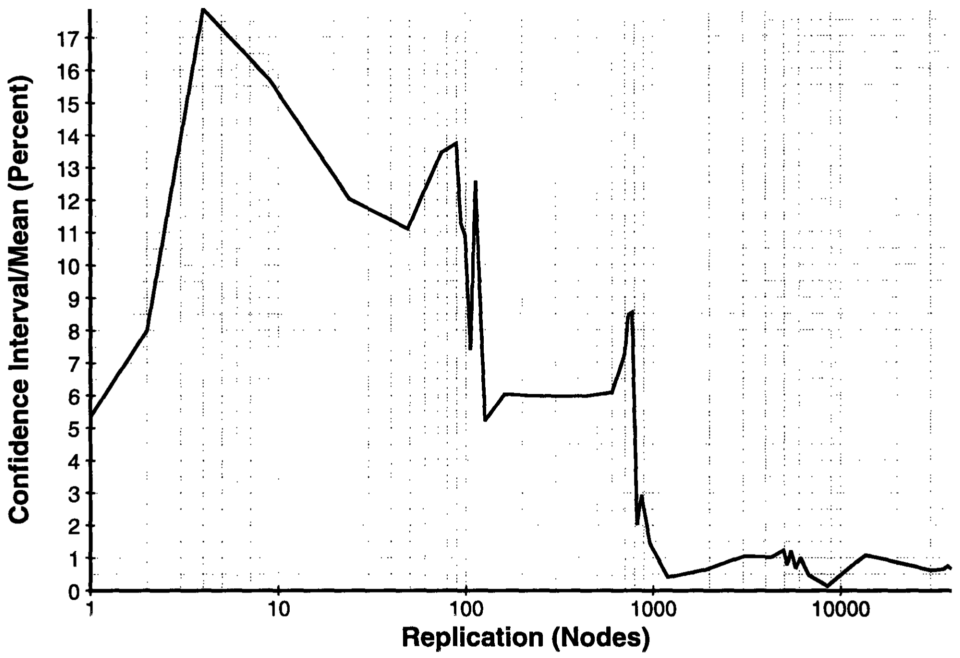


Figure 5-5: 95% Confidence Interval for Experimental Mean – Base Case

- Branch factor of 10, B-tree with 10,000 entries,
- Branch factor of 30, B-tree with 10,000 entries

Figure 5-6 shows the results for the tree with a branch factor of 10 and 10,000 entries. This configuration creates a tree of 5 levels with sizes averaging (from the root down) 1, 4, 28, 186 and 1,345 nodes. The model closely matches the results of the simulations.

Figure 5-7 shows the results for the tree with a branch factor of 30. This configuration creates a tree of 3 levels with sizes averaging (from the root down) 1, 19, and 453 nodes. The sudden increase in predicted throughput at just over 400 copies is not a transition between tree levels, but is a result of a shift from use of the bottleneck model to the high replication model.

The x-axes of figure 5-6 and figure 5-7 are not directly comparable because the B-tree nodes are of different size, but we can do a rough comparison between a given replication when the branch factor is 30 with three times that replication when the branch factor is 10. At 200 copies, BF=30 yields a throughput of around $2.5 * 10^{-2}$, while at 600 copies, BF=10 yields $2.0 * 10^{-2}$. At 1000 copies, BF=30 yields a throughput of around $3.0 * 10^{-2}$, while at 3000 copies, BF=10

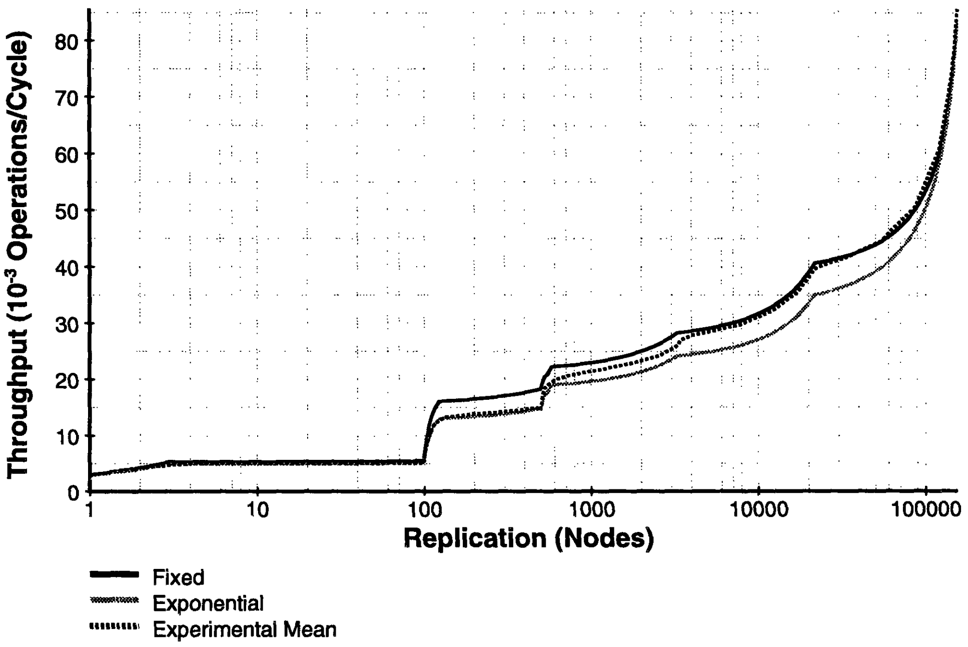


Figure 5-6: Throughput vs. Replication - Branch = 10, 10,000 Entries

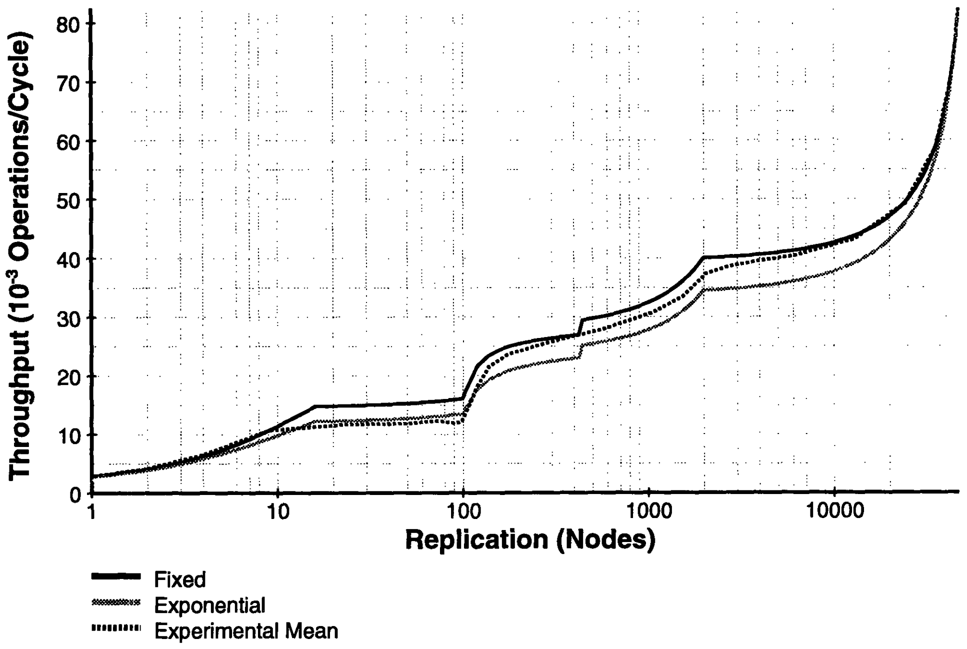


Figure 5-7: Throughput vs. Replication - Branch = 30, 10,000 Entries

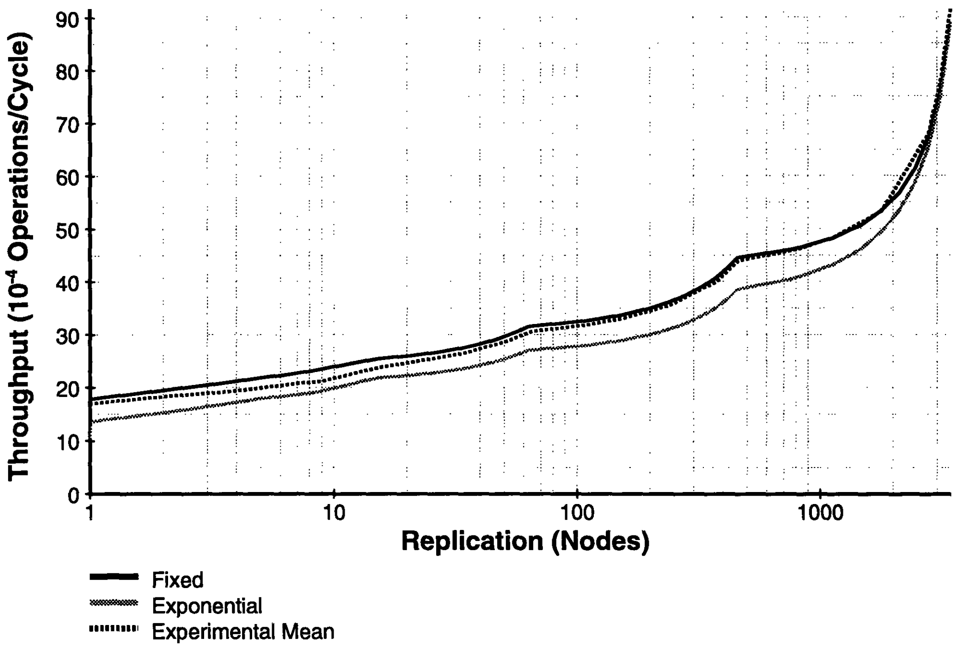


Figure 5-8: Throughput vs. Replication – 10 Processors

yields $2.5 * 10^{-2}$. The shallower broader tree provides a higher throughput because it requires fewer remote messages.

## 5.3 Changing Number of Processors

We compare simulation results and our queueing model for:

- 10 processors,
- 200 processors

With 10 processors, the total number of copies for each node is increased in the following progression: 1, 2, 3, 5, 8, 10. Figure 5-8 shows the results. Observed simulation throughput is very closely matched by the prediction of the model using fixed service times. With 10 processors there is no obvious flattening of the throughput curve. The "bottleneck" presented by the 7 tree nodes below the root does not significantly restrict the growth in throughput as the root is fully replicated.

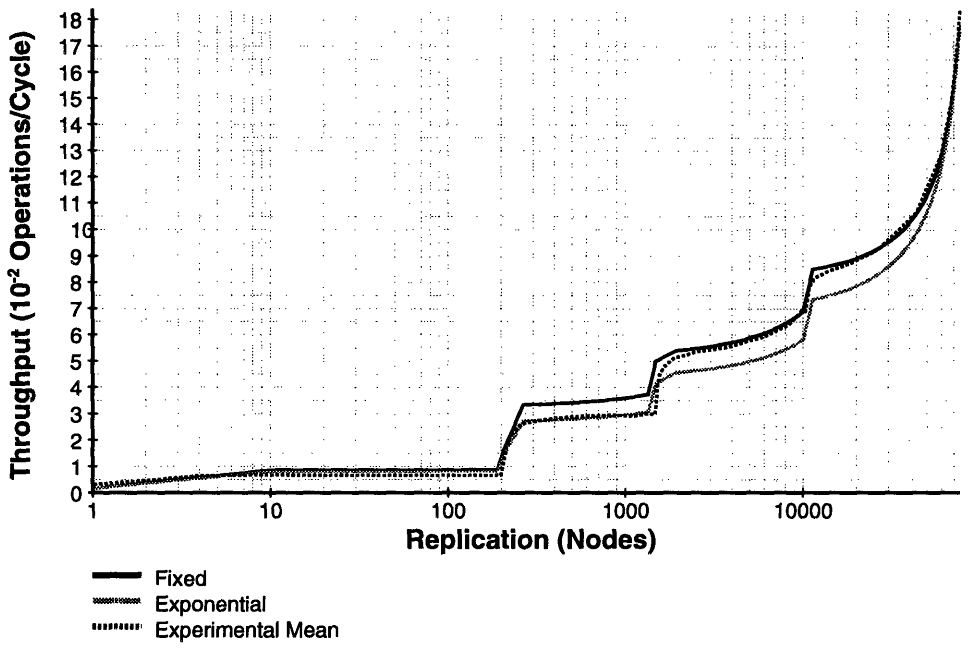


Figure 5-9: Throughput vs. Replication – 200 Processors

In comparison with the base case, the system throughput with 10 processors is significantly lower because there are fewer operations active in the system at a time.

Figure 5-9 shows the results for a network of 200 processors. The increase in performance that occurs when the leaves are first replicated (around 10,000 copies) indicates that the unreplicated leaf level is a bottleneck. In chapter 2 we introduced the formula, *place*$(m, P)$, indicating the expected number of processors that $m$ objects will cover when randomly placed across $P$ total processors. Applying this formula to this situation, *place*$(343, 200) = 164.16$ processors, which verifies that we do expect the unreplicated leaf level to be a bottleneck.

## 5.4 Changing Message Overhead

The base case value for message overhead, approximately 508, was drawn from the actual observed cost of our implementation. To vary the cost of this element we use Proteus' ability to directly manipulate cycle counts.

We compare simulation results and our queueing model for:

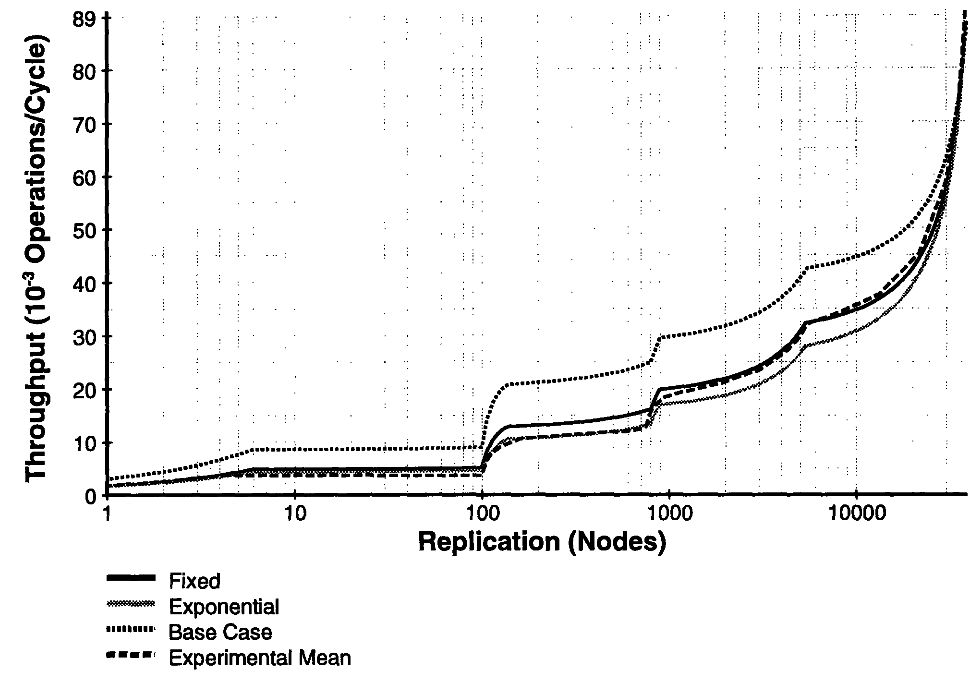


Figure 5-10: Throughput vs. Replication – Extra 500 Overhead

- Base *mesg_ovhd* + 500
- Base *mesg_ovhd*+ 2000

Figure 5-10 shows the throughput from simulations and the queueing model for the case where message overhead is the base + 500. Again, the model closely matches the results of simulation. In comparison to the base case, throughput is substantially lower for low replication, but becomes equal when the tree is fully replicated. This is expected, for at full replication no remote messages need be sent.

Figure 5-11 shows the throughput from simulations and the queueing model for the case where message overhead is the base + 2000. Note that as message overhead cost increases, the throughput for full replication remains roughly the same, but that the throughput for less than full replication is significantly lower than the base case, as expected.

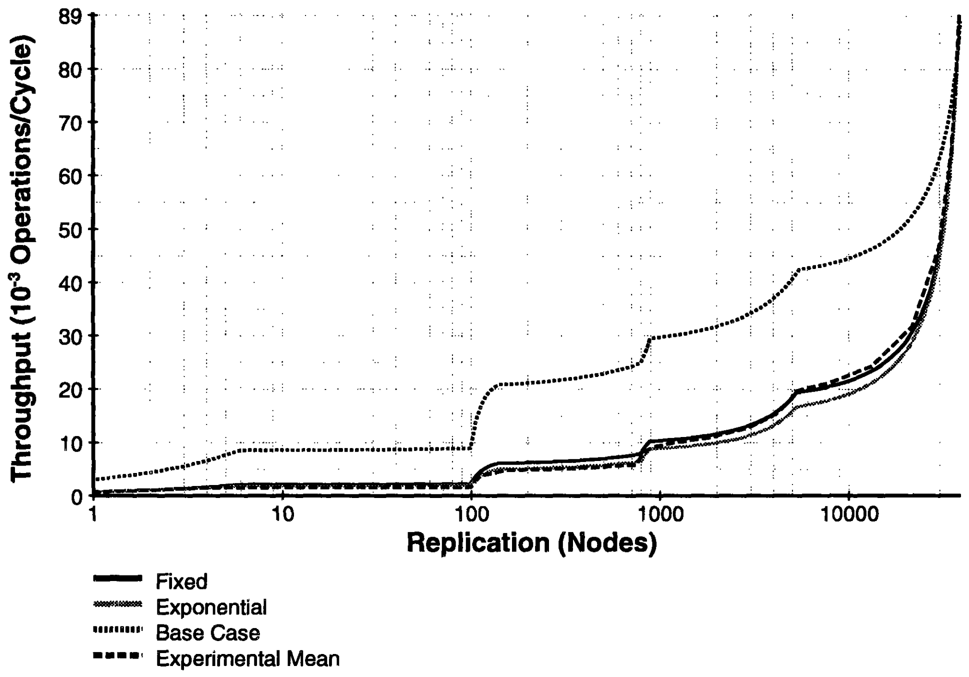


Figure 5-11: Throughput vs. Replication – Extra 2000 Overhead

## 5.5 Changing Network Delay

Our base value for network delay came from Proteus' model of the J-Machine network. This model assumes a wire delay of 1 unit per hop and a switch delay of 1 unit. We alter that model and compare simulation and our queueing model for wire delay of 15 and switch delay = 15. The results are shown in figure 5-12. The overall costs of network transmission rises from 17 in the base case to 216, but remains relatively small in comparison to other costs. The implication for multi-processor design is that the ability to send and receive messages with low processor overhead is relatively more important than low network transmission delay. Network delays and message send/receive overhead both contribute to longer latency, but message overhead also uses a significant amount of a resource in high demand – processing time.

## 5.6 Changing Start/End Costs

In our simulations a small amount of time is spent between the completion of one operation and the initiation of a new one to update and record performance statistics. This period of time

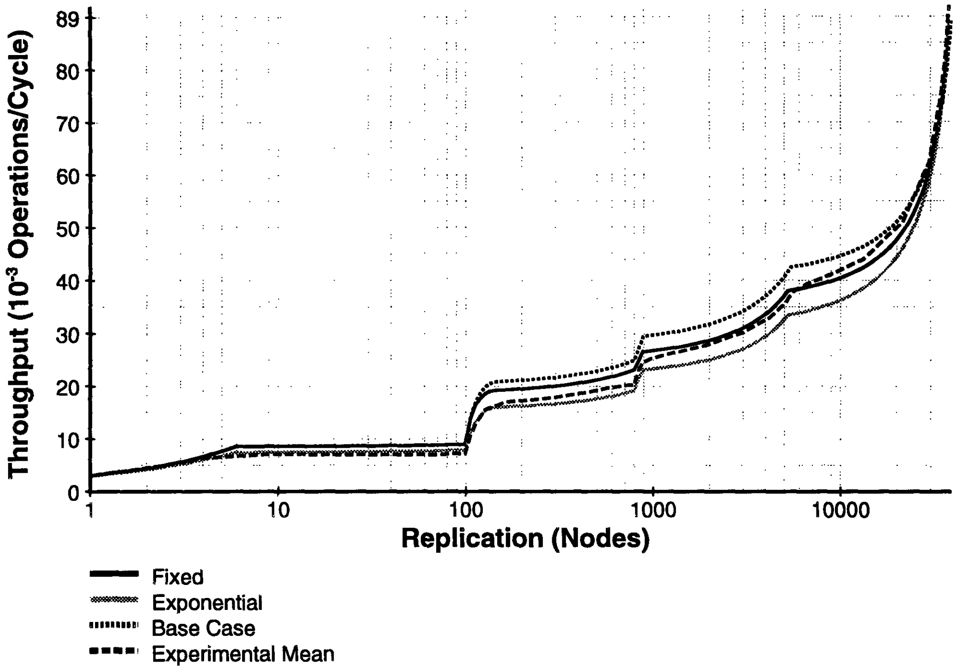


Figure 5-12: Throughput vs. Replication – Wire delay = 15, Switch delay = 15

represents the "think" time of an application program between B-tree operations. We again use Proteus' ability to manipulate cycle counts to increase this "think" time. We compare simulation and queueing model for:

- Base *start_cost* + 500
- Base *start_cost* + 2000

Figures 5-13 and 5-14 show that the results of the queueing model closely match simulated results in both cases. Throughput stays lower than the base case even with full replication because we have increased the minimum amount of work required for every operation.

## 5.7 Changing Operations per Processor

All prior examples allowed each processor to have one active operation in the system at a time. By increasing the number of active operations per processor (in queueing network theory terminology: increasing the number of tasks per class) we would expect system throughput to increase at points where it is not restricted by a fully utilized bottleneck.

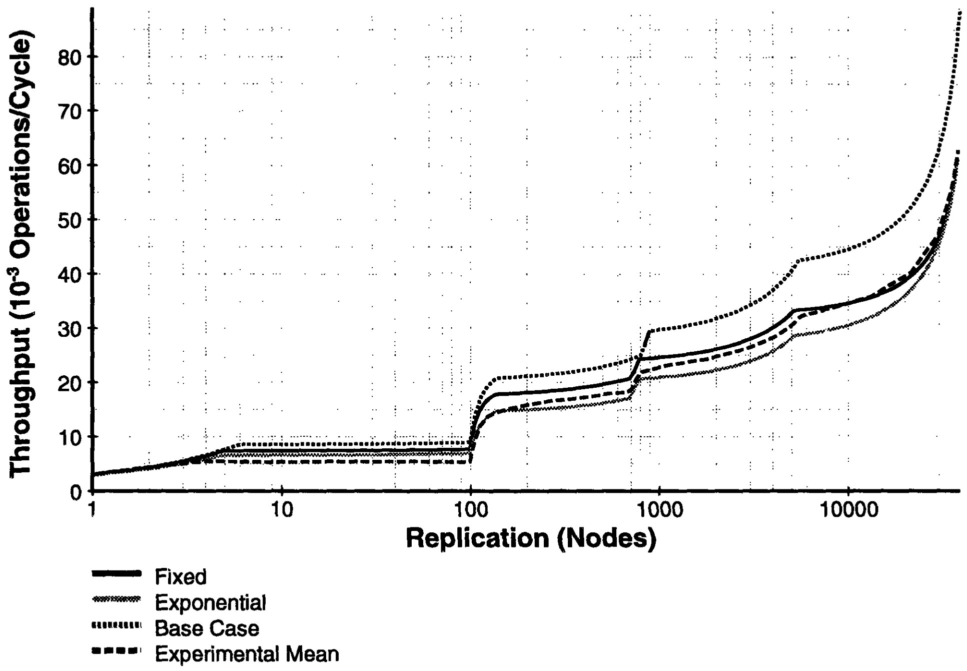


Figure 5-13: Throughput vs. Replication – Extra 500 Start

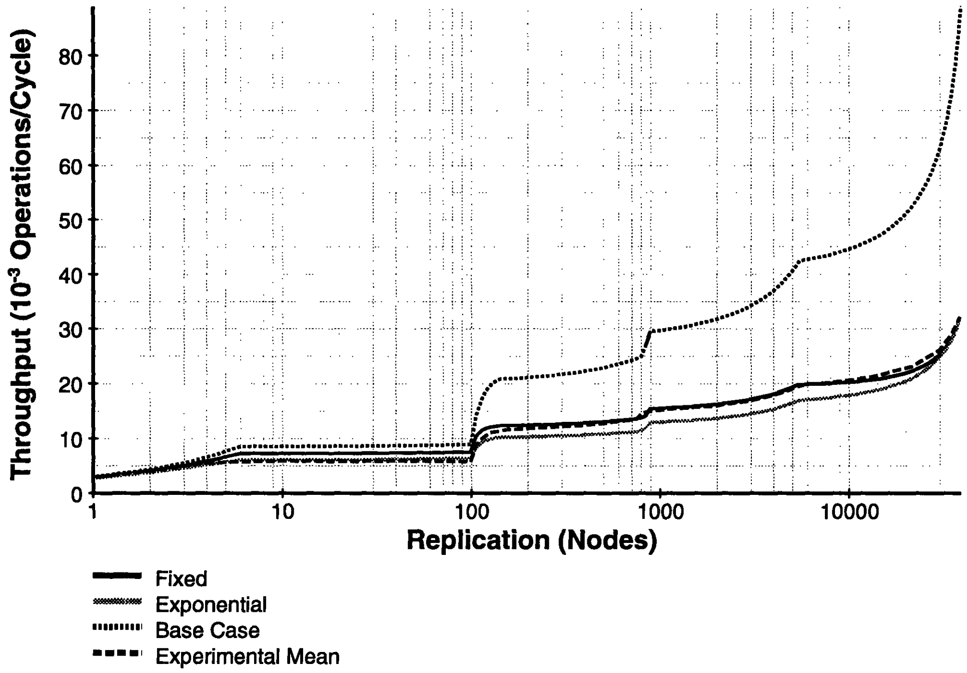


Figure 5-14: Throughput vs. Replication – Extra 2000 Start

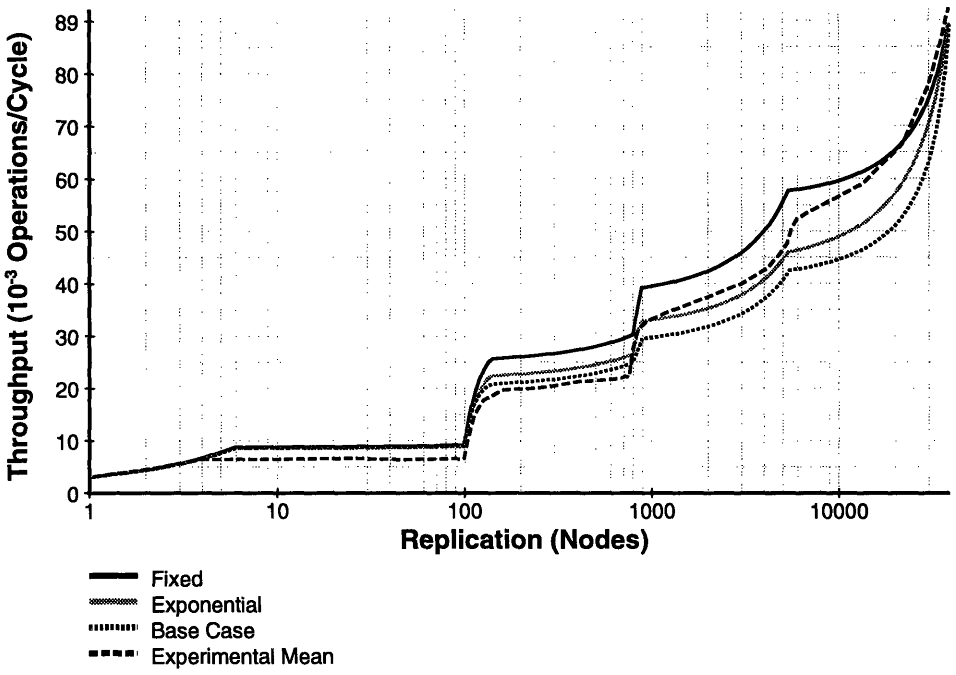


Figure 5-15: Throughput vs. Replication – 2 Operations Per Processor

We compare simulation and queueing model for:

- 2 operations per processor
- 4 operations per processor

Figures 5-15 and 5-16 show the results for 2 and 4 operations, respectively. Note that at very low replication, the system throughput is very similar to the base case – the capacity of the bottleneck processors places a hard upper bound on throughput. As the bottleneck is distributed across more processors, the addition of extra tasks to the system does increase system throughput. When the tree is fully replicated there is no net throughput gain – every processor will be fully utilized, forming a "bottleneck" of sorts.

Although throughput can be increased by adding operations to the system, in these cases it comes at the cost of latency. Figure 5-17 compares latency for the system with 1 task per processor and with 4 tasks per processor, showing that latency increases, roughly 3 to 4 times when 4 operations per processor are allowed. Figure 5-18 shows that as replication increases, the ratio approaches 4, as one would expect.

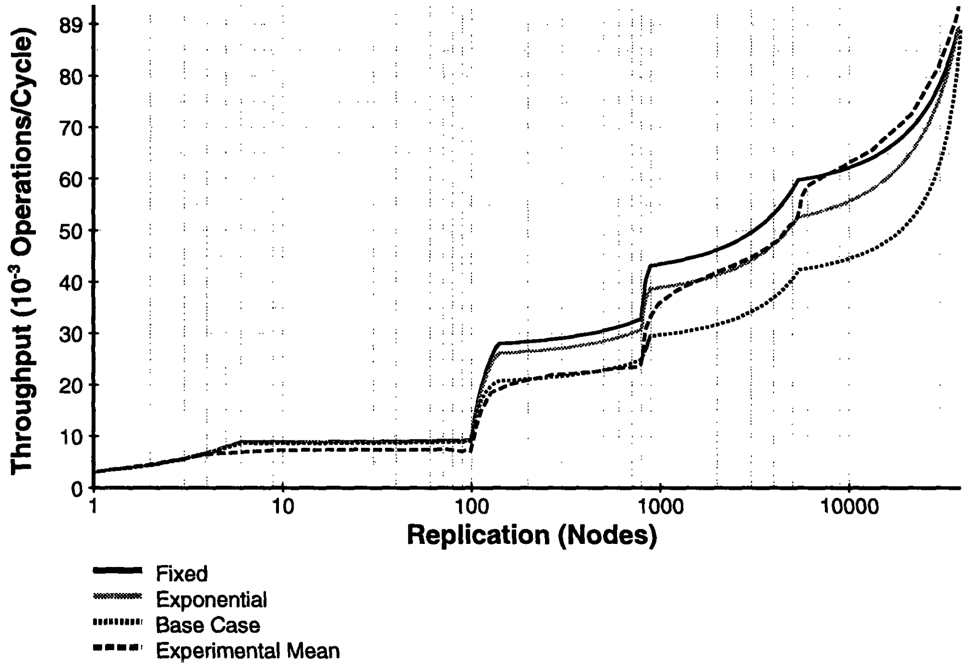


Figure 5-16: Throughput vs. Replication – 4 Operations Per Processor

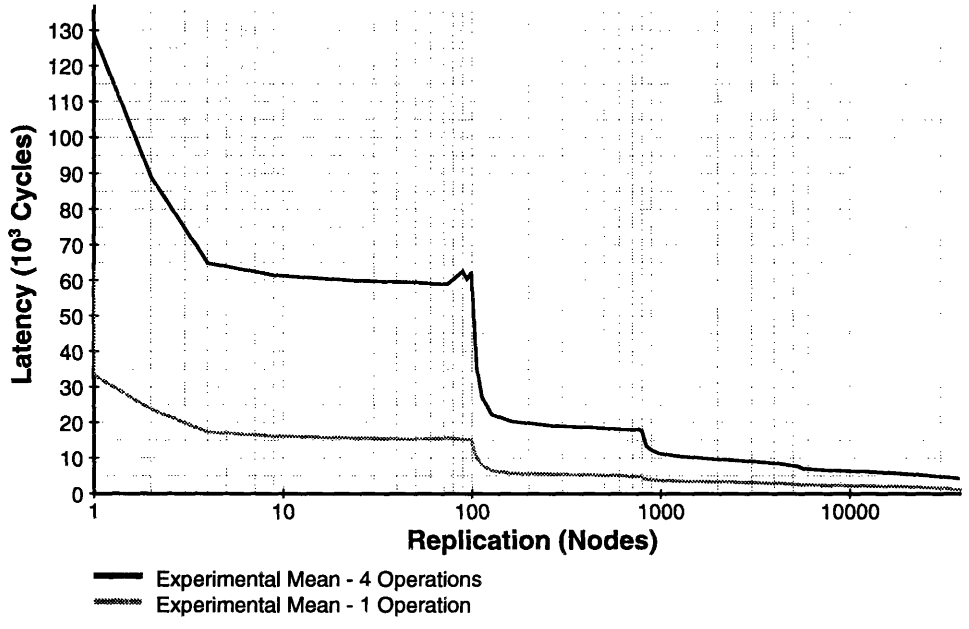


Figure 5-17: Latency vs. Replication – 1 and 4 Operations Per Processor

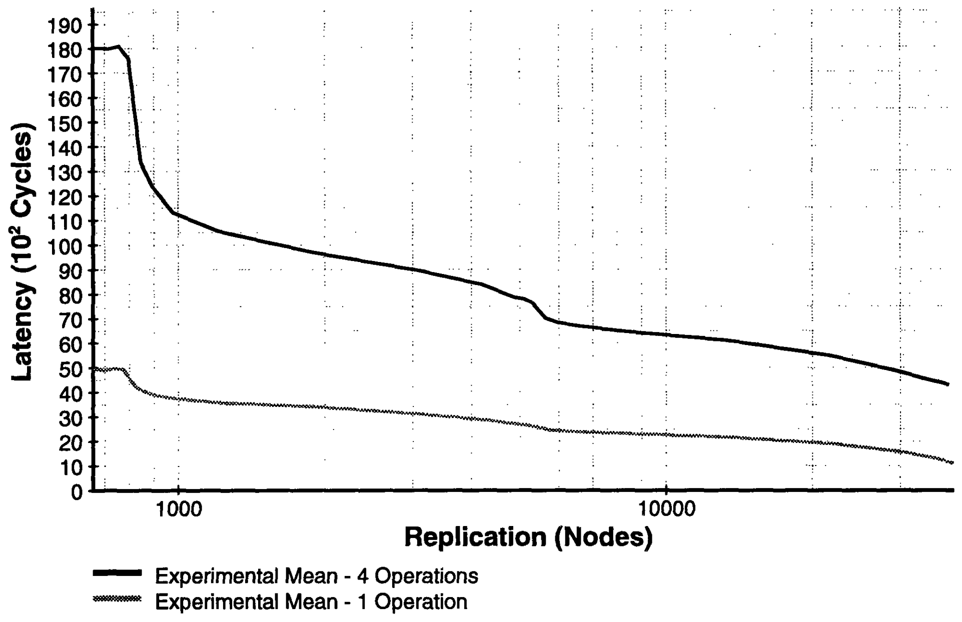


Figure 5-18: Latency vs. Replication – 1 and 4 Operations Per Processor, High Replication

## 5.8 Changing Operation Mix

When the mix of B-tree operations is changed from exclusively lookups to include insertions and deletions, we expect to see system throughput decrease. Modifications to the tree will incur costs to change the contents of tree nodes, create new tree nodes, and update copies of changed nodes.

We compare simulation and queueing model for three scenarios:

- 95% lookups, 4% inserts, 1% deletes – (95/4/1)
- 70% lookups, 20% inserts, 10% deletes – (70/20/10)
- 50% lookups, 25% inserts, 25% deletes – (50/25/25)

We use the format x/y/z to refer to an operation mix with x% lookups, y% insertions, and z% deletions.

When the operation mix is changed to allow insertions and deletions, the size of the B-tree can change during the simulation. While our queueing network model accounts for the costs of

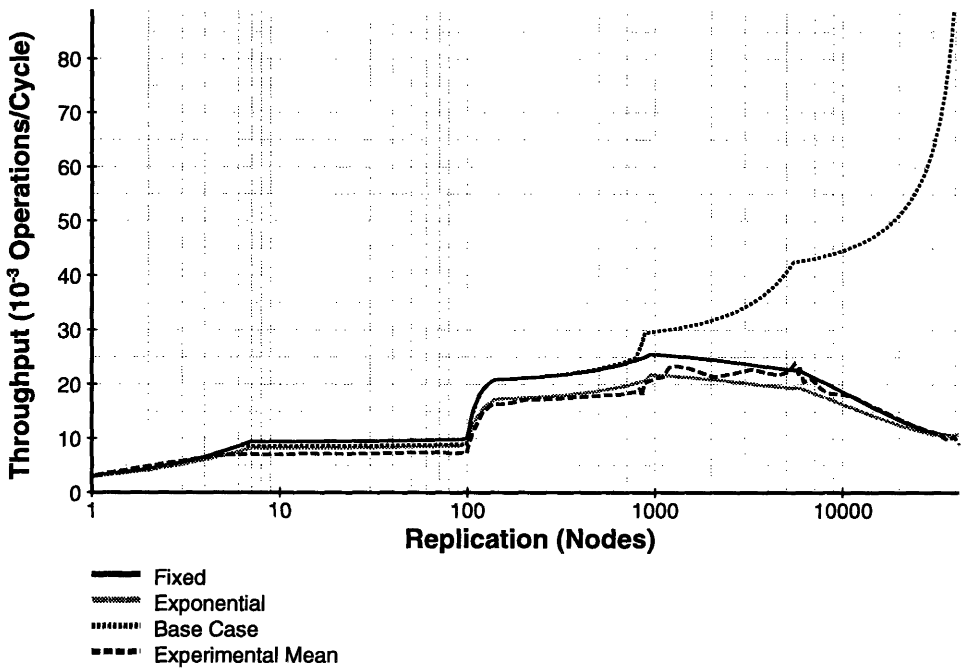


Figure 5-19: Throughput vs. Replication – Operation Mix 95/4/1

B-tree node splits and updates, it does not change the tree size. To compare the queueing model with the simulation, we use the size of the tree after the simulation has completed. Figure 5-19 shows the results for the 95/4/1 case. Throughput is very close to that of the base case while the top two levels of the tree are replicated. It is clear that throughput starts to decrease below that of the base case when replication starts at the third level.

Figure 5-20 shows the results for the 70/20/10 case. In this case performance is lower than that of the base case as the second level of the tree is replicated. It is very noticeable that, even though throughput is generally dropping after replication starts at the second level, the removal of the bottleneck as replication starts at the third level produces an increase in throughput.

For the 50/25/25 case, we first note that Johnson and Shasha [JS89] showed that the expected leaf utilization will drop to approximately 40%, not the 70% that generally holds when inserts are more prevalent than deletes. Even if we are making no net additions to the tree, a significant number of nodes will be added as the tree adjusts from 70% leaf utilization from tree construction to the new steady state 40% leaf utilization. We are interested in the steady state behavior, not the transition, so to reshape the tree appropriately we have run

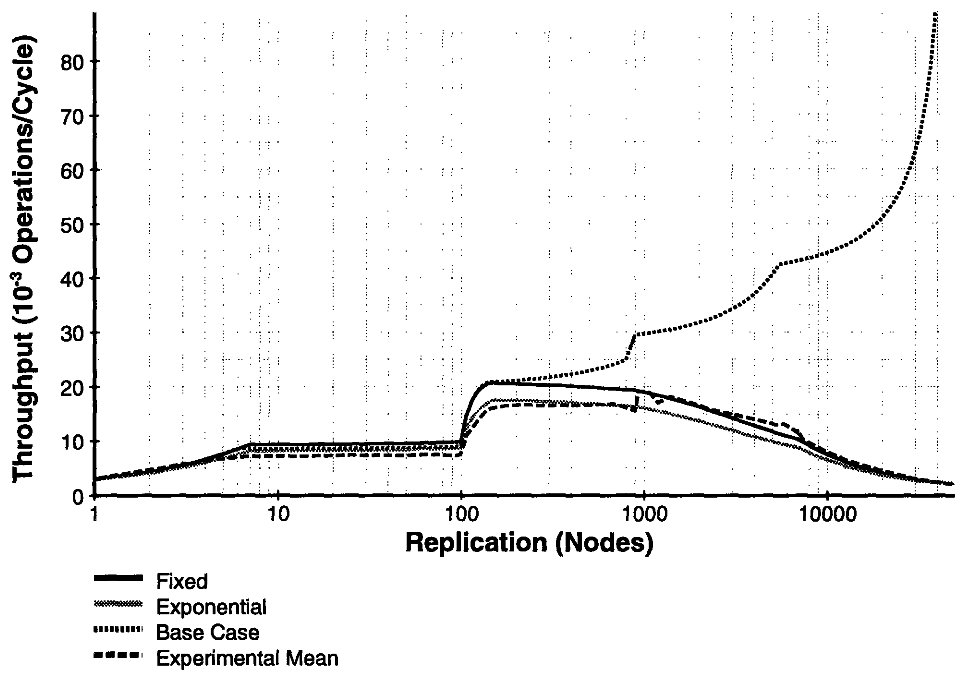


Figure 5-20: Throughput vs. Replication – Operation Mix 70/20/10

40,000 modifications at the equal mix before starting the measured simulation. Figure 5-21 shows a noticeable discrepancy between the results of simulation and the model in the range between 2,000 and 7,000 copies. This discrepancy is caused by a slight mismatch between the model and our simulation. If the balance of inserts and deletes is truly 50/50, the model says there should be no net growth in the leaves and therefore no insertions above the leaf level. This was not the the case in the simulation, there continued to be some insertions in the level above the leaves. In figure 5-22 we use the model results when the insert/delete balance is changed from 50/50 to 51/49. This models a small amount of change in the level above the leaves and produces a close match with the observed results of simulation.

## 5.9 Summary

In this chapter we have used a variety of simulations on relatively small trees (100 processors, $<$ 10,000 entries) to validate our queueing network model. This model has been shown to accurately model the behavior of distributed B-trees built with random placement of tree nodes and copies, when the B-tree operations use a uniform distribution of search keys.

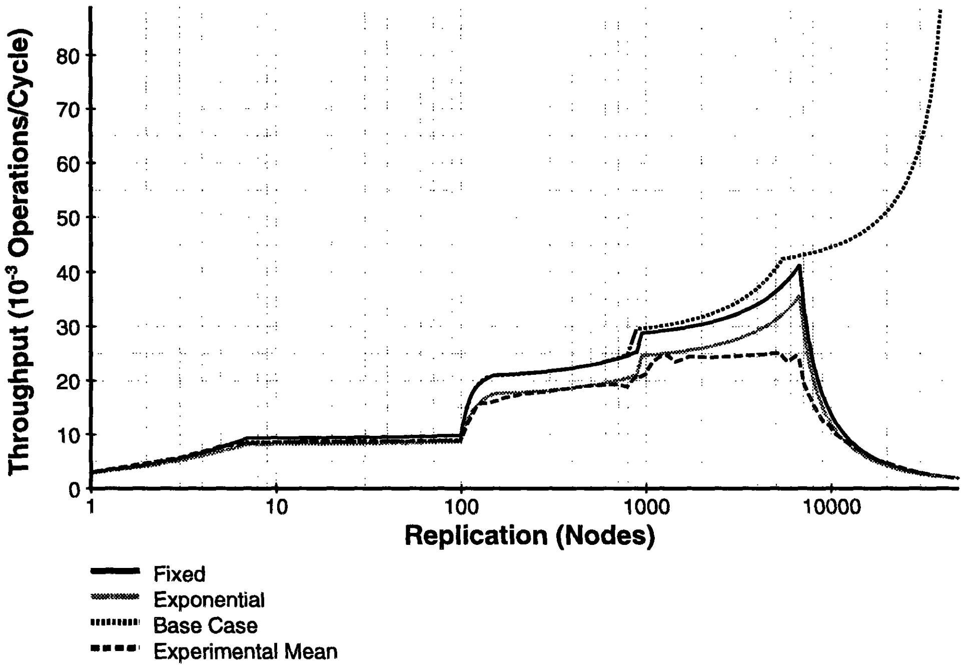


Figure 5-21: Throughput vs. Replication – Operation Mix 50/25/25

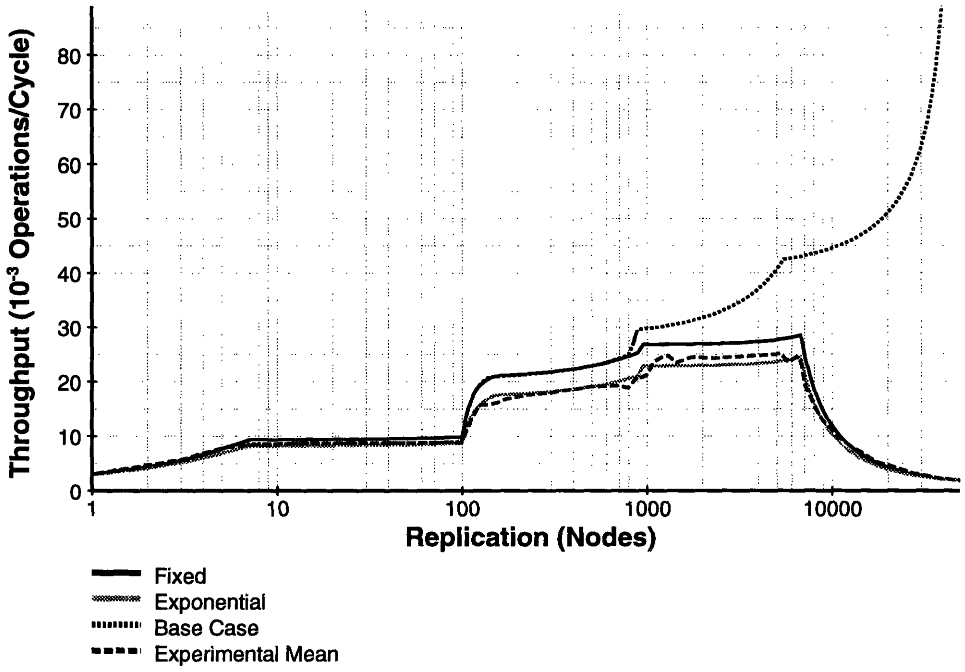


Figure 5-22: Throughput vs. Replication – Operation Mix 50/25/25 (Modified)

The use of the queueing network model and simulations has also clearly indicated three key results from our study of replication:

- Replication of B-tree nodes can increase throughput and there is a tradeoff between space used for replication and throughput.
- Replicating as we have done so far (from the top down) creates bottlenecks to throughput.
- The inclusion of tree modifying operations can reduce throughput.

We explore these results in more detail in the next chapter.

# Chapter 6

# Static Replication

The results presented in the previous chapter to validate our queueing network model demonstrate that replication of B-tree nodes can improve system throughput. They also clearly indicate there is a tradeoff between the use of space for replication and throughput. Thus, it is not adequate to suggest a single pattern for replication based only on the tree size and the number of processors. Instead we must be able to suggest how best to replicate tree nodes given an amount of space available to use and be able to describe the potential value or cost of using more or less space for replication.

This problem is challenging because our results also indicate that the value of an additional copy of a B-tree node is not always the same for all nodes. In particular, three characteristics of replication and throughput are observable:

1. A bottleneck created by one level of the tree can severely restrict throughput, limiting the value of additional replication of other levels,

2. The marginal value of additional replication tends to decrease as nodes lower in the tree are replicated (although within a level the marginal value can increase as more copies are made),

3. When inserts and deletes are included in the operation mix, replication of lower levels of the tree can actually reduce throughput.

In this chapter we explore these characteristics to develop rules for the use of replication to maximize throughput. We first describe the rules for replication assuming lookup operations only. We next augment these rules to include insert and delete operations. We then remove the assumption that the distribution of search keys is uniform across the key space and demonstrate the potential need for dynamic control of replication. We conclude the chapter by comparing the performance of distributed B-trees using these rules for replication with an alternate placement method, Johnson and Colbrook's path-to-root.

## 6.1 Rules for Replication – Lookup Only

Every B-tree lookup requires (on average) the same amount of time for what we earlier termed "productive work" – each operation must visit one node at each level of the tree. If the latencies of two lookups are different, there are two possible sources for the difference: remote message costs and queueing delays. Remote message costs increase latency as overhead service time is incurred for sending and receiving messages and for the actual transmission of the messages over the communication network. Queueing delays increase latency as an operation waits its turn for service at a processor. Replication of B-tree nodes serves to reduce latency from both sources. However, minimizing queueing delays does not minimize the number of remote messages and vice versa. In this section we develop the rules for the use of replication; first we propose and describe two basic rules for replicating B-tree nodes, aimed at reducing remote messages and queueing delays, respectively:

- Fully replicate the most frequently used node before making additional copies of any other node.
- Add copies to balance the capacity of each tree level and eliminate bottlenecks,

We develop each of these rules in turn and then propose a hybrid of the two rules and show that it produces results that closely match the best results of of these approaches.

### 6.1.1 Rule One – Replicate Most Frequently Accessed Nodes

The first rule is the rule applied in all simulations shown so far. This rule seeks to reduce latency (and increase throughput) by reducing the number of remote messages required to complete a B-tree operation. The number of remote messages will be reduced most by making an additional copy of the most frequently used tree node that is not fully replicated. We first explain why that is true and then introduce a graph of suggested replication versus relative frequency of access that characterizes the rule.

Given a uniform distribution of keys for lookup operations, the relative frequency of access to a B-tree node over a large number of operations is the same for all nodes at the same level. If we define the root node to have relative frequency of access 1 (it is accessed for every operation), the relative frequency of access for nodes at other levels is $\frac{1}{branch\_factor^{depth}}$, where *depth* is the distance from the root node. As we expect, a node higher in the tree is used more often than a node lower in the tree.

For a B-tree node with $r$ total copies placed randomly on a system with $C$ processors, the probability of having to forward a message remotely to reach a copy is $1 - \frac{r}{C}$. The relative rate of remote access is then the product, $\frac{1}{branch\_factor^{depth}} * (1 - \frac{r}{C})$. The addition of another copy of a B-tree node reduces the total rate of remote access by an amount proportional to $\frac{1}{branch\_factor^{depth}} * \frac{1}{C}$. Thus, when the access pattern is uniform, the number of messages is always reduced more by adding a copy higher in the tree than it is by adding a copy lower in the tree.

Figure 6-1 shows how the number of remote messages (and thus their frequency) decreases with increasing replication for our base case simulation. It graphically demonstrates that replication from the top down is the most efficient way to reduce remote messages. With no replication, the total number of remote messages required for the 5,000 lookups was nearly 25,000 (visible in figure 6-2 using a log scale for the x-axis). With full replication of the root, the number drops to about 20,000; replication of the level below the root further reduces the number to about 15,000. For each of the top three levels of this four level tree, full replication removes approximately 20% of the remote messages (60% total for the three levels), with replication of the leaves removing the remaining 40%. When all higher levels are fully replicated, replication of the leaves eliminates twice as many messages as replication of each of the other levels does because it eliminates both a forwarding message and the return to the home processor. The slope of the curve in figure 6-1 represents the reduction of remote message frequency per additional copy. As predicted, this is constant within each level and decreases as the level being replicated is farther below the root.

This rule for replication yields a graph of suggested replication versus relative frequency of

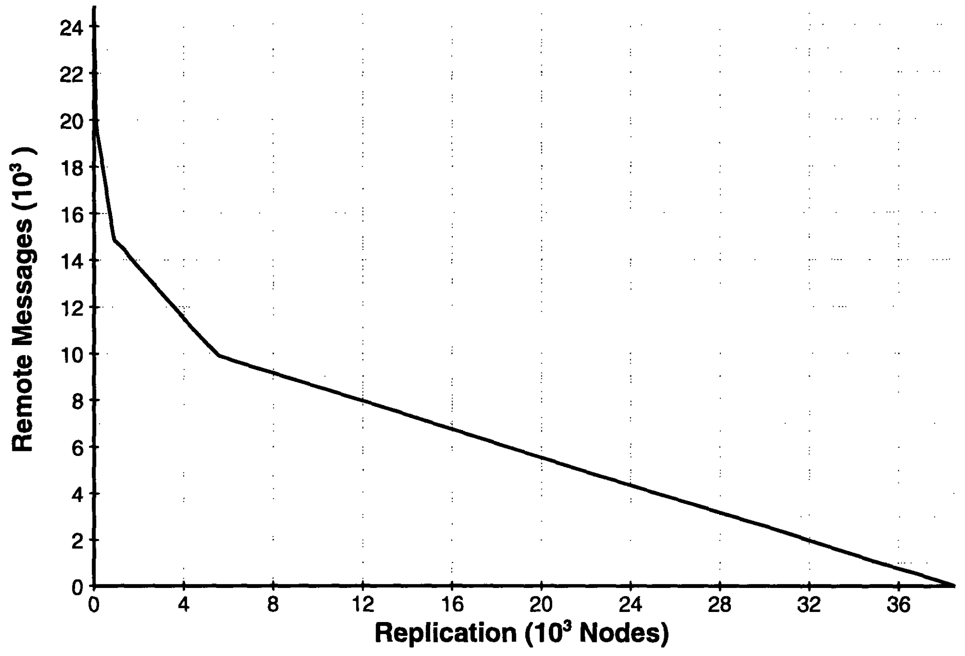


Figure 6-1: Number of Remote Accesses vs. Replication – Base Case

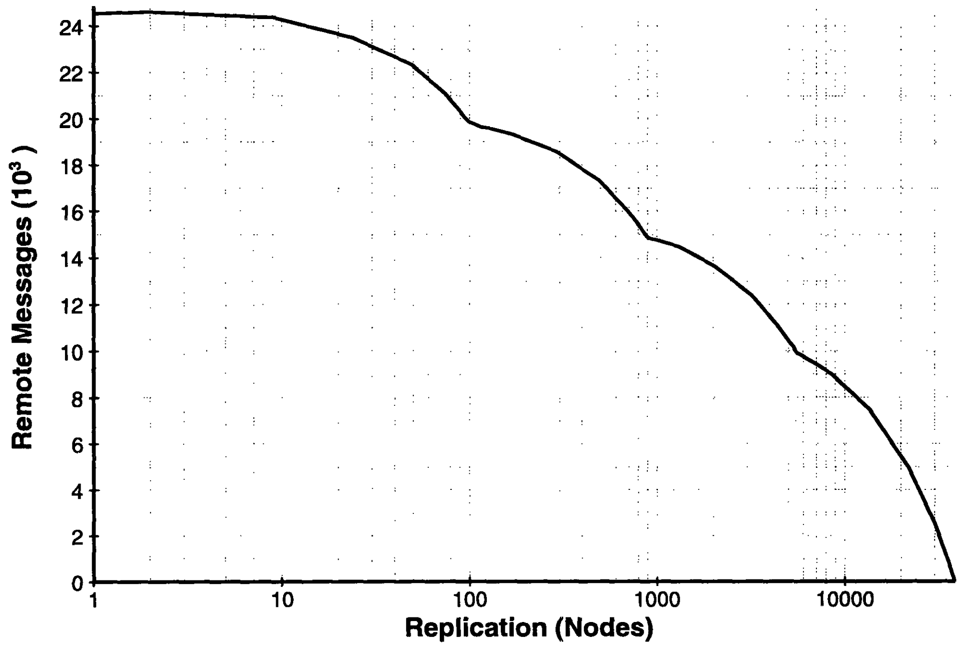


Figure 6-2: Number of Remote Accesses vs. Replication – Base Case, Log Scale

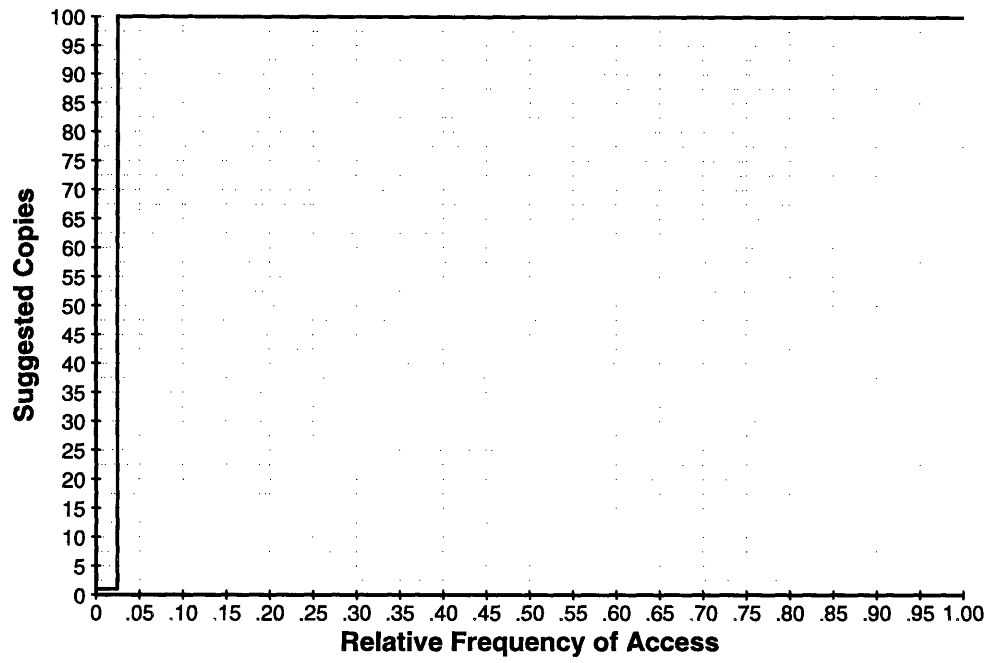


Figure 6-3: Suggested Copies versus Relative Access Frequency – Rule One

access of the form shown in figure 6-3. The graph shows the number of copies (including the original) that the rule suggests be made for a specified relative frequency of access, where the root node has a relative frequency of access of 1.0. All nodes with relative access frequency above a threshold are fully replicated. Nodes with relative access frequency below the threshold are not replicated at all. Nodes with relative frequency of access equal to the threshold may be partially replicated. Changing the threshold frequency will change the amount of space used – a lower threshold uses more space, while a higher threshold will use less space.

### 6.1.2 Rule Two – Balance Capacity of B-Tree Levels

The existence of bottleneck levels in our simulations indicates that our first rule is insufficient. The second rule seeks to reduce latency by eliminating severe queueing delays that result from capacity bottlenecks in the tree. This rule makes intuitive sense if the B-tree is thought of as a pipeline with each tree level a stage of the pipe. Every tree operation must pass through each level of the tree, so processing capacity would intuitively be best allocated if given equally to all tree levels. Any level with less processing capacity than the others will be a bottleneck and limit throughput.

The processing capacity of a tree level can be roughly thought of as the number of processors that hold an original or copy of a node belonging to the level. In an unreplicated tree, for example, the leaves will likely be spread across all processors while the root is on only one processor – the leaf level has more processing capacity than the root. As we replicate the root, we eliminate the root bottleneck by increasing the processing capacity for the root, and thereby increase throughput.

The replication rule we have used so far explicitly ignores this capacity balance rule. In our queueing network model and simulations presented so far we have been adding copies from the top down; we make no copies at a tree level until all higher levels are fully replicated. The results show graphic evidence of bottlenecks. Graphs of throughput versus replication frequently have flat spots where the throughput does not increase (or the rate of increase drops significantly) as replication increases. As replication continues to increase, throughput suddenly jumps dramatically and then continues with a gradual increase until, perhaps, it hits another flat spot. Figure 6-4 shows again the results for our base case simulation, expanding on the low end of replication up to initial replication of the third level of the tree. Between the point of 7 or 8 copies of the root and the first additional copies of the nodes below the root (at approximately 100 copies), throughput increases very little. In that range system throughput is limited by the processing capacity of the second level in the tree. The phenomenon occurs again, to a less dramatic degree, between roughly 150 copies and 800 copies. During that range the third level of the tree is forming a bottleneck.

By balancing capacity, these flat spots can be avoided. Figure 6-5 shows the results of allowing replication of the second level before fully replicating the root. In these experiments, we partially replicated the root, then repeatedly added an additional copy of each node below the root and ran a simulation, stopping at 10 copies of each node. The four new curves show the results for 10, 30, 50 and 70 total copies of the root. They illustrate the need to add copies to balance capacity across levels. First, they demonstrate clearly that the level below the root was limiting throughput, for adding a copy of every node at that level yields a large increase in throughput. Second, they show that as the level below the root is replicated, the capacity of the root level can once again become a bottleneck. When there are only 10 copies of the root,

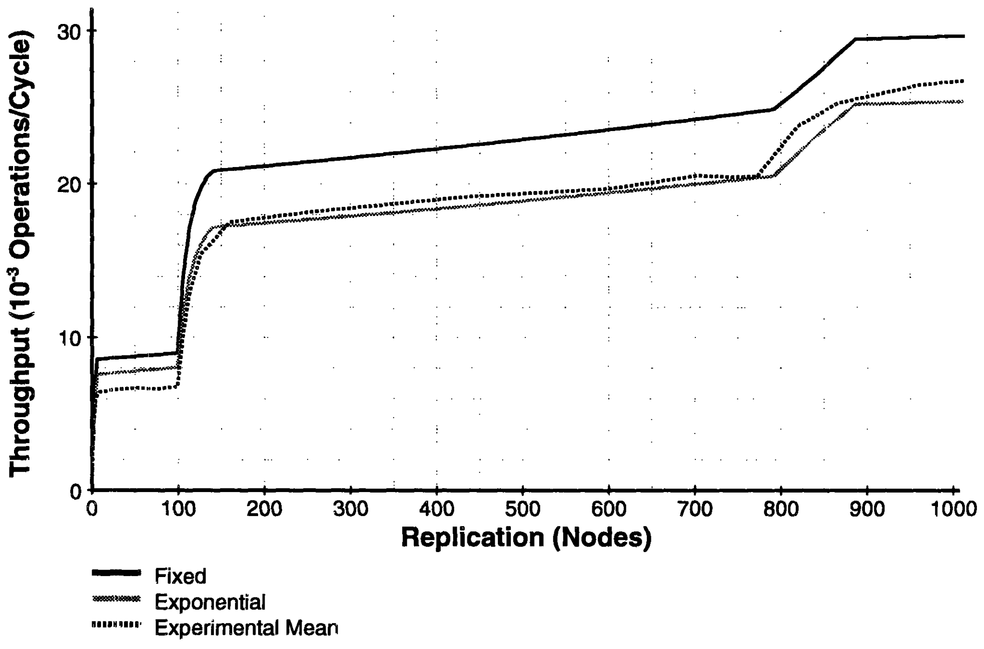


Figure 6-4: Throughput vs. Replication – Base Case

for example, throughput increases as the first copy is made of each node directly below the root, but then stays roughly flat as additional copies are made. The 10 processors holding the root have once again become the bottleneck. When more copies of the root are made, throughput once again increases. Thus, replication should be used to balance the capacity of the tree levels for as long as the potential for bottlenecks exists.

We have said that processing capacity can be roughly thought of as the number of processors that hold an original or copy of a B-tree node. If $r$ copies of each of $n$ nodes are made they will not, however, cover $r * n$ processors. Since placement for each node is independent of the other nodes, actual processor coverage is significantly less. When $r$ total copies of a node are distributed across $C$ processors, each processor has probability $\frac{r}{C}$ of holding a copy and probability $1 - \frac{r}{C}$ of not holding a copy. If we make $r$ total copies of each of $n$ B-tree nodes, each processor has probability $(1 - \frac{r}{C})^n$ of not holding a copy of any of the nodes. Thus, when $r$ total copies are made of each of $n$ nodes, the expected number of processors holding one or more copies, the processing capacity for the level, is:

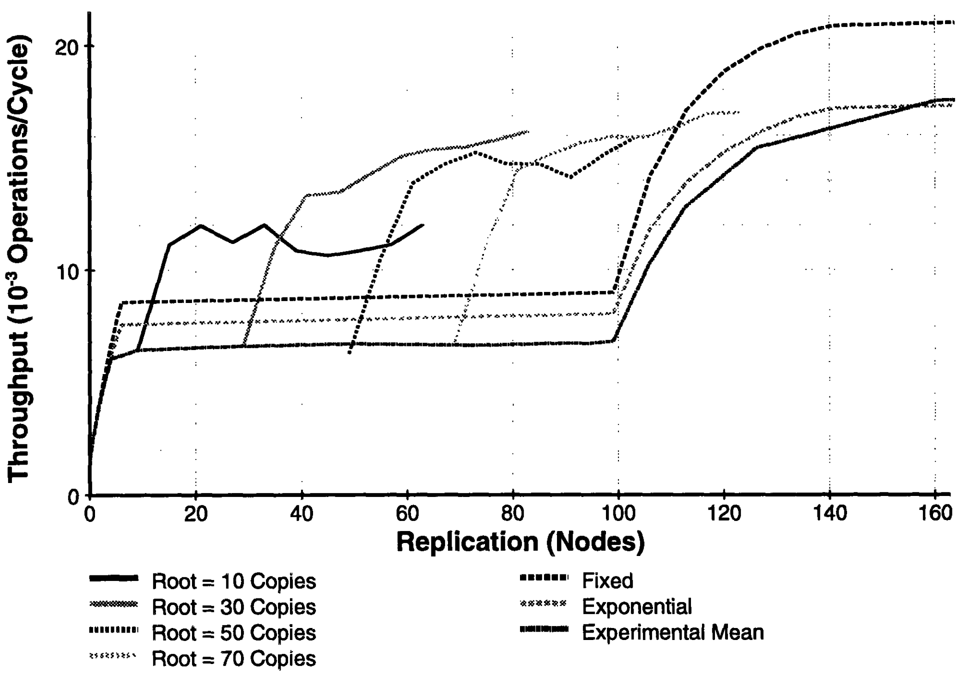


Figure 6-5: Throughput vs. Replication - Balancing Level Capacity

$$Capacity = C * (1 - (1 - \frac{r}{C})^n)$$

The graph of capacity versus copies per node for $n = 7$ and $C = 100$ is shown in figure 6-6. Note that when 15 copies have been made of each node, 105 total, they will cover, on average, only 66 processors. When 30 copies have been made of each node, making 210 total, 88 processors will hold one or more nodes. More than 60 copies of each node are required to result in one or more copies on 99 processors.

If we call $\gamma$ the fraction of the processors $C$ that we wish to hold one or more copies, then:

$$\gamma = 1 - (1 - \frac{r}{C})^n$$

and, solving for $r$, the number of copies needed to cover $\gamma * C$ processors when there are $n$ objects:

$$r = C * (1 - (1 - \gamma)^{\frac{1}{n}})$$

The second rule is characterized by this equation for $r$ and the resulting graphs, shown in figure 6-7, of the suggested replication, $r$, versus relative frequency of access for several values

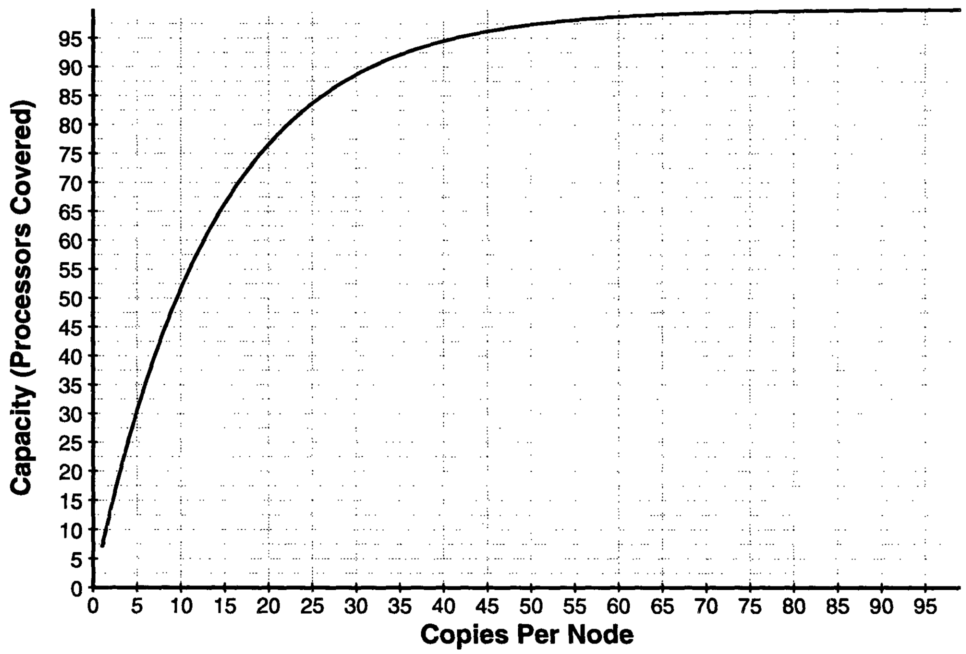


Figure 6-6: Capacity versus Copies per Node – $n = 7$, $C = 100$

of $\gamma$. Relative frequency of access is equivalent to $1/n$ for every node on a level with $n$ nodes. To use this "capacity balancing" rule, we would pick a value for $\gamma$ that would use the amount of space available for replication, then use the equation for $r$ to determine how many total copies should be made of each node. Increasing $\gamma$ will make more copies and use more space, reducing $\gamma$ will use less space. Of course, each node has a minimum of 1 copy, the original.

This rule, add copies to balance the capacity of B-tree levels, is also found in other approaches to distributed B-trees. The pipelined approach to B-trees presented by Colbrook et al. in [CBDW91] assures equal processing capacity per tree level, by dedicating a single processor to each level. Of course, this method cannot utilize more processors than there are tree levels. Wang's method creates the same number of nodes per level, which creates roughly the same capacity per level. Johnson and Colbrook's Path-to-root scheme enforces the balanced capacity rule as well. For every leaf node on a processor, the processor also holds every intermediate node on the path up to the root. Thus every processor that holds any leaf node also holds at least one node from each level, assuring, in our rough sense, equal processing capacity per level. In fact, the formula for the number of copies for random path-to-root given in section

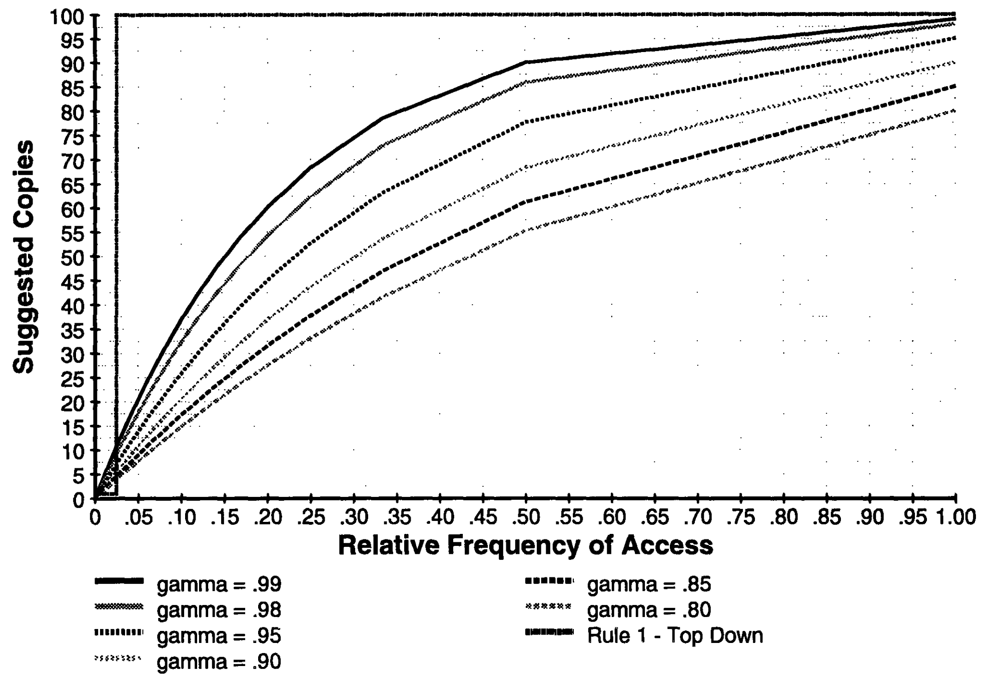


Figure 6-7: Suggested Copies versus Relative Access Frequency – Rule Two

2.2.1 is very similar to the formula for this rule. When we equate the two and solve for $\gamma$ we find they use the same amount of space per level when $\gamma = 1 - (1 - \frac{1}{C})^{\#leaves}$, that is, when the target capacity per level is equal to the capacity presented by the randomly placed leaves.

$$place(\frac{\#leaves}{n}, C) = C * (1 - (1 - \gamma)^{\frac{1}{n}})$$

$$C * (1 - (1 - \frac{1}{C})^{\frac{\#leaves}{n}}) = C * (1 - (1 - \gamma)^{\frac{1}{n}})$$

$$(1 - \frac{1}{C})^{\#leaves} = 1 - \gamma$$

$$\gamma = 1 - (1 - \frac{1}{C})^{\#leaves}$$

### 6.1.3 Hybrid Rule

Figure 6-8 shows throughput versus replication for the capacity balancing and top down replication rules. Neither rule is consistently better. The capacity balancing rule produces better results at low replication because it eliminates the bottleneck flat spots, but it yields poorer results as the amount of space used for replication increases because it makes more copies of

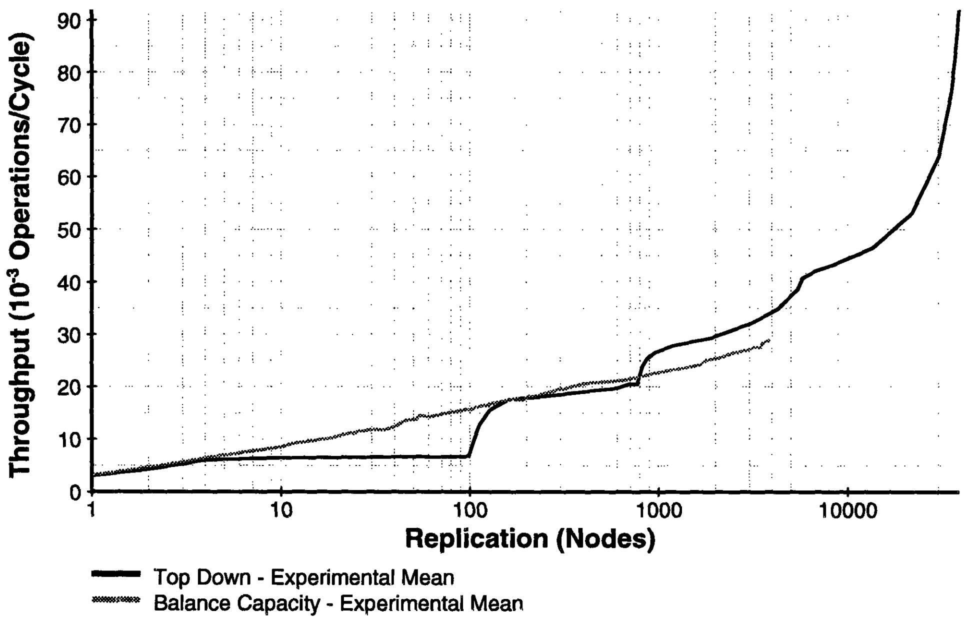


Figure 6-8: Throughput versus Replication – Rules One and Two

lower level nodes than are needed. The results of experimentation suggest that performance is improved by creating a hybrid of the two rules – adding some copies to lower levels in the tree to reduce bottlenecks, but providing more processing capacity for higher levels to also reduce remote messages.

To demonstrate the value of a hybrid rule we create a rule that meets this description – the rule is illustrative, it is not intended to be the theoretically optimal rule. This hybrid rule is based on the capacity balancing rule. We first calculate the replication recommended by that rule. If it is above a threshold (60%), we fully replicate the node. If the recommendation is below a second threshold (30%) we cut the replication half. When reducing the replication, however, we try to avoid reintroducing bottlenecks by never replicating less than the amount required to provide 90% of the capacity requested in the initial application of the capacity balancing rule. Mathematically, we first calculate $r'$ using the equation for $r$ given by the capacity balancing rule for a given $\gamma$, then calculate $r$ as follows:

$$r = \begin{cases} C & \text{if } r' > .6 * C \\ r' & \text{if } .3C < r' \leq .6 * C \\ max(\frac{r'+1}{2}, r'') & \text{if } r' \leq .3 * C \end{cases}$$

where $r'' = C * (1 - (1 - .9 * \gamma)^{\frac{1}{n}})$, the number of copies necessary to provide 90% of the desired capacity. Figure 6-9 shows, for this hybrid rule and our two previous rules, a representative graph of the number of copies per node, $r$, versus relative frequency of access. The graph for the hybrid rule has the desired shape – compared to the capacity balancing rule it provides more copies for frequently used nodes high in the tree, and fewer copies for less frequently used nodes lower in the tree.

Figure 6-10 shows the results of applying the hybrid rule, compared to the results of our two previous rules. With a few exceptions, the hybrid results generally track the maximum of the two other models – the hybrid results are similar to the capacity balancing results for low replication, and similar to the top down results for higher replication.

Using this (or similar) hybrid rule we can, for a given tree, determine how to use replication to either produce a desired level of throughput, or maximize throughput for a given amount of available space.

### 6.1.4 Additional Comments on Throughput and Utilization

Another characteristic of note on the graphs of throughput versus replication is the slight upward curve in the graph while increasing replication within each level. This is most noticeable as the leaves are replicated. Since remote messages are dropping approximately linearly with increasing replication, we might expect throughput to be increasing linearly as each level is replicated. The slight upward curve occurs because message sending and receiving overheads are not the only barrier to higher throughput – queueing delays are also dropping.

As tasks are independently routed between processors, queues will temporarily form at some processors while other processors may be temporarily idle. Figure 6-11 shows that average processor utilization increases as replication within the lower levels is increased. (The results from simulation differ from the queueing model predictions because our measurement of utilization in the simulation is rather crude.) As replication increases, operations stay on their home proces-

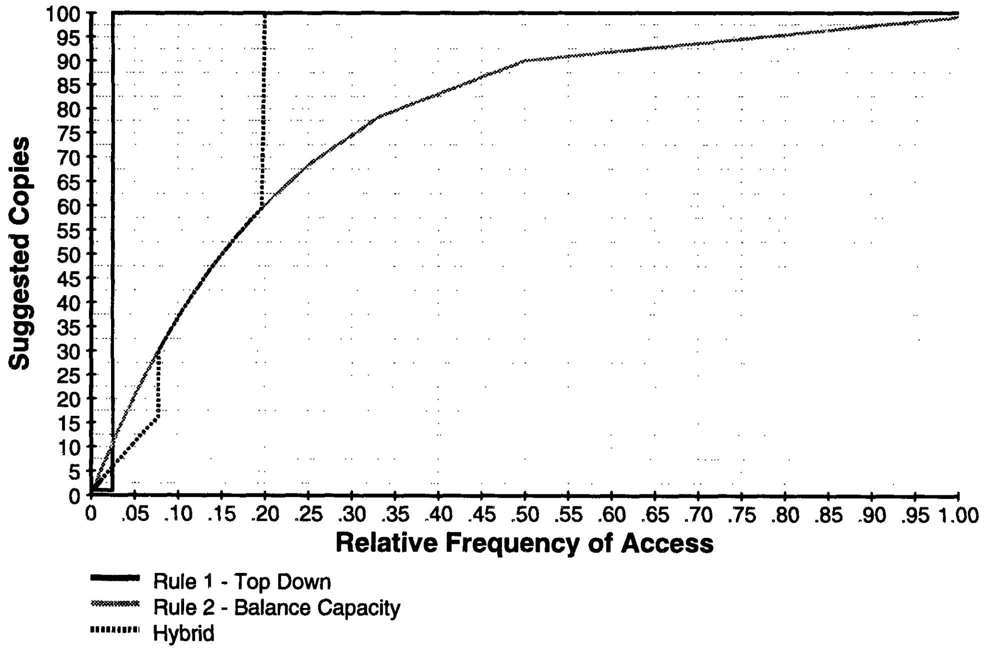


Figure 6-9: Suggested Copies versus Relative Access Frequency – Hybrid Rule

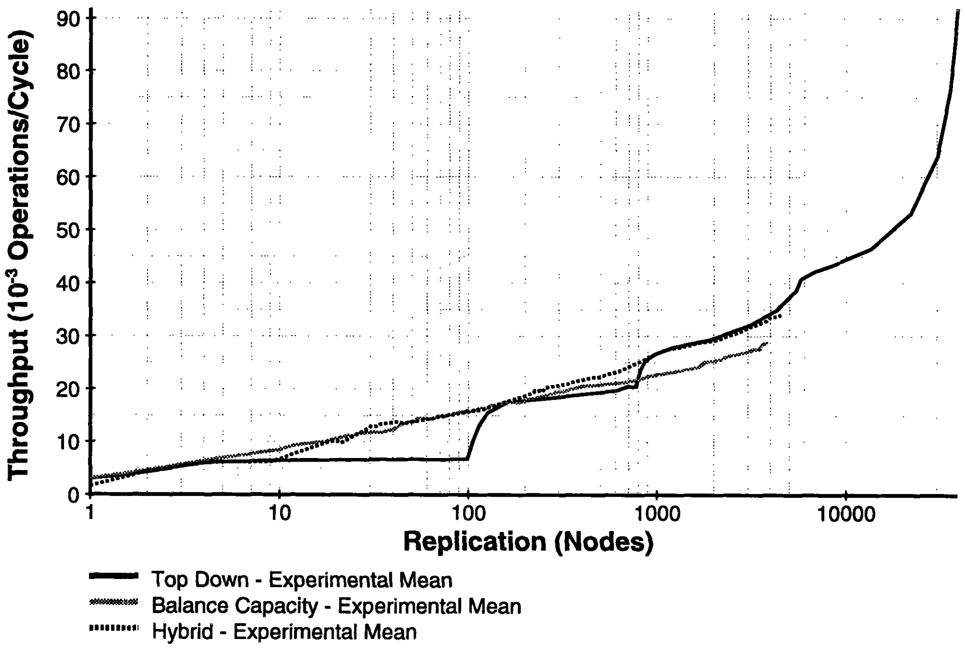


Figure 6-10: Throughput versus Replication – Hybrid Rule

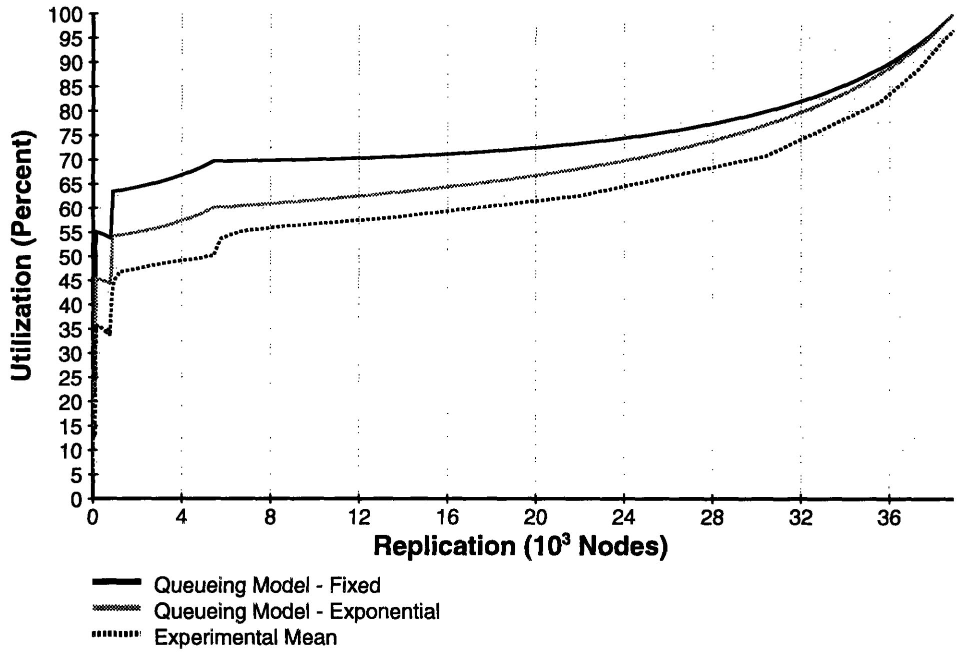


Figure 6-11: Processor Utilization vs. Replication - Base Case

sor longer, spreading work across the full set of processors, creating less queueing interference, and increasing average utilization. The upward curve in utilization, combined with the linear reduction in message overhead, produces the resulting upward curve in throughput.

Average utilization can also be increased by adding more tasks to the system. Figure 6-12 shows throughput versus replication for a system with 4 operations per processor (the same as figure 5-16 in the previous chapter, but using a linear scale). The greater operation load raises the overall utilization of the processors closer to 100%, so there is less possibility for utilization increase as replication increases. The result is as we might expect, the throughput increase during the replication of the leaf level is more nearly linear.

Figure 6-13, a graph of average processor utilization vs. replication using a log scale shows one of the more unexpected results of this work: as replication of the root increases in the presence of a bottleneck at the next level, overall system utilization drops. Although subtle changes are occurring in the system, the phenomenon has a simple explanation. Utilization is given by the product of throughput and service demand, $U = x * D$. In our simulations the bottleneck level is holding throughput essentially constant while the overall service demand is

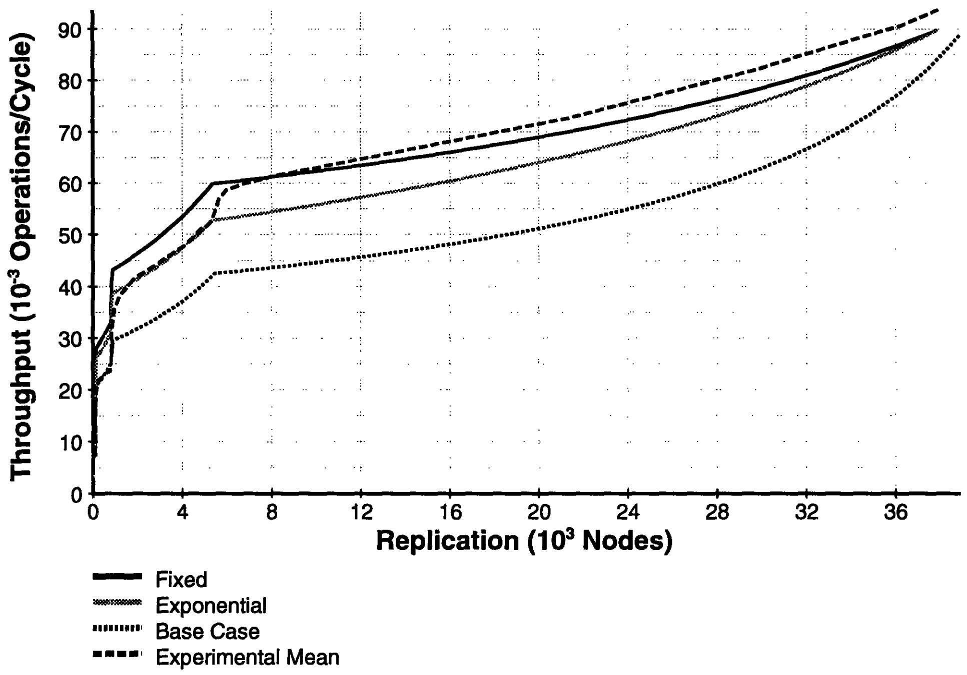


Figure 6-12: Throughput vs. Replication – 4 Operations Per Processor

decreasing, so utilization will also decrease.

### 6.1.5 Additional Comments on Random Placement of Nodes

A corollary to the fact that the number of processors covered by the random placement of $r$ copies of each of $n$ nodes is less than $r * n$ is that the processors holding nodes will not all hold the same number of nodes. This uneven distribution can reduce throughput as it will lead to uneven utilization of the processors – some will be busier than average and may create additional queueing delays, while others will have unused processing capacity. The probability distribution of the number of nodes, $x$, on a processor, given random distribution of $r$ copies of $n$ nodes is:

$$P(x) = \left(\frac{r}{C}\right)^x * \left(\frac{C-r}{C}\right)^{(n-x)} * \begin{pmatrix} n \\ x \end{pmatrix}$$

Figures 6-14 and 6-15 show the probability distribution of nodes per processor for a four level B-tree produced by our hybrid model, with 100 copies of the root, 57 copies of level 2, 6 copies of level 1, and only the original of the leaves. The first figure shows the probability

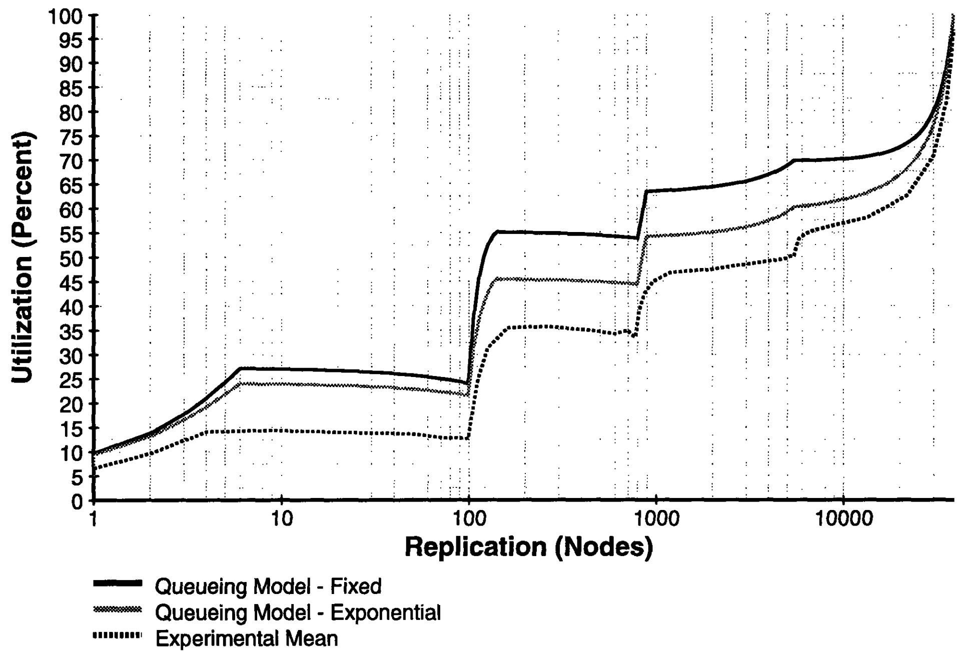


Figure 6-13: Utilization vs. Replication – Base Case, Log Scale

distribution for each level and the second for the combination of levels. They indicate that while, probabilistically, each processor will hold several nodes, some processors may not hold any node from a particular level and the overall distribution of nodes will be uneven. Given random placement, however, this uneven distribution cannot be avoided.

With the exception of Johnson and Colbrook's path-to-root placement, we have not explored alternative placement mechanisms that might create a more even distribution. In section 6.4, the comparison of random placement with path-to-root placement, we will provide additional evidence that suggests that an unequal distribution of nodes does reduce throughput.

## 6.2 Rules for Replication – Inserts and Deletes

When B-tree operations are limited to lookups only, our hybrid replication rule yields improved throughput with every added copy. When we allow insert and delete operations that modify the B-tree, replication starts to have a processing cost associated with it, namely when a node is modified all the copies of the node must be updated. Additional replication will then stop providing incremental benefit when the benefits of having copies of a node are exceeded by the

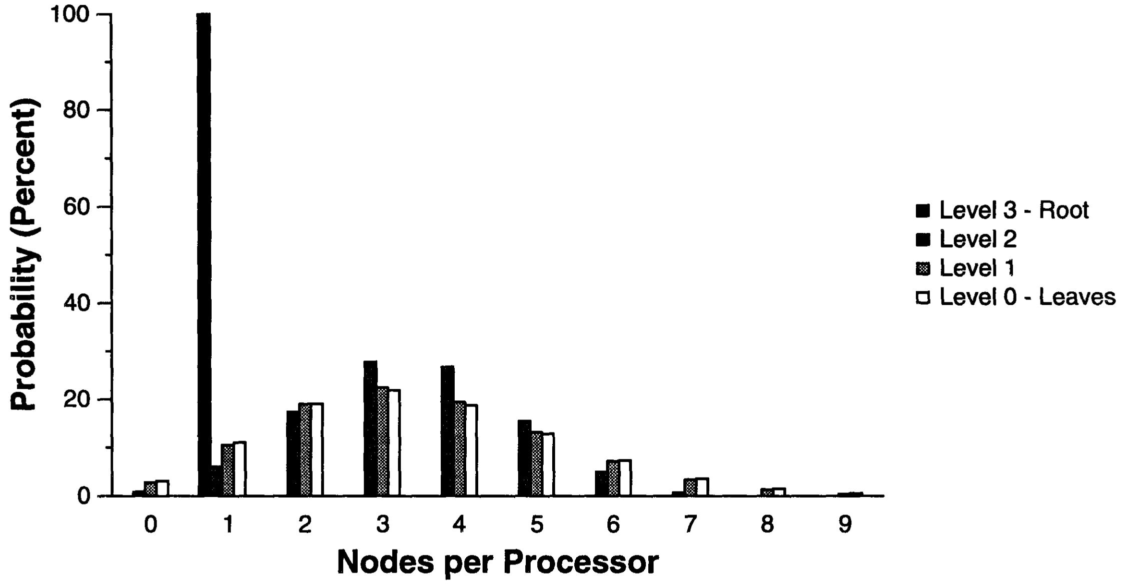


Figure 6-14: Distribution of Nodes Per Processor – Individual B-tree Levels

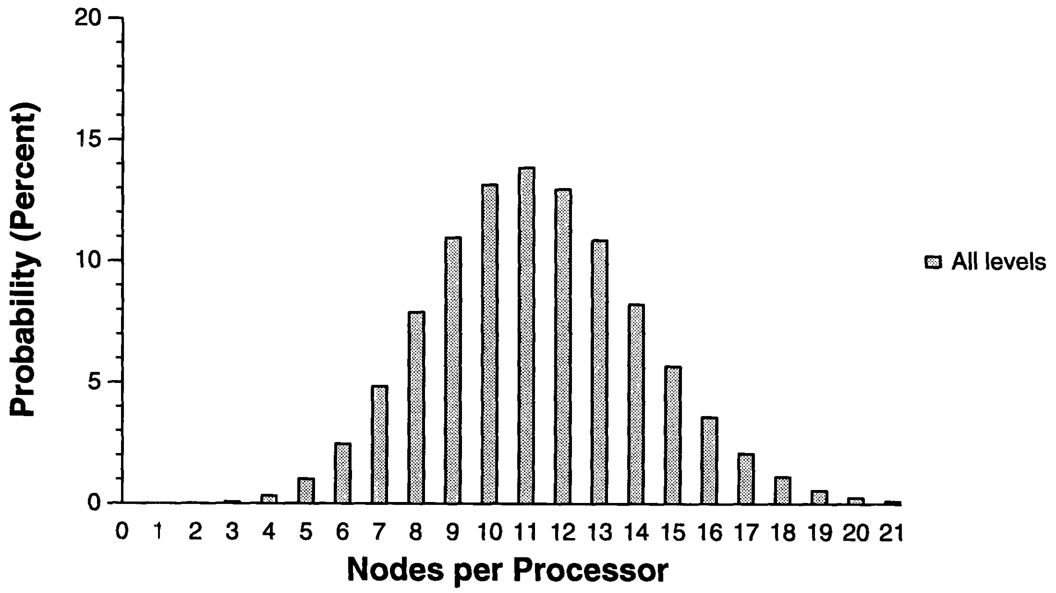


Figure 6-15: Distribution of Nodes Per Processor – All Levels Combined

overhead cost of updating the copies.

Conveniently, the benefits of replication tend to drop as we replicate nodes lower in the tree, while the costs of updating node copies increase. To approximate the point of tradeoff, we make simplifying assumptions for the benefits and costs of replication. For the benefit, we consider only the benefit of reduced messages, since the tradeoff point is usually low in the tree, where capacity is not a problem. We specifically ignore the increase in utilization that comes with replication.

As mentioned earlier, the probability that a remote message is required to reach a node with $r$ copies is $1 - \frac{r}{C}$; each additional copy of a node reduces the probability of that an operation using that node will need a remote message to reach it by $\frac{1}{C}$. If the costs of a single remote message is given by the processing overheads of sending and receiving messages, *mesg_ovhd* as defined in Chapter 4, the service demand benefit per operation referencing a node associated with adding a single additional copy is given by:

$$benefit = \frac{mesg\_ovhd}{C}$$

For the cost, we ignore the cost associated with splitting a node and use only the cost of updating. Using the equations from Johnson and Shasha we introduced in chapter 4, the increased service demand per operation that references a node associated with adding a single additional copy of the node is approximated by:

$$cost = \begin{cases} mod\_pct * update\_cost & \text{for } level = 0 \\ mod\_pct * (1 - 2 * del\_pct) * update\_cost * \frac{1}{branch\_factor^{level}} & \text{for } level > 0 \end{cases}$$

where *level* is the height of the node above the leaves.

Thus, replication at levels above the leaves continues to be a net benefit as long as:

$$\frac{mesg\_ovhd}{C} \geq \frac{mod\_pct * (1 - 2 * del\_pct) * update\_cost}{branch\_factor^{level}}$$

or

$$branch\_factor^{level} \geq \frac{C * mod\_pct * (1 - 2 * del\_pct) * update\_cost}{mesg\_ovhd}$$

or

$$level \geq log_{branch\_factor}\left(\frac{C * mod\_pct * (1 - 2 * del\_pct) * update\_cost}{mesg\_ovhd}\right)$$

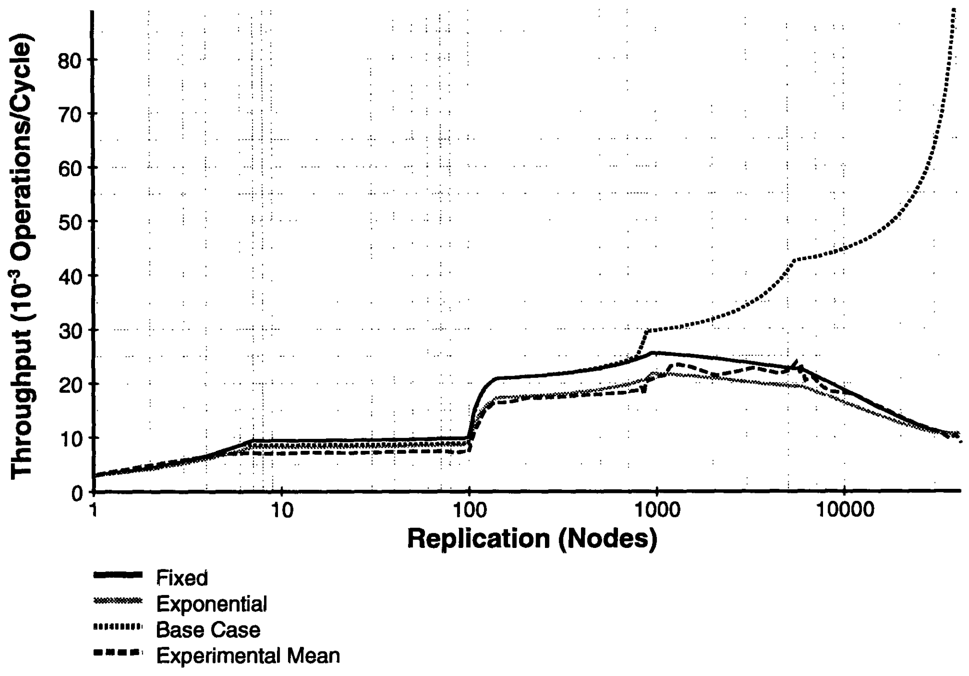


Figure 6-16: Throughput vs. Replication – Operation Mix 95/4/1

Given the costs from our simulation with operation mix 95% lookups, 4% inserts, 1% deletes, this works out to

$$level \geq log_7\left(\frac{100 * .05 * (1 - 2 * .2) * 1600}{500}\right) = log_7(9.6) = 1.16$$

As figure 6-16 confirms, in our simulations throughput is similar to the base case with lookups only for all but the lowest two levels of the tree.

This approximation suggests that, for a given operation mix, replication at lower levels in the tree can be made productive by either reducing the cost of update or increasing the branch factor. If we use the observed parameters of our simulations but change the average branch factor to 50 (producing a tree with $50^3 = 125,000$ leaves), the point of tradeoff changes to:

$$level \geq log_{50}\left(\frac{100 * .05 * (1 - 2 * .2) * 1600}{500}\right) = log_{50}(9.6) = 0.578$$

Figure 6-17 shows the queueing model results for a tree with branch factor 50. As expected, throughput continues to rise until replication of the leaves is started.

The results shown in figure 6-17 demonstrate why a detailed knowledge of the relationship between replication and performance is valuable. A 4 level tree with an actual branch factor of

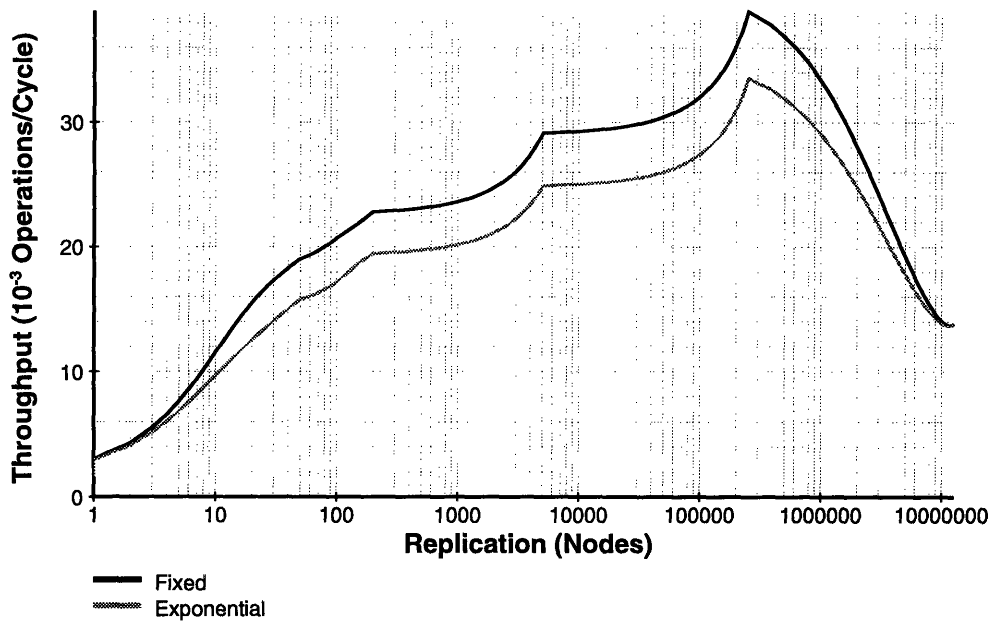


Figure 6-17: Throughput vs. Replication - Branch Factor 50, 95/4/1

50 per node consists of 127,551 nodes (1+50+2,500+125,000). Unreplicated on 100 processors, throughput would be around $3 * 10^{-3}$ operations/cycle. If we were to use Wang's guidelines, setting the replication factor equal to the branch factor, we would make 127,549 additional copies (99 of the top two levels and 49 of the level above the leaves). Figure 6-17 indicates this doubling of space used would increase throughput to about $3.2 * 10^{-2}$ operations/cycle. Alternatively, with only 200 copies added to the unreplicated tree, throughput can be raised to almost $2.3 * 10^{-2}$ operations/cycle, about 69% of the throughput increase requiring less than 0.2% of the additional space. The extra throughput may be necessary, but it comes at a significant cost.

## 6.3 Performance Under Non-Uniform Access

Thus far our models and simulations have included the assumption that the distribution of the search keys selected for B-tree operations is uniform. In this section we remove that assumption and examine B-tree performance using the replication rules developed earlier in this chapter.

We replace the uniform search key distribution with a distribution limiting search keys to a

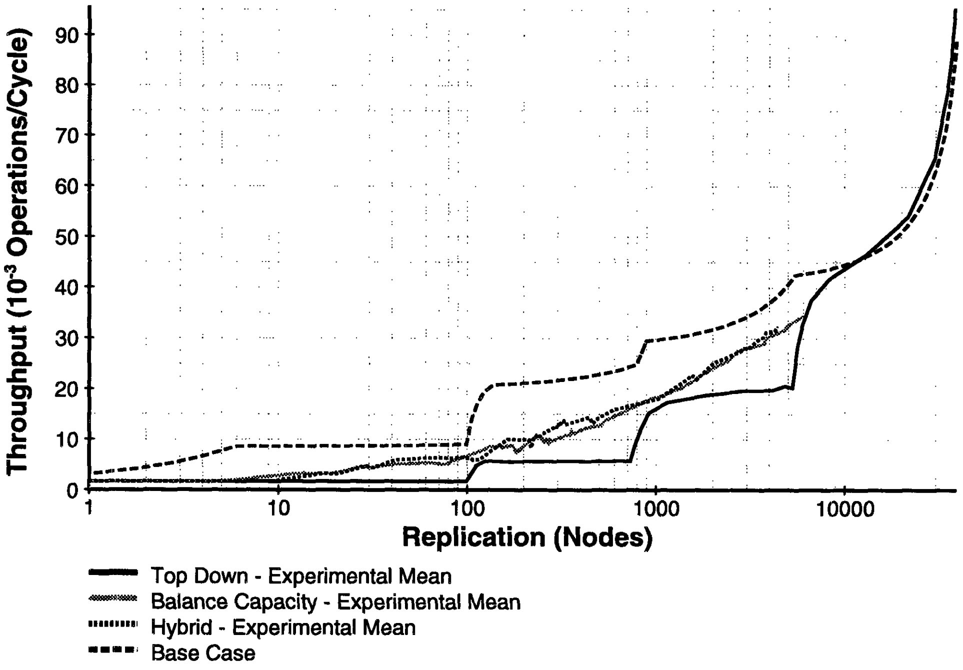


Figure 6-18: Throughput vs. Replication – Access Limited to 10% of Range

range containing only 10% of the search key space. Within this limited range the distribution remains uniform. We do not suggest that this is representative of any real access pattern. Our objective is to introduce some form of non-uniformity and study the results.

Figures 6-18 shows the throughput versus replication for our three replication rules under only lookup operations. These are shown with the queueing network model prediction for our base case simulation.

As might be expected, when using the top down rule we continue to encounter noticeable capacity bottlenecks until we do much more replication than with the uniform key distribution of the base case. Since we are limiting access to roughly 10% of the nodes at each level, we must make more copies of lower level nodes to create adequate throughput capacity from nodes actually used. The capacity balancing and hybrid replication rules once again do a better job of utilizing space to avoid severely limiting bottlenecks.

All three replication rules exhibit performance significantly poorer than the base case. Only when the upper levels of the tree are fully replicated and the leaves are partially replicated does the throughput for space used match that of the base case.

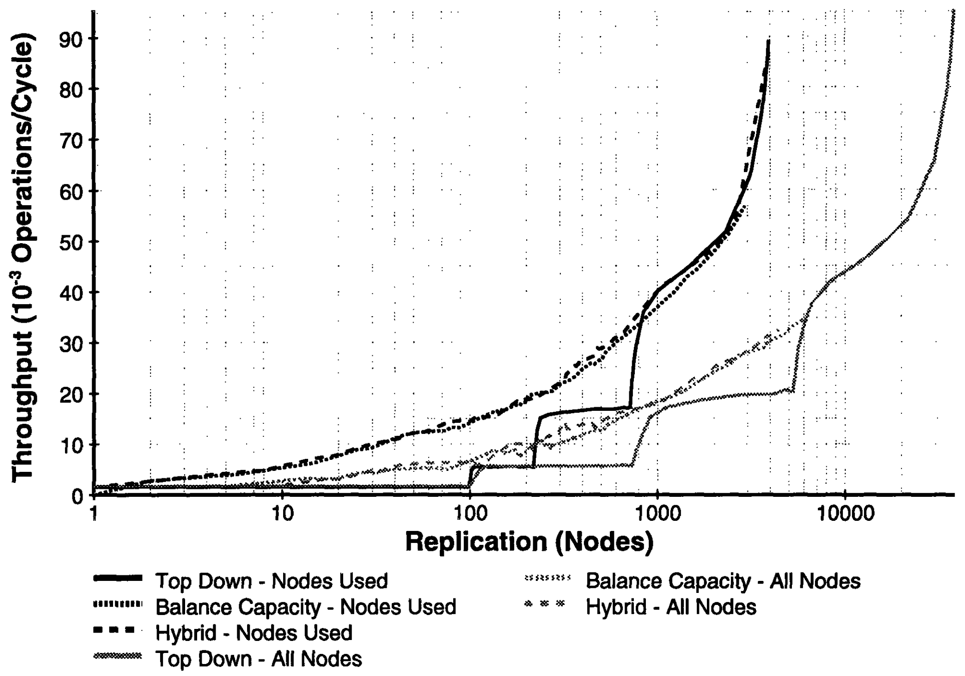


Figure 6-19: Throughput vs. Replication - Access to 10%, Copy Only Nodes Used

Performance is reduced because these replication rules waste space on copies of nodes that are never used. With our non-uniform access pattern, roughly 90% of the copies (not including copies of the root) are never used. If we were to make copies only of nodes that are actually used, we might expect to achieve the same throughput levels using roughly one-tenth the replication. Since the root is always used, we actually expect to achieve the same throughput using one-tenth the replication of nodes below the root level.

Figure 6-19 shows the results when copies are restricted to the nodes that hold keys in the 10% range. Maximum throughput of 0.09 operations per cycle is reached with roughly 4000 copies, not the nearly 40,000 copies required in the previous case. This is consistent with our one-tenth expectation. The one-tenth expectation appears to hold at lower replications as well. For example, when copying is limited to nodes used, throughput reaches .02 at about 220 copies. This same throughput is reached at about 1100 copies when all nodes are copied. Removing the 100 copies of the root, gives 120 and 1000 copies below the root, near the one-tenth we might expect.

Limiting copying to nodes actually used also translates to greater throughput for a given amount of space. In general, this limiting leads to throughput 1.5 to 2 times higher than replication of all nodes, for the same amount of space.

These experiments suggest that throughput can be significantly enhanced if the replication pattern can be adapted to the actual usage pattern. In the next chapter we examine mechanisms to dynamically reconfigure the replication of a B-tree in response to observed changes in access pattern.

## 6.4 Comparison with Path-to-Root

Johnson and Colbrook have proposed a different scheme for static replication of B-tree nodes that we refer to as "path-to-root". Their rule for placement of copies is: for every leaf node on a processor, all the ancestors of that leaf should also be copied on that processor. Their rule for placement of original nodes is not as fully developed. They propose the ideal of having sequences of leaf nodes on the same processor. This would minimize the number of copies of upper level nodes (many, if not all, descendants might be on the same processor), but require a mechanism to keep sequences of leaves together and balance the number of leaves across processors as the tree grows dynamically. Johnson and Colbrook are developing the dE-tree (distributed extent tree) for this purpose.

We have not implemented their scheme to build and maintain B-trees using the dE-tree, but we can synthetically create a B-tree that looks like their tree and test performance under lookups only. We first build an unreplicated B-tree in one of two ways:

- Ideal placement model – Entries are added to the tree in increasing order, so that the right-most leaf node always splits. To create 70% utilization of the leaves, the split point in a node is adjusted from 50/50 to 70/30. The number of leaves per processor, $l$, is calculated in advance so that the first $l$ can be placed on processor 1, the second $l$ on processor 2, and so on. When a new parent must be created it is placed on the processor of the node that is being split. For simulations of this model we perform 2,800 inserts to create a B-tree with 400 leaves of 7 entries each, 4 leaves per processor.

- Random placement model – New leaf nodes are placed randomly, but when a new parent is created it is placed on the processor of the node that is being split.

|  | Replication | Throughput |
|---|---|---|
| Ideal P-T-R | 282 | 0.0260 |
| Random P-T-R | 708 | 0.0234 |
| Hybrid Random | 711 | 0.0244 |

Figure 6-20: Path-to-Root Comparison – Throughput for Uniform Access

After the unreplicated tree is built, we make additional copies of each node to satisfy the "path-to-root" criteria. Because the ideal placement model places logically adjacent leaf nodes on the same physical processor, it uses significantly less space for replication than the random placement model. For example, given a node just above the leaves with 7 children, if all 7 leaf children are on the same processor, there will be only one copy of the parent. If the 7 children were randomly placed on 7 different processors, there will be 7 copies of the parent.

The results of simulation are shown in figure 6-20, along with results for our random placement method using the hybrid rule to use approximately the same space as the path-to-root random placement model. (The number of copies per level was 100, 57, 6, and 1, moving from root to leaves.) As before, the results shown are the means of five simulations. The ideal path-to-root model produces the best throughput result and requires significantly less space than the other two models. The throughput for the hybrid model is about 6% lower than the ideal path-to-root model produces.

The throughput of the random path-to-root model is reduced from what it might be because not all processors hold tree nodes. As indicated in chapter 2, the B-tree created with 2,400 inserts is expected to place leaves on only 97 processors. We can estimate the throughput that might be obtained if all 100 processors were used by assuming the simulation produced 97% of the potential throughput. This increases the throughput for random path-to-root to 0.0241 operations/cycle, within 1% of the throughput for the random hybrid.

It is no coincidence that the throughput in all three cases is similar. The characteristics that affect throughput, the capacity of each level and the number of remote messages per operation, are similar. In all three cases there are sufficient copies of nodes at each level to eliminate any capacity bottlenecks. We can calculate the expected number of remote messages per operation to compare the different rules. We define the number of copies (including the original) of the parent node and a child as $r_{parent}$ and $r_{child}$ respectively. For the two versions of the path-to-

|  | P-T-R Ideal | P-T-R Random | Hybrid Random |
|---|---|---|---|
| Root | 0 | 1-(97/100)= .03 | 0 |
| 1 | 1-(13/100) = .87 | 1-(39/97)= .6 | 1-(57/100)=.43 |
| 2 | 1-(2.5/13) = .81 | 1-(7/39) = .80 | 1-(6/100) = .94 |
| Leaf | 1-(1/2.5) = .6 | 1-(1/7) = .86 | 1-(1/100) = .99 |
| Return | 1-(1/100) = .99 | 1-(1/100) = .99 | 1-(1/100) = .99 |
| Remote messages | 3.27 | 3.28 | 3.35 |

Figure 6-21: Path-to-Root Comparison – Average Number of Remote Messages

root algorithm, the likelihood of not changing processors when moving between two levels is $\frac{r_{child}}{r_{parent}}$, so the probability of a remote message is $1 - \frac{r_{child}}{r_{parent}}$. When copies are placed randomly, the likelihood of there being a child located on the same processor as a parent, requiring no remote message is $\frac{r_{child}}{C}$. The probability of a remote message is therefore $1 - \frac{r_{child}}{C}$. For all three approaches, the probability of needing a remote message to return the final result is $1 - \frac{r_{leaves}}{C}$.

Figure 6-21 uses these probabilities to show the expected number of remote messages per operation for each of the three replication rules, given a uniform key distribution. The two path-to-root models use remote messages at different levels in the tree, but use roughly the same number of remote messages. Our random placement model uses approximately 3% more remote messages.

The calculations of the expected number of messages suggests that the two path-to-root models should produce nearly identical results. That they do not is attributed to the fact that random path-to-root does not distribute leaves evenly across the processors. Using only 97 processors limits reduces throughput, and so does having an uneven distribution on those processors. When we place leaf nodes sequentially (i.e., the first to be created on processor 0, the second on processor 1, ...) we can create a more even, but still random, distribution of leaf nodes. Five simulations of this placement rule produces an average throughput of 0.0263 operations/cycle, about 1% greater than our results for ideal path-to-root. This is a 12% improvement over the "fully random" path-to-root, and a 9% improvement over the results adjusted to utilize all processors. This suggests that the uneven distribution of nodes can significantly reduce performance and that methods for balancing the data and processing load should be explored.

For the particular case just examined the path-to-root placement scheme is competitive

| | Replication | Throughput |
|---|---|---|
| Ideal P-T-R | 282 | 0.0068 |
| Random P-T-R | 685 | 0.0125 |
| Hybrid Random | 750 | 0.0179 |

Figure 6-22: Path-to-Root Comparison – Throughput for Access Limited to 10% of Range

with, or preferable to, our random placement method. This is not, however, always the case. For example, the ideal path-to-root algorithm requires maintenance of leaf placement that we have not estimated. Both the ideal path-to-root and the random path-to-root algorithms will frequently require multiple copies of nodes just above the leaves, which we have shown can be detrimental to performance when the operation mix includes inserts and deletes. Perhaps the greatest weakness, however, is that the copy distribution pattern is fixed and is based on the tree structure, not the access pattern. No adjustments can be made to use more or less space to make a tradeoff between the cost of replication and the benefit of increased performance, or to dynamically change the use of space. Figure 6-22 shows the results of performance under a non-uniform access pattern, again limiting access to only 10% of the key space. The performance of the ideal path-to-root placement model suffers dramatically. The placement of sequences of leaf nodes on the same processor now has a cost – only 10% of the processors are actually used for levels below the root. The performance of the random path-to-root placement model suffers from a similar fate – only the processors that hold the 10% of the leaf nodes being accessed can participate in processing operations below the root level. For this simulation this is about 34 leaves, so at most 34 (of 100) processors are used. Our hybrid random placement model offers significantly better performance than either path-to-root method, but, compared to the results shown in figure 6-19 provides only half the throughput for the amount of space used that it could provide if only nodes actually used are replicated.

These comparisons show that replication control rules such as our hybrid model are more generally useful than the path-to-root models. While the ideal path-to-root model can produce higher throughput for lower space used under ideal conditions, our hybrid model yields much higher throughput when conditions vary from the ideal. This is particularly true if our hybrid model can be extended to create replication to match an observed access pattern.

These comparisons have also suggested that the results produced by our hybrid model would be improved if the random placement of nodes and copies could be replaced with a placement method that can more evenly distribute load across the processors. We do not explore that path in this work.

## 6.5 Summary

In this chapter we have analyzed the relationship between replication and throughput to develop a new rule for the use of replication. We have gone past the intuition of prior work to produce guidelines for near-optimal use of whatever amount of space is available for replication. The result is based on the relative frequency of access to each node, as intuition and prior work suggest, but also includes a slight bias to replicate nodes at higher levels of the B-tree where the marginal additional copy can have the greatest contribution to reducing inter-processor messages. We have also examined the impact of adding inserts and deletes to the B-tree operation mix and indicated the transition point where the value of replication is overcome by the costs of updating copies of tree nodes.

Through simulations using non-uniform access patterns we have shown that our hybrid random placement rule is preferable to the alternative path-to-root rules. We have also shown that performance is improved if the use of replication can be made to match the actual pattern of accesses.

In the next chapter we will apply the results of this chapter to replicating B-tree nodes based on observed frequency of access, not assumptions of uniform branch factor and uniform access pattern.

# Chapter 7

# Dynamic Replication

In the previous chapter we showed that distributed B-tree performance can be improved when the number of copies of a particular B-tree node is made to depend on the relative frequency of access to the node. In that chapter we modeled the frequency of access to a node as a static function of its level in the tree. In this chapter we explore dynamic control of replication based on observed frequency of access to nodes. We introduce a simple approach to dynamic control and explore its ability to produce performance similar to that from our capacity balancing and hybrid algorithms under a static access pattern, and its ability to change replication in response to a change in access pattern. We cannot compare behavior across all access patterns or suggest that one particular pattern is more valid for comparison than any other – our objective is to introduce one approach and identify the challenges to efficient dynamic control of replication.

In this chapter we first describe a simple dynamic caching algorithm, then present the results of simulations that demonstrate the algorithm's ability to perform as desired. We also introduce the initial results from an update to this simple algorithm.

## 7.1 Dynamic Caching Algorithm

Replication control is a caching problem. Additional copies of a B-tree node are cached on one or more processors to improve the overall performance of the system by eliminating tree node and processor bottlenecks. Dynamic control of this caching must address three questions: how many copies of a tree node to create, where to place the copies (including local management of caches), and how to re-map copies of a parent node to copies of a child when the replication of one of them changes.

The simple approach to dynamic caching developed in this chapter uses a fixed size cache for each processor to hold copies, and an access count associated with each copy of a B-tree node to estimate frequency of access and determine which nodes should be copied. Replication decision making in this algorithm is decentralized – each copy of a node determines independently whether it should request the creation of additional copies, and the processor holding the "master" copy of a tree node determines where it should place any new copies, independently of placement decisions for all other nodes.

### 7.1.1 Copy Creation

The results of the previous chapter indicated that the replication for a B-tree node should be determined as a function of relative frequency of access to the node. The results indicated that the optimal replication is a slight variation from directly proportional to relative frequency of access – slightly more copies of more frequently used nodes, and slightly fewer copies of less frequently used nodes. The ultimate objective should be to create this replication pattern with dynamic control. However, it is difficult to calculate the frequency of access to a B-tree node. First, frequency cannot be directly measured at a point in time, but must be observed as an average over time. Second, accesses are likely to be made to more than one copy, so no single processor can directly observe the full set of accesses and overall frequency. As a result, with this simple algorithm our goal is less ambitious than achieving the optimal replication pattern. We want only to establish a nearly proportional relationship between relative access frequency and replication and study its characteristics.

We model relative frequency of access by including an access count with every copy of a node and defining two parameters to link the access count to changes in replication, an access threshold and an access time lag. A copy's access count is incremented each time the copy is used, if the time elapsed since the previous access is less than the time lag. If the time between accesses is greater than the time lag, the access count is decremented, but never decremented below zero. When the access count reaches the threshold, an additional copy of the node is requested.

For the replication of a node to be increased, there must be a sufficient number of accesses to a single copy of the node within a limited period of time, i.e., the observed frequency of access must, at least temporarily, be above a certain rate. The two parameters establish the characteristics of access frequency necessary to create additional copies. They also help control the overhead required to perform dynamic replication control. The time lag establishes a frequency of access necessary for additional copies of a node to be created and eliminates the slow accumulation of access count over a long period of time. The access threshold defines how long a frequency of access must be observed to have an effect on replication; a larger threshold can reduce the frequency of copy creation and the associated overhead.

### 7.1.2 Copy Placement and Cache Management

When the access count of a copy of a B-tree node reaches the access threshold, the access count is reset and a request to create an additional copy is sent to the processor holding the "master" copy of the tree node. As with our static placement model, the additional copy is placed by selecting at random a processor that does not hold a copy.

Each processor has a fixed size cache for holding copies of nodes. When a processor receives a node to add to its cache, it must allocate an unused cache entry or discard a currently cached node. (When a cache entry is discarded, the "master" copy of the node is notified.) For the simulations in this chapter we manage the cache with a replacement algorithm developed by the MULTICS project [Cor69], sometimes called second chance, [PS85], clock or marking replacement. In this algorithm a cache entry is "marked" every time it is used. A pointer points to the last cache location discarded. When an empty cache location is needed, the pointer is advanced, wrapping back to the beginning like the hands of clock when it reaches the end of the cache. If an unmarked entry is found, its contents (if any) are cleared and the entry returned to the requester. If the entry is marked, the algorithm "unmarks" the entry, but does not immediately discard it (the entry is given a "second chance") and instead advances the pointer to check the next entry. If all entries are marked when the replacement algorithm starts, the first entry to be unmarked will eventually be the entry discarded. We have also implemented least recently used cache replacement, with similar results.

### 7.1.3 Parent/Child Re-mapping

As copies of B-tree nodes are added and discarded, the mapping between copies of parents and copies of children must be updated to retain a balanced distribution of work across the processors. This is the most expensive part of dynamic replication control. When the replication of a node is changed, it may be necessary to notify every processor that holds a copy of the parent of a change in mapping. In addition, it may also be necessary to inform every processor holding a copy of the node itself of a change in mapping to copies of its children. Further, if a node does not know all the locations of its parent and children (as we assume it does not), it must rely on the "master" copies of the parent and children to perform the re-mapping. This requires calculation and many inter-processor messages. To minimize overhead, when replication of a node changes we only update the mapping between the parent and the node being changed; the mapping between the node and copies of its children does not change. Instead, when a new copy is created it will be mapped to use the "master" copy of any children it might have. The parent of the node, however, is informed of the updated set of locations and re-maps its own copies to the copies of the changed node.

This approach to re-mapping was developed to minimize the amount of information about copy location that is distributed around the system. While this is adequate for the small number of replication changes that occur under static replication (replication changes only when nodes are created), it has significant weaknesses when used dynamically. In section 7.4 we introduce an improvement to this approach to re-mapping that allows the "master" copy of a node to know the location of all copies of its parent. This eliminates the need to involve the "master" copy of the parent node when the re-mapping must change.

### 7.1.4 Root Node Exception

The one exception to these rules is that the root node is distributed to all processors and is forced to stay resident in every cache. This provides two major benefits. First, no processing cost is incurred in replicating the root (except for distributing updates when the contents of the root change). There is no need to maintain access counts, request additional copies or re-map when a copy is added or removed. Second, the home processor of the root node does not have to be involved in re-mapping copies of the root to copies of its children, since each child knows that the root is on all processors. The result is the elimination of a potentially severe bottleneck on the home processor of the root tree node.

This exception also helps assure that, as our hybrid model recommended, nodes with a high frequency of access (e.g., the root) are replicated more fully than in direct proportion to observed relative frequency of access.

## 7.2 Dynamic Caching – Proof of Concept

For the simulations in this chapter we use the same system configuration and initial B-tree as described in Chapter 5: 100 processors supporting a B-tree with a node branch factor of 10, instantiated by 2400 randomly selected insert operations. In each simulation we test performance under a uniform access pattern and then change to a pattern with access limited to 10% of the search space. Each simulation consists of five phases:

1. The tree is constructed,

2. With dynamic caching enabled, a series of 100,000 lookup operations is executed using a uniform lookup distribution,

3. With dynamic caching temporarily disabled, a series of 10,000 lookup operations is executed using a uniform lookup distribution,

4. With dynamic caching enabled, a series of 100,000 lookup operations is executed with access limited to 10% of the search space,

5. With dynamic caching disabled, a series of 10,000 lookup operations is executed with access limited to 10% of the search space,

This structure allows us to test the performance of the algorithm during transitions and at steady state. It also allows us to test the performance possible from the tree constructed by the dynamic algorithm with the overhead of further changes temporarily turned off.

Before studying the behavior of the algorithm in detail, we first provide evidence that the algorithm can dynamically produce results close to the throughput we seek. We perform simulations using a per processor cache size of 3 and of 10. A cache of size 3 is relatively

| Search Key Distribution | Cache Size = 3 | | Cache Size = 10 | |
|---|---|---|---|---|
| | Hybrid | Cap. Bal. | Hybrid | Cap. Bal. |
| Uniform | .021 | .020 | .027 | .023 |
| Limited to 10% | .022 | .022 | .040 | .036 |

Figure 7-1: Target Throughput (Operations/Cycle)

small, but large enough to hold two entries beyond the root. A cache of size 10 is relatively large, allowing 1000 additional copies beyond the original tree of around 400 nodes. Figure 7-1 shows the throughputs we might hope to realize, based on the results of the hybrid and capacity balancing algorithms developed in Chapter 6, and shown in figures 6-10 and 6-19. The numbers in figure 7-1 represent the estimate for mean throughput when using 300 and 1,000 copies, as distributed by the two algorithms. For the cases where access is limited to 10% of the search space, we provide the results obtained when copies were made only of nodes actually used. We should expect results similar to the capacity balancing algorithm since our dynamic algorithm, like the capacity balancing algorithm, is attempting to create copies roughly in proportion to relative frequency of access.

Figure 7-2 shows the results of a single experimental run with cache size 3, time lag 5,000, and access threshold of 70. We use the number of operations completed for the x-axis rather than time to aid comparison between different simulations. During the first series of 100,000 lookups, the measured average throughput was 0.0145 operations per cycle (not shown). After the initial ramp up as the cache begins to fill and throughput has stabilized, the average is between 0.016 and 0.017 operations per cycle. Inefficiencies or overhead of the algorithm appear to hold throughput below our targets. When the caching algorithm is turned off the measured average for the 10,000 lookups rises to 0.207, very close to our expected targets of around 0.021.

When the access pattern is limited to 10% of the search space, the algorithm requires over 40,000 operations to adjust its use of replication for throughput to rise above the peak level it had reached under uniform access. At steady state this simulation produces throughput of around 0.023 operations per cycle. With dynamic caching turned off, throughput rises to over 0.026 operations per cycle. These are both actually greater than the targets set by our static algorithms. There are two possible reasons for this better than expected performance. First,

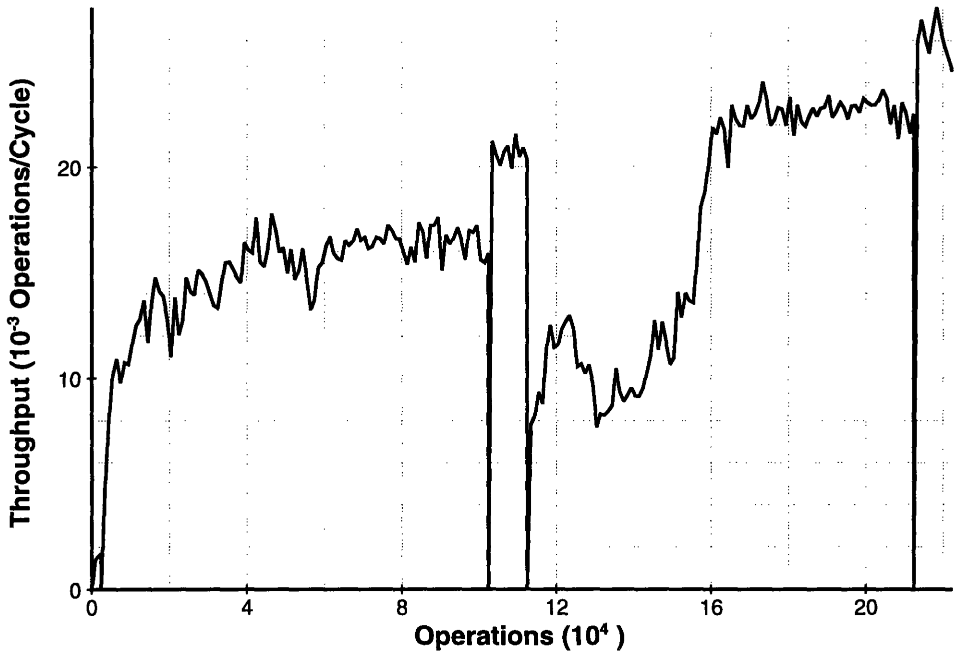


Figure 7-2: Throughput vs. Operations, Cache Size = 3

these dynamic results come from a single run, and results will vary for different runs. Second, and more interesting, in chapter 6 we made an equal number of copies of each node at a level. With this dynamic control, we can vary the number of copies by individual node, tailoring capacity node by node, not just per level.

Figure 7-3 shows the results of a single experimental run with cache size 10, time lag 10,000, and access threshold of 40. Under a uniform access pattern, throughput rises to just below 0.019 operations per cycle. Throughput goes up to around 0.023 operations per cycle when caching is turned off, at the target set by the capacity balancing algorithm, but below that of the hybrid algorithm. When access is limited to 10% of the search space, only around 20,000 operations are required before throughput exceeds the peak throughput reached under uniform access. Steady state throughput is around 0.027 operations per cycle, rising to 0.037 when caching is turned off.

These results indicate that the dynamic control algorithm can produce replication patterns that provide the performance of the target static algorithms. Although the results with dynamic caching enabled are always below the target, it is helpful to put this in perspective. With access

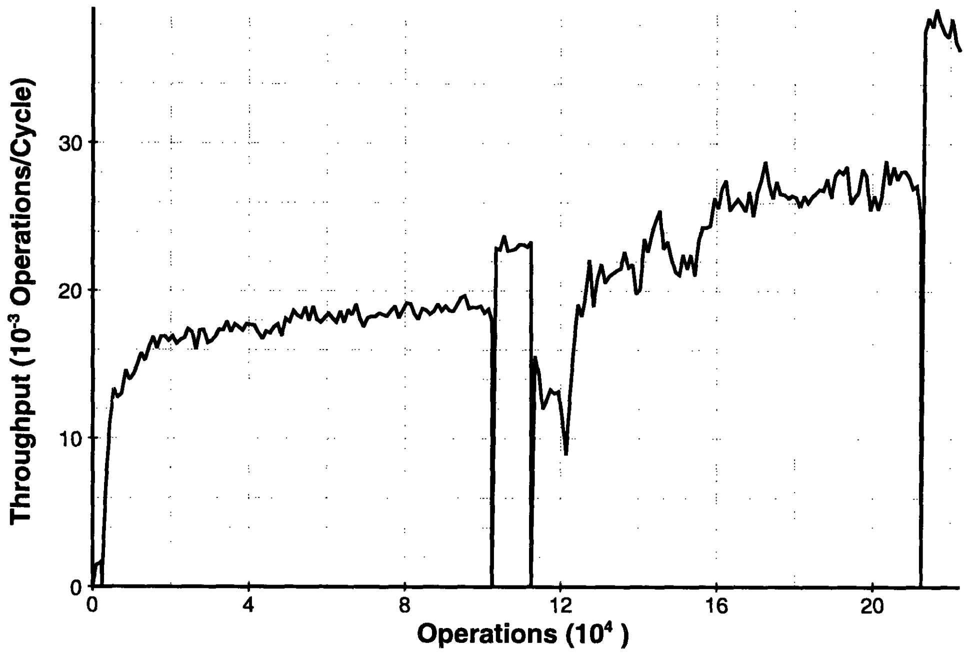


Figure 7-3: Throughput vs. Operations, Cache Size = 10

limited to 10% of the search space and replication of 1,000, our static hybrid algorithm achieved throughput of around 0.040 operations per cycle if it replicated only nodes actually being used. But when it had no specific usage knowledge and used replication optimized for uniform access it produced a throughput of around 0.019 operations per cycle. This dynamic algorithm produces steady state results for the limited access of around .027 operations per cycle, below the best results of hybrid model, but significantly better than the most general static model assuming uniform access. The challenge for the future is to reduce the overhead required to support dynamic caching and raise performance closer to that of the best of the static algorithms.

## 7.3 Simulation Results and Analysis

For more detailed analysis of the behavior of this dynamic caching algorithm we restrict our simulations to a system with a cache size of 10 and explore how the values of the parameters affect performance and the use of the cache space. Our objective is to identify opportunities to improve this algorithm or create better dynamic replication control algorithms. Clearly, one important way to improve the algorithm is to reduce the overhead required to manage dynamic

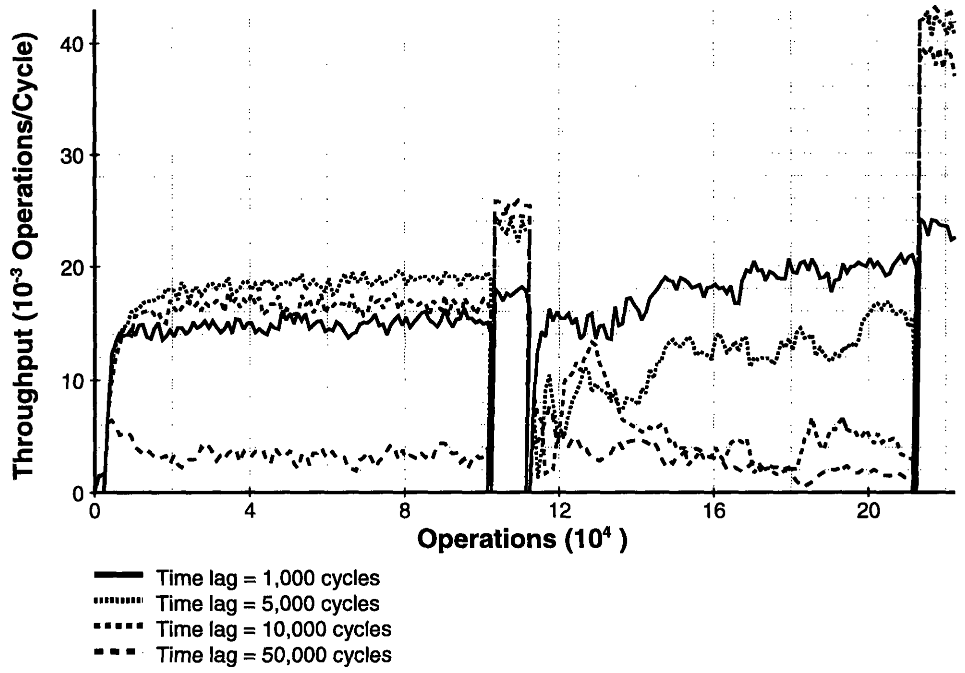


Figure 7-4: Throughput - Access Threshold = 5

caching. In this section we explore the impact of adjusting the parameters that can affect overhead. In the next section we introduce a modification to the re-mapping algorithm that can improve this basic algorithm by reducing the potential for bottlenecks.

Figure 7-4 shows the throughput results for several different values of time lag and a relatively low access threshold of 5. Figure 7-5 shows the total number of cache entries used for the same simulations. For the lowest time lag, 1000, throughput is relatively high and remains fairly constant, but as figure 7-5 shows, the cache never fills up. Thus, although this time lag provides good performance when caching is active, when dynamic caching is turned off throughput is much lower than that produced by larger values for time lag. At the other extreme, a long time lag of 50,000 allows all cache space to be used, but appears to create so much cache control overhead that the throughput is very low. The long time lag means that almost all accesses increase the count; combined with the low threshold value the rate of requests for additional copies must rise, with accompanying overhead.

Figures 7-6 and 7-7 show results for similar simulations with the access threshold set to 20. Once again, when the time lag is 1,000 the performance is consistent, but the cache never comes

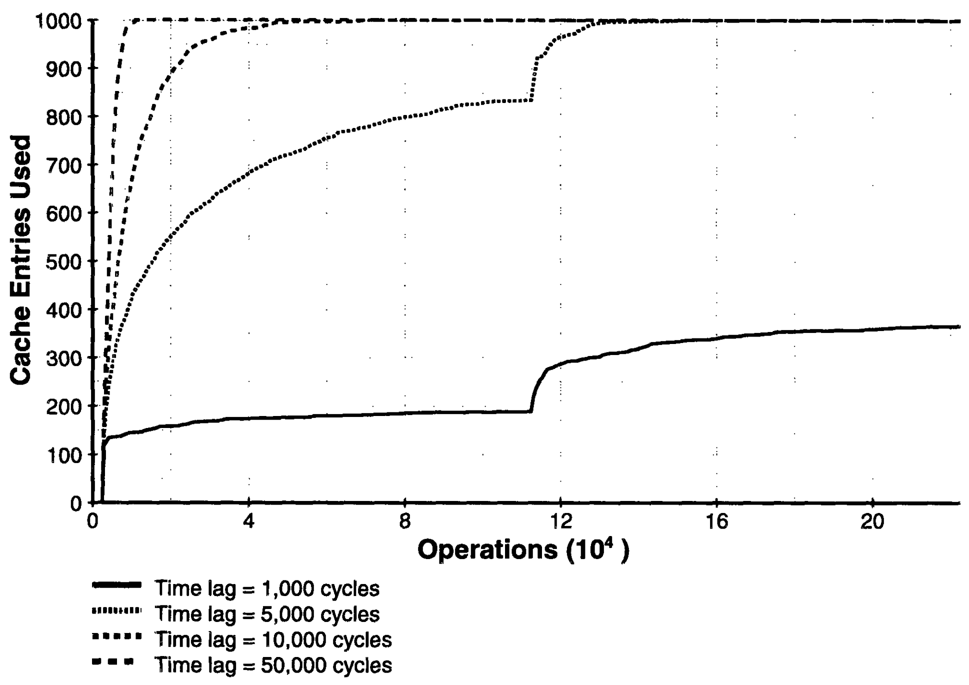


Figure 7-5: Cache Usage – Access Threshold = 5

close to filling up. With this very low time lag, throughput is consistent because there is not much caching overhead – but there is not much use of the cache either. The higher threshold has the effect we expect, the performance for the other three values of time lag has improved in response to the reduced frequency of requests for new copies. This is particularly noticeable for the part of the simulation with access limited to 10% of the search space.

In this set of simulations, dynamic performance is best for time lag = 5,000. Figure 7-7 shows that in the first 100,000 accesses the caches were not filled up and that net additions to the caches were tapering off. During the second 100,000 accesses, with access limited to 10%, the caches are becoming nearly full, with net additions to the caches again tapering off. This tapering off of net changes before the caches fill up, combined with the comparatively small jump in throughput when caching is turned off, indicate that there is significantly less caching overhead than in the simulations where the cache can be filled up quickly. As might be expected, performance is best when the cache space can be productively used without significant changes in its contents.

Figures 7-8 and 7-9 show results for simulations with the access threshold set to 50. The

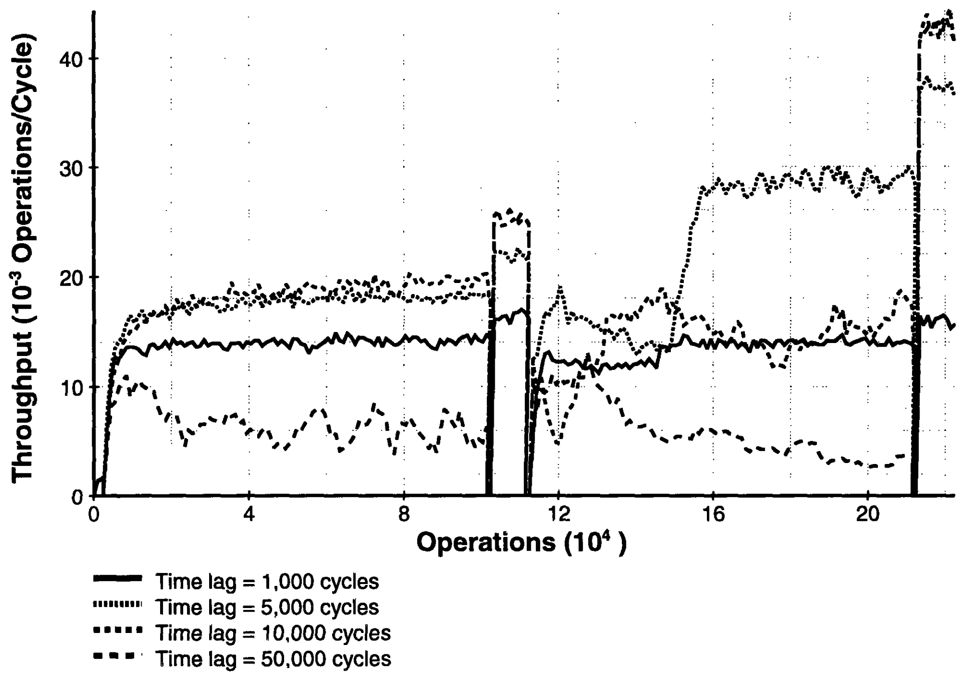


Figure 7-6: Throughput – Access Threshold = 20

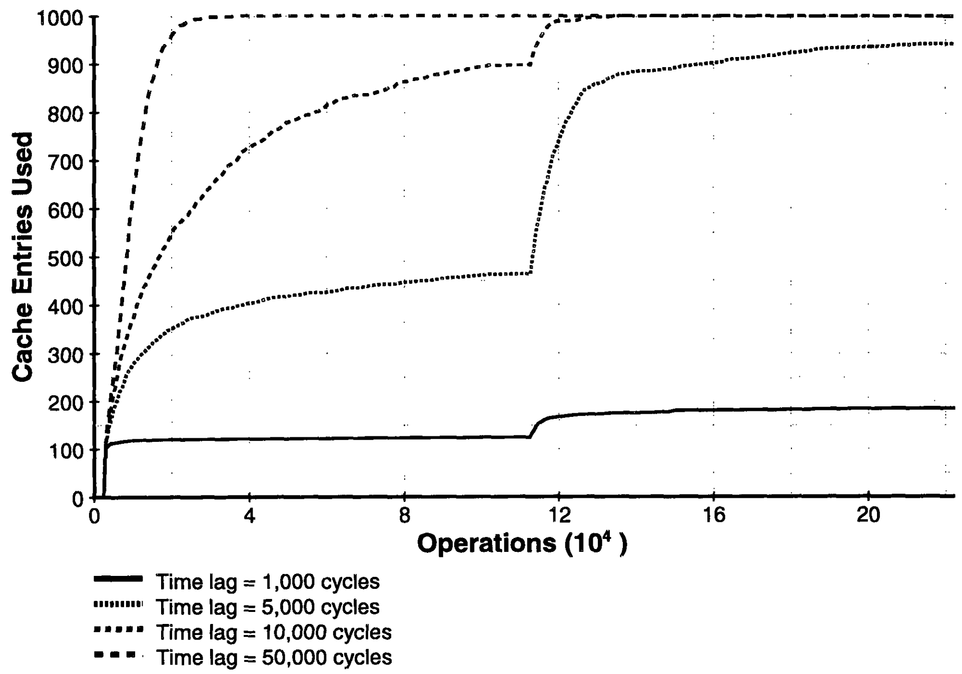


Figure 7-7: Cache Usage – Access Threshold = 20

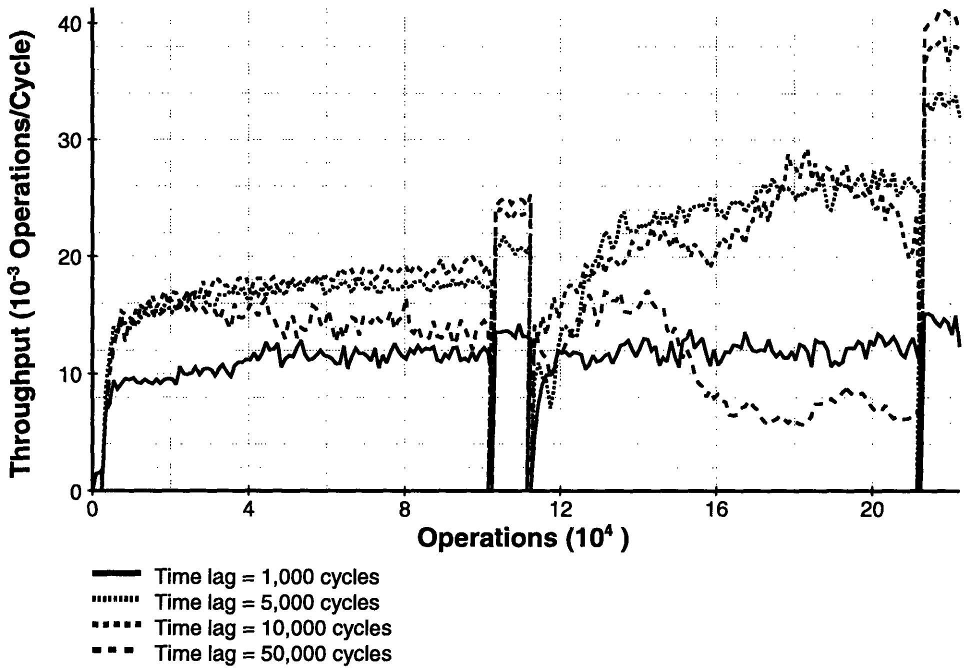


Figure 7-8: Throughput – Access Threshold = 50

messages from this set of simulations are consistent with prior simulations: a low time lag limits throughput by not using enough of the cache and a high time lag allows the cache to fill too rapidly, creating excessive caching overhead. For values in between, the cache is used productively, but without excessive overhead.

Thus far we have been comparing performance for varying time lags, given a fixed access threshold. In figure 7-10 we show the results for a series of simulations where the time lag is 10,000 and the access threshold is varied. (We use 10,000 because in the simulations presented so far it has consistently filled, or come close to filling the available cache space.) In figure 7-10, for the uniform access pattern there is no strongly noticeable difference between the different thresholds, although the lowest threshold, 5, does produce somewhat lower throughput.

When access is limited to 10% of the search space, however, it is very noticeable that performance increases with increasing access threshold. It is not surprising that this should be the case. When the system reaches "steady state", a longer access count should lead to fewer requests for additional copies and lower caching overhead. As we might expect from the results of previous simulations, figure 7-11 shows that for the threshold values that produce the best

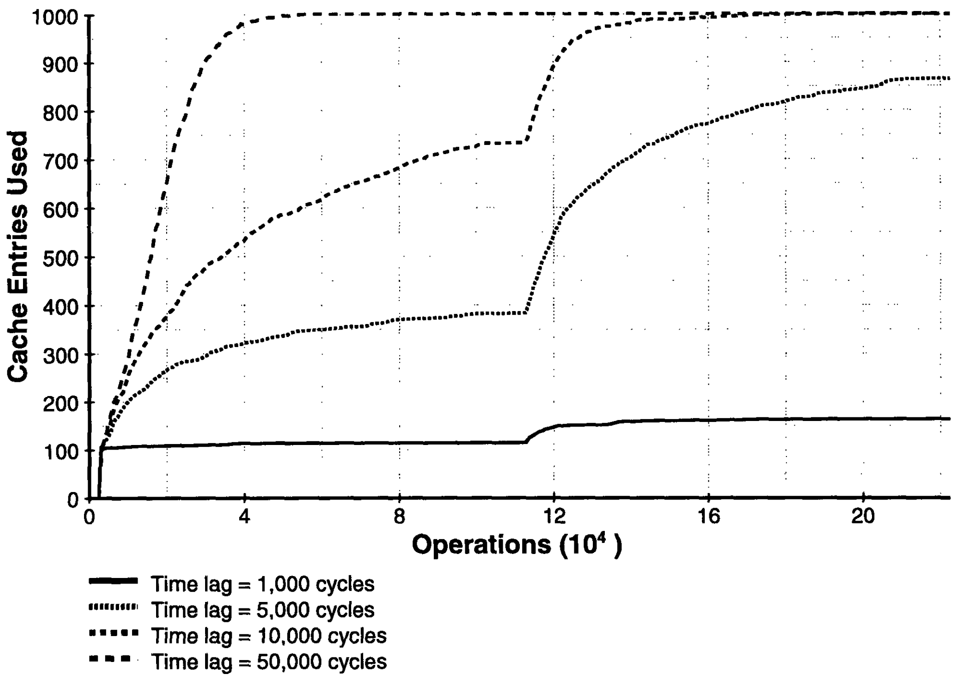


Figure 7-9: Cache Usage – Access Threshold = 50

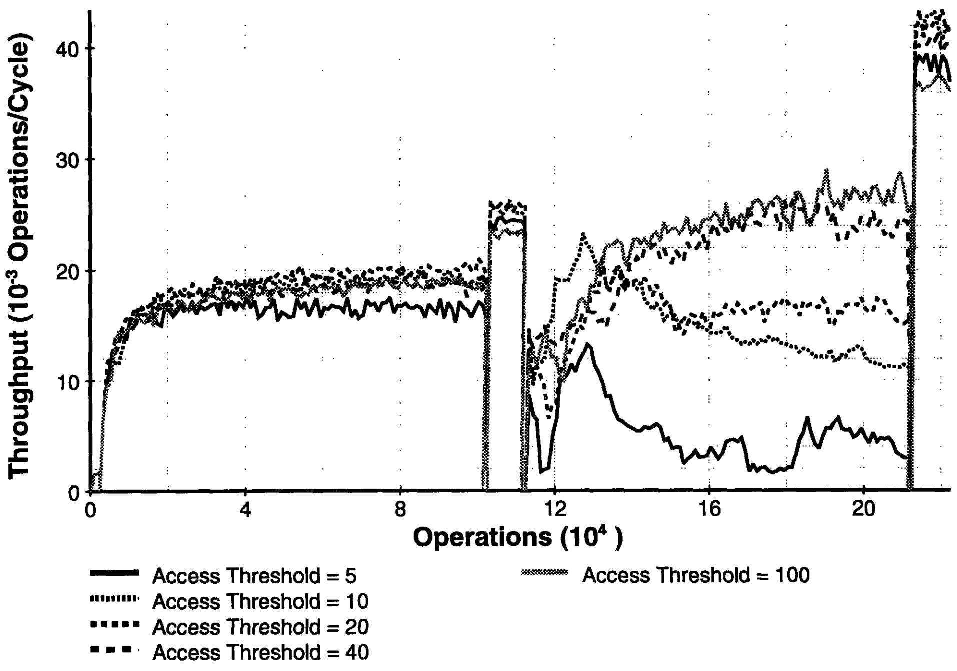


Figure 7-10: Throughput – Time lag = 10,000

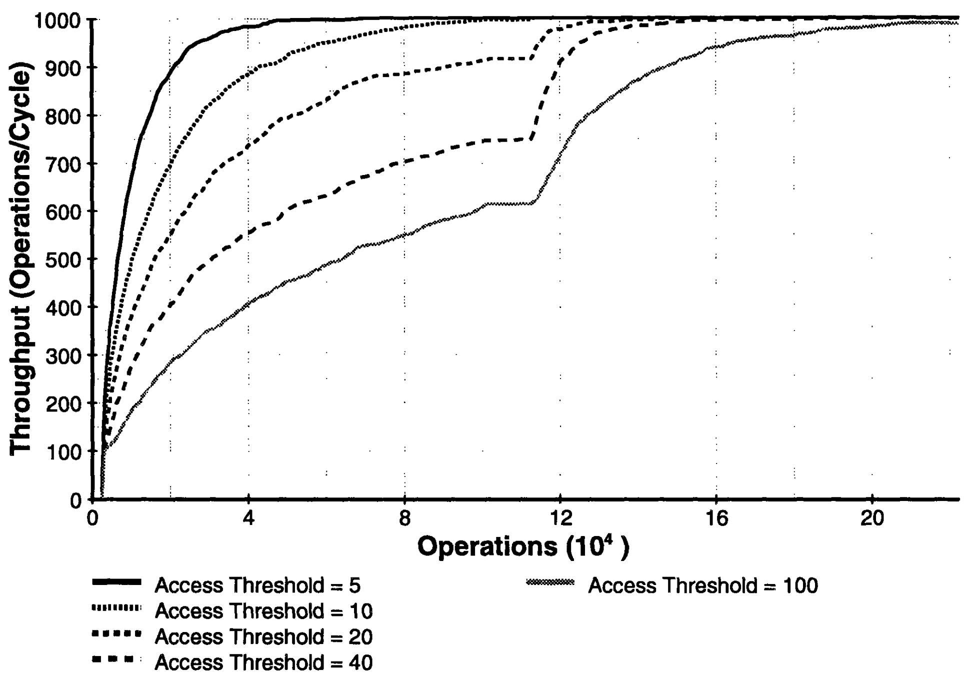


Figure 7-11: Cache Usage – Time lag = 10,000

throughput, the cache is filling more gradually than the lower throughput cases and perhaps not completely.

Thus far we have compared observed throughput with expected throughput and looked at total replication, but not looked in detail at how replication is actually being used. In figure 7-12 we show the use of cache space, by tree level, for time lag of 10,000 and access count of 10. The base tree for this simulation has, from the root down to the leaves, 1, 7, 45, and 336 nodes per level. Counting original nodes and copies, at the end of the 100,000 uniformly distributed accesses, there are 100, 384, 400, and 502 nodes per level (again, from the root down the leaves). Leaving out the root, which is fully replicated, we see that the second and third levels have roughly the same number of nodes, as we had hoped, but that the leaf level has more than desired. Most of the 502 total nodes at the leaves, however, are originals. When access is limited to 10% of the search space, we estimate we will be using a subset of the tree consisting of, from root to leaves, 1, 1, 5 and 34 nodes. At the end of the 100,000 accesses, there are 100, 102, 416, and 423 nodes per level. This time the nodes used in the top two levels are fully replicated, and the lower two levels have very nearly the same number of nodes, as

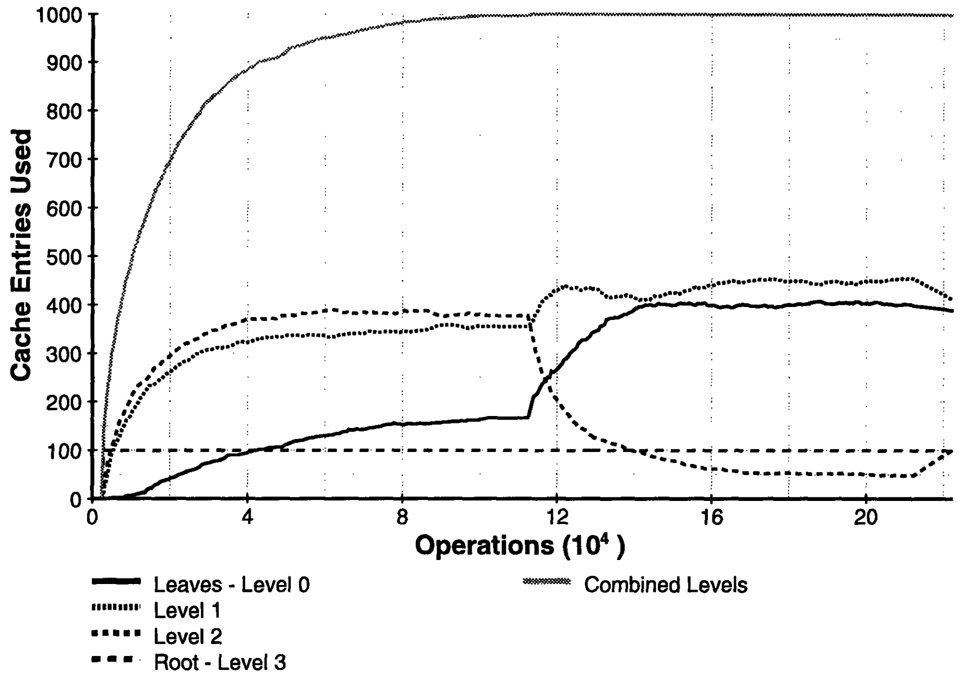


Figure 7-12: Cache Usage By Level – Time lag = 10,000, Access Threshold = 10

hoped.

Figure 7-12 shows an interesting phenomenon during the second set of 100,000 accesses – the number of copies at level two of the tree drops well below 100 for most of the time, but rises back up to 100 as the simulation completes. This is caused by a re-mapping bottleneck on the processor holding the one node at that level that is to be replicated. One reason why there is so much greater variation in performance when access is limited to 10% of the search range is that there is only one B-tree node from the level below the root that is being used (out of the total seven nodes we expect at that level). That single node must distribute all re-mappings caused by changes in the level below it. The nodes used at the level below it will be used much more frequently than under uniform access, so they will be generating a higher rate of cache changes. When the access pattern is restricted like this, that second level node becomes a "pseudo-root" and we would like to avoid having the processor holding the "master" of that node involved in every re-mapping of its children, just as we avoided having the root involved in every re-mapping of its children. In the next section we present initial results from an update to our initial re-mapping algorithm that addresses this problem.

## 7.4 Dynamic Algorithm – Improved Re-mapping

The re-mapping algorithm used in the previous section assumed that each B-tree node does not know all the locations of its parent and children. As a result, the parent must be involved in processing all re-mappings when the replication of a node changes. In this section we explore the potential benefits from allowing the master copy of a node to know (within the limits of this knowledge being kept up to date) the location of all copies of its parent.

In this modification, knowledge of parent locations is kept up to date by sending a copy of the location map to the master copy of each child when the replication of a tree node changes. If each node has this information about its parent, when the replication of a tree node changes the master copy of the node can directly perform the re-mapping of parent copies to its own copies, without involving the master copy of the parent. We also made one additional change – rather than telling each copy of the parent about only one copy to which it can forward descending tree operations, we send each copy the full location map of its children and allow a random selection from the full set of copies each time an operation is forwarded.

The results for a time lag of 10,000 and several different values of the threshold are shown in figure 7-13. Performance for the uniform access portion of the simulation is very similar to, but slightly lower than that of our initial model. There is slightly more overhead in sending location maps and making forwarding decisions, and this updated algorithm also must send a message to the master copy of each child.

When access is limited to 10% of the search space, the updated algorithm exhibits better performance for all values of the access threshold. For the cases with large values for the access threshold, the throughput shows a similarly shaped curve, but with consistently higher throughput. For the simulations with lower access threshold, throughput no longer tails off as the simulation progresses. With the elimination of the re-mapping bottleneck at the "pseudo-root", throughput is significantly higher and can continue to grow as the cache contents are adjusted.

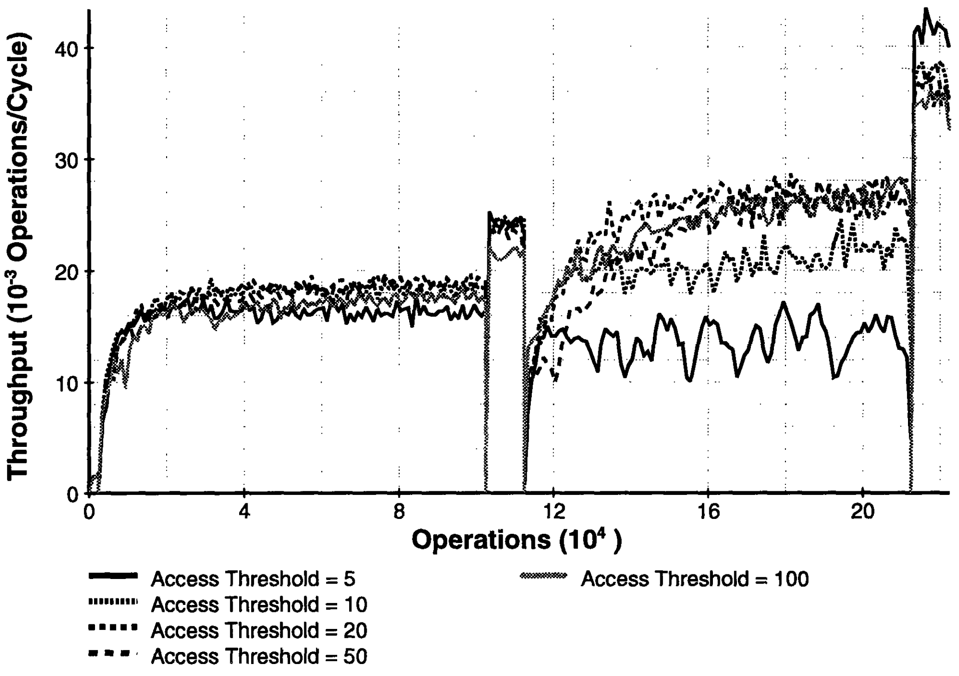


Figure 7-13: Throughput, Improved Re-mapping – Time lag = 10,000

## 7.5 Future Directions

In algorithms of the type presented in this chapter, when the cache reaches "steady state", overhead does not drop to zero. Instead, nodes are added and removed from caches with no significant net change in the use of replication, merely a shuffling of the cache contents. We have begun to explore "centralized" control of replication to reduce this steady-state overhead It is based on the distributed capture of access counts at each copy of a node, but replication change decisions are made centrally by the master copy of a node.

For much of the time this new algorithm is active, the only overhead is the accumulation of access counts. When it is time to review and possibly change replication (determined by a time interval or a number of accesses to a tree node) rebalancing of the B-tree is started at the root node. The root node polls each of its copies for their local access count, which is then reset to zero. The sum of the counts indicates the number of operations that have passed through the root node since the last rebalance and serves as the measure for 100% relative frequency of access.

As in the algorithm tested earlier, the root would generally be kept fully replicated. When any necessary changes in the replication of the root are completed, the new location map of the root and the count of the total number of operations is passed to each of its children. Each child begins a similar process to that performed at the root. It first polls its copies for their access counts and sums the results. The ratio of that sum to the total operations through the system gives the relative frequency of access to the tree node. Relative frequency of access is translated into the desired number of copies using curves such as those developed in chapter 6. If more copies are desired than currently exist, additional copies are sent to randomly selected processors not currently holding copies. If fewer copies are desired than currently exist, some processors are instructed to remove their copies. When these replication adjustments have been made, the node then remaps the copies of its parent to its own copies. Finally, it forwards its new location map and the total operation count to its own children.

While this algorithm can introduce a potentially heavy burden while it rebalances, between rebalancings it has virtually no overhead. Further, if there is little or no need for change during a rebalancing, overhead remains quite low. This algorithm would be weakest when the pattern of access changes quickly and dramatically.

## 7.6 Summary

In this chapter we have taken the results of prior chapters that indicated how replication could be optimally used given a static access pattern, and successfully applied those results using a dynamic replication control algorithm. We introduced a simple algorithm for dynamic control of B-tree replication in response to observed access patterns. Through simulation we showed that it does respond to observed access patterns and that it produces a replicated B-tree that, with the overhead of dynamic cache management turned off, matches the throughput produced by the best of our static replication algorithms. When dynamic cache management is active, of course, the overhead of management does reduce the throughput. We also introduced an update to this simple algorithm to eliminate potential bottlenecks and demonstrated that the update had a noticeably beneficial effect.

# Chapter 8

# Conclusions

Our objective in starting the work described in this thesis was to investigate two hypotheses:

1. Static Performance: Given a network, a B-Tree and a static distribution of search keys, it is possible to predict the performance provided by a static replication strategy.

2. Dynamic Balancing: Under certain changing load patterns, it is possible to apply the knowledge of static performance and change dynamically the replication of B-Tree nodes to increase overall performance.

In this work we have shown both of these hypotheses to be true. In doing so we have expanded on prior knowledge and assumptions on how replication can best be used with distributed B-trees.

In investigating the first hypothesis, we demonstrated and described through modeling and simulation, the trade off between replication and performance in a distributed B-tree. Earlier work had used heuristics to select a single point for the appropriate amount of replication to use. We developed insights into the optimal relationship between relative frequency of access to a node and the number of copies to make of a node. While prior work assumed that replication should be proportional to relative frequency of access, we showed that the optimal relationship appears to be a slight variation of that - more copies should be made of frequently used nodes and fewer copies made of less frequently accessed nodes. We also showed that B-trees built using the prior heuristics, or any static placement algorithm, provided good performance (as measured by throughput) only when the pattern of access is fairly uniform. Finally, we showed that, particularly for large B-trees, the prior heuristic approaches can use far more space than appears appropriate for the additional increase in performance.

We used the results from our analysis of static algorithms to direct our investigation of our second hypothesis on dynamic replication control. We introduced a simple algorithm for dynamic control of processor caches and demonstrated that dynamic replication control for B-trees is practical. This initial work presented the continuing challenge of lowering the overhead necessary to support B-tree caching.

The main avenue for future work is in dynamic control of replication. There are two directions future work can proceed. First, algorithms such as the one presented here can be fine tuned and adjusted to reduce overhead. They can also be extended to dynamically adapt the values of the controlling parameters in response to changing operation load. Second, radically different approaches such as the "centralized" balancing algorithm described in section 7.5 can be explored. In both cases the objective is create an algorithm that can react quickly to changes in the access pattern, but present low overhead when the access pattern is stable.

An additional direction for future work extends from our comments in chapter 6 that B-tree performance can be improved by creating a more balanced distribution of nodes and copies than random placement can provide. Future work on any dynamic replication control algorithm, and particularly the "centralized" approach of section 7.5, would benefit from additional work on low cost load balancing techniques.

# Appendix A

# "Ideal" Path-to-Root Space Usage

In chapter 2 we indicated that the "ideal" path-to-root model will use space such that, on average, the number of copies per node $n$ levels above the leaves, for a tree of depth $h$ and branch factor $BF$, distributed across $P$ processors, is:

$$\text{average number of copies} = P * BF^{n-h} + 1 - P/BF^{h}$$

To prove this result we first introduce the symbol $m$ to stand for the number of descendant leaf nodes below an intermediate node, and the symbol $lp$ to stand for the average number of leaf nodes per processor. Given a node with $m$ descendant leaf nodes, our objective is to determine the number of processors that one or more of the $m$ leaves will be found on, and thus the total number of copies that must be made of the intermediate level node.

"Ideal" placement means that there are $lp$ leaf nodes on each processor and that the logically first $lp$ nodes are on the first processor, the logically second $lp$ nodes are on the second processor, and so on. An "ideal" placement of $m$ leaves covers a minimum of $\left\lceil \frac{m}{lp} \right\rceil$ processors. Similarly, it covers a maximum of $\left\lceil \frac{m}{lp} \right\rceil + 1$ processors.

We call an *alignment* the pattern of distribution of $m$ nodes across processors, defined by the number of nodes placed on the first processor in sequence. For example, if 7 nodes are placed on processors with 4 nodes per processor, there are 4 distinct patterns possible,

- 4 nodes on the first processor in sequence, 3 on the next processor;
- 3 on the first processor, 4 on the next processor;

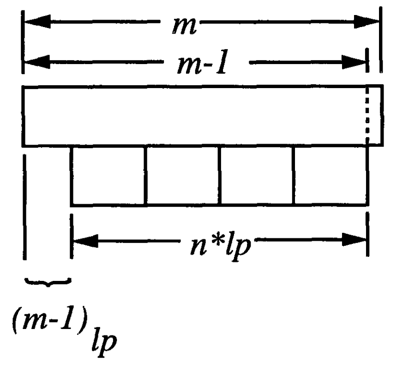


Figure A-1: Alignments Covering Maximum Processors

- 2 on the first processor, 4 on the next processor, 1 on the next after that;
- 1 on the first processor, 4 on the next processor, 2 on the next after that.

There are always $lp$ possible alignments, then the cycle repeats. The maximum number of processors is covered for $(m-1)_{lp}$ of the alignments, where $n_{lp}$ means $n$ modulo $lp$. When an alignment has only one leaf node on the right-most processor it is covering, it will be covering the maximum number of processors. (The only exception is if $(m-1)_{lp} = 0$, in which case all alignments cover the minimum number of processors.) As the alignment is shifted right, there would be $(m-2)_{lp}$ additional alignments covering the maximum number of processors. (See figure A-1). The minimum number of processors is covered by the rest of the alignments, or $lp - (m-1)_{lp}$ of the alignments.

Combining these pieces produces:

$$\text{average number of copies} = \frac{\left\lceil \frac{m}{lp} \right\rceil * (lp - (m-1)_{lp}) + (\left\lceil \frac{m}{lp} \right\rceil + 1) * (m-1)_{lp}}{lp}$$

or

$$\text{average number of copies} = \frac{\left\lceil \frac{m}{lp} \right\rceil * lp + (m-1)_{lp}}{lp}$$

We evaluate this for two cases. First, when $m_{lp} = 0$ (and $m > 0$), $\left\lceil \frac{m}{lp} \right\rceil * lp = m$ and $(m-1)_{lp} = lp - 1$, the sum being $m + lp - 1$. Second, when $m_{lp} \neq 0$, $\left\lceil \frac{m}{lp} \right\rceil * lp = m + lp - m_{lp}$ and $(m-1)_{lp} = (m-1)_{lp}$, the sum again being $m + lp - 1$.

This yields:

$$\text{average number of copies} = \frac{m + lp - 1}{lp}$$

For a tree of depth $h$, with branch factor $BF$, on $P$ processors, the average number of leaf nodes per processor is $BF^h/P$. The number of descendant children for a node $n$ levels above the leaves is $BF^n$, thus:

$$\text{average number of copies} = \frac{BF^n + BF^h/P - 1}{BF^h/P}$$

or

$$\text{average number of copies} = P * BF^{n-h} + 1 - P/BF^h$$

# Appendix B

# Queueing Theory Notation

The following notation is used in the queueing theory model of chapter 4:

| | | |
|---|---|---|
| $K$ | = | Number of service centers in the system. |
| $C$ | = | Number of task classes in the system. |
| $N$ | = | Number of tasks in the system. |
| $N_c$ | = | Number of tasks of class $c$ in the system. |
| $\overline{N}$ | = | Population vector = $(N_1, ..., N_C)$. |
| $X(N)$ | = | Throughput given $N$ tasks. |
| $X_c(\overline{N})$ | = | Throughput for class $c$ given $\overline{N}$ tasks. |
| $S_k(N)$ | = | Mean visit service requirement per task for service center $k$. |
| $S_{c,k}(\overline{N})$ | = | Mean visit service requirement per task of class $c$ for service center $k$. |
| $V_k(N)$ | = | Mean visit count per task for server $k$. |
| $V_{c,k}(\overline{N})$ | = | Mean visit count per task of class $c$ at service center $k$. |
| $D_k(N)$ | = | Service demand at service center $k$. $D_k(N) \equiv V_k(N)S_k(N)$ |
| $D_{c,k}(\overline{N})$ | = | Service demand of class $c$ at service center $k$. $D_{c,k}(N) \equiv V_{c,k}(\overline{N})S_{c,k}(\overline{N})$ |
| $Q_k(N)$ | = | Mean queue length at service center $k$. |
| $Q_{c,k}(\overline{N})$ | = | Mean queue length of tasks of class $c$ at service center $k$. |
| $R_k(N)$ | = | Total residence time for a task at server $k$ when there are $N$ tasks in the system. |
| $R_{c,k}(\overline{N})$ | = | Total residence time for a task of class $c$ at server $k$ when there are $N$ tasks in the system. |
| $U_{c,k}(\overline{N})$ | = | Mean utilization of server $k$ by tasks of class $c$. |
| $\overline{1_c}$ | = | $C$-dimensional vector whose $c$-th element is one and whose other elements are zero. |

# Bibliography

[ACJ+91] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, Ben-Hong Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991.

[Bar79] Y. Bard. Some Extensions to Multiclass Queueing Network Analysis. In M. Arato, A. Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems*, pages 51–62. North-Holland Publishing Co., 1979.

[Bar80] Y. Bard. A Model of Shared DASD and Multipathing. *Communications of the ACM*, 23(10):564–572, October 1980.

[BDCW91] E. Brewer, C. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: a High-performance Parallel Architecture Simulator. Technical Report TR-516, MIT, 1991.

[BM72] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1(9):173–189, 1972.

[Bre92] E. Brewer. Aspects of a Parallel-Architecture Simulator. Technical Report TR-527, MIT, 1992.

[BS77] R. Bayer and M. Schkolnick. Concurrency of Operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[CBDW91] A. Colbrook, E. Brewer, C. Dellorocas, and W. E. Weihl. Algorithms for Search Trees on Message-Passing Architectures. Technical Report TR-517, MIT, 1991. Related paper appears in Proceedings of the 1991 International Conference on Parallel Processing.

[CN82] K. M. Chandy and D. Neuse. Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems. *Communications of the ACM*, 25(2):126–134, February 1982.

[Com79] D. Comer. The Ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[Cor69] F. J. Corbató. A Paging Experiment with the MULTICS System. In H. Feshbach and K. Ingard, editors, *In Honor of Philip M. Morse*, pages 217–228. M.I.T. Press, 1969.

[Cox62] D. Cox. *Renewal Theory.* Wiley, 1962.

[CS90] A. Colbrook and C. Smythe. Efficient Implementations of Search Trees on Parallel Distributed Memory Architectures. *IEE Proceedings Part E*, 137:394–400, 1990.

[CT84] M. Carey and C. Thompson. An Efficient Implementation of Search Trees on lg N + 1 Processors. *IEEE Transactions on Computers*, C-33(11):1038–1041, 1984.

[Dal90] W. Dally. Network and Processor Architecture for Message-Driven Computers. In R. Suaya and G. Birtwistle, editors, *VLSI and Parallel Computation*, pages 140–218. Morgan Kaufmann Publishers, Inc., 1990.

[Del91] C. Dellarocas. A High-Performance Retargetable Simulator for Parallel Architectures. Technical Report TR-505, MIT, 1991.

[dSeSM89] E. de Souza e Silva and R. Muntz. Queueing Networks: Solutions and Applications. Technical Report CSD-890052, UCLA, 1989.

[HBDW91] W. Hsieh, E. Brewer, C. Dellarocas, and C. Waldspurger. Core Runtime System Design – PSG Design Note #5. 1991.

[HL84] P. Heidelberger and S.S Lavenberg. Computer Performance Evaluation Methodology. Research Report RC 10493, IBM, 1984.

[JC92] T. Johnson and A. Colbrook. A Distributed Data-balanced Dictionary Based on the B-link Tree. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 319–324. IEEE, 1992.

[JK93] T. Johnson and P. Krishna. Lazy Updates for Distributed Search Structure. In *Proceedings of the International Conference on Management of Data*, pages 337–400. ACM, 1993. (ACM SIGMOD Record, Vol. 20, Number 2).

[JS89] T. Johnson and D. Shasha. Utilization of of B-trees with Inserts, Deletes, and Modifies. In *ACM SIGACT/SIGMOD/SIGART Symposium on Principles of Database Systems*, pages 235–246. ACM, 1989.

[JS90] T. Johnson and D. Shasha. A Framework for the Performance Analysis of Concurrent B-tree Algorithms. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 273–287. ACM, 1990.

[Kle75] L. Kleinrock. *Queueing Systems, Volume 1: Theory.* Wiley Interscience, 1975.

[Kru83] C. P. Kruskal. Searching, Merging, and Sorting in Parallel Computation. *IEEE Transactions on Computers*, C-32(10):942–946, 1983.

[KW82] Y. Kwong and D. Wood. A New Method for Concurrency in B-trees. *IEEE Transactions on Software Engineering*, SE-8(3):211–222, May 1982.

[LS86] V. Lanin and D. Shasha. A Symmetric Concurrent B-tree Algorithm. In *1986 Fall Joint Computer Conference*, pages 380–389, 1986.

[LY81] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Transactions on Computer Systems*, 6(4):650–670, 1981.

[LZGS84] E. Lazowska, J. Zahorjan, G. S. Graham, and K. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Inc., 1984.

[MR85] Y. Mond and Y. Raz. Concurrency Control in $B^+$-Trees Databases Using Preparatory Operations. In *11th International Conference on Very Large Databases*, pages 331–334. Stockholm, August 1985.

[PS85] J. Peterson and A. Silberschatz. *Operating Systems Concepts.* Addison-Wesley Publishing Co., 1985.

[Rei79a] M. Reiser. A Queueing Network Analysis of Computer Communication Networks with Window Flow Control. *IEEE Transactions on Communications*, C-27(8):1199–1209, 1979.

[Rei79b] M. Reiser. Mean Value Analysis of Queueing Networks, A New Look at an Old Problem. In M. Arato, A. Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems*, pages 63–. North-Holland Publishing Co., 1979. Also IBM RC 7228.

[RL80] M. Reiser and S. S. Lavenberg. Mean-Value Analysis of Closed Multichain Queuing Networks. *Journal of the ACM*, 27(2):313–322, April 1980.

[Sag85] Y. Sagiv. Concurrent Operations on $B^*$-Trees with Overtaking. In *Fourth Annual ACM SIGACT/SIGMOD Symposium on the Principles of Database Systems*, pages 28–37. ACM, 1985.

[SC91] V. Srinivasan and M. Carey. Performance of B-tree Concurrency Control Algorithms. In *Proceedings of the International Conference on Management of Data*, pages 416–425. ACM, 1991. (ACM SIGMOD Record, Vol. 20, Number 2).

[SM81] K. C. Sevcik and I. Mitrani. The Distribution of Queueing Network States at Input and Output Instants. *Journal of the ACM*, 28(2):358–371, April 1981.

[Wan91] P. Wang. An In-Depth Analysis of Concurrent B-tree Algorithms. Technical Report TR-496, MIT, 1991. Related paper appears in Proceedings of the IEEE Symposium on Parallel and Distributed Processing, 1990.

[WBC$^+$91] W. E. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldsburger, and P. Wang. Prelude: A System for Portable Parallel Software. Technical Report TR-519, MIT, 1991.

[WW90] W. E. Weihl and P. Wang. Multi-Version Memory: Software Cache Management for Concurrent B-trees. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 650–655. IEEE, December 1990.