286[illegible]7

This is to certify that the

thesis entitled

A Linear Hashed Main Memory Database in a
Non-Uniform Memory
Multi-Processor System

presented by

Charles R. Severance

has been accepted towards fulfillment
of the requirements for

Masters degree in Computer Science

[illegible]

Major professor
Dr. Sakti Pramanik

Date 02/19/90

O-7639 *MSU is an Affirmative Action/Equal Opportunity Institution*

A LINEAR HASHED MAIN MEMORY DATABASE IN A NON-UNIFORM MEMORY MULTI-PROCESSOR SYSTEM

By

Charles R. Severance

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

1990

6042673

ABSTRACT

A LINEAR HASHED MAIN MEMORY DATABASE IN A
NON-UNIFORM MEMORY
MULTI-PROCESSOR SYSTEM

By

Charles R. Severance

Data structures and algorithms are developed to implement a high performance hashed main memory database on a multi-processor system with non-uniform memory access (NUMA). Memory performance limitations of NUMA systems are characterized. The memory performance of the BBN GP-1000 system is measured and reported. High performance database control and database search structures are developed to minimize the impact of the memory performance limitations of NUMA systems. The database performance is analyzed on BBN GP-1000. Different designs for database control variables and the performance of the designs are compared.

Dedicated to my wife Teresa, daughter Amanda, and parents Russell and Marcia for all of their support.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Sakti Pramanik for his guidance in this research. I would also like to thank my committee members Dr. Carl Page and Dr. Anthony Wojcik for their encouragement.

I would like to thank the MSU Computer Lab for supporting this effort in many important ways. I would like to thank Dr. Lewis Greenberg and Richard Moore for their support and encouragement.

TABLE OF CONTENTS

List of Figures.......viii

1.0 Introduction.......1

1.1 Previous Work.......2

1.2 Linear Hashing.......3

1.3 Description of Concurrent Linear Hashing.......10

1.4 Motivation.......14

2.0 Memory Performance of a NUMA System.......17

2.1 Experiment Details.......20

2.2 Overall Memory Performance of the GP-1000.......21

3.0 Distributed Linear Hashing.......25

3.1 Retry Logic.......27

3.2 Multi-threaded Reorganization.......28

3.3 Distributed Index Computation.......30

3.4 Maintaining M.......31

3.5 Maintaining P.......31

4.0 Performance Analysis.......34

4.1 Experiment Details.......34

4.2 Performance Impact of Distributed Control Variables.......35

4.3 Performance of the Overall System.......39

4.4 Performance of Database Reorganization 40

5.0 Conclusion 42

Appendix A: KDL-RAMFILE System Documentation 44

Appendix B: Pseudocode for the KDL Ramfile System.......................................49

Bibliography 55

LIST OF FIGURES

Figure 1.1 Simple hashed database with overflow buckets 4

Figure 1.2 Example linear hashed database..7

Figure 1.3 Example linear hashed database after split 8

Figure 1.4 Linear hashed database with overflow bucket.......................................8

Figure 1.5 Linear hashed database with overflow bucket removed........................9

Figure 1.6 Concurrent linear hashed database 10

Figure 1.7 Bucket lock compatibility 11

Figure 1.8 Lock requirements for database transactions 11

Figure 1.9 Inserting a record into an incorrect bucket..12

Figure 1.10 Rehashing loop 13

Figure 1.11 Inserting a record correctly with rehashing.......................................14

Figure 2.1 Layout of a GP-1000 node 18

Figure 2.2 Logical layout of the GP-1000 switch...19

Figure 2.3 An individual crossbar in the switch network......................................20

Figure 2.4 Memory test parameters 21

Figure 2.5 Effect of memory location on bandwidth..22

Figure 2.6 Effect of memory location on bandwidth (enlarged scale) 23

Figure 2.7 Effect of transfer style on overall bandwidth 24

Figure 3.1 Main memory data structures 26

Figure 3.2 Pseudocode for finding the proper bucket 28

Figure 3.3 Distributed linear hashed database 29

Figure 3.4 Comparison of techniques for database control variables 30

Figure 3.5 Data structure for distributed M 32

Figure 3.6 Maintaining local copies of P 33

Figure 4.1 Experiment details 34

Figure 4.2 Read performance of different implementations 36

Figure 4.3 Insert performance 37

Figure 4.4 Retry percentage for various control strategies 39

Figure 4.5 Comparison of inserts, deletes and reads 40

Figure 4.6 Chain length control using multi-threaded reorganization 41

Figure A.1 Configurable parameters 47

CHAPTER 1

INTRODUCTION

1. Introduction

Computer systems with many parallel processors such as the BBN Butterfly[BBN86] and the NCUBE[Ncu85] have become generally available in recent years. These systems have been used for a number of different applications. One application which may be well suited for implementation on these multi-processing systems is a high performance main memory database.

These systems have a number of advantages which can be used when implementing a main memory database. With a large number of processors, these systems can have large amounts of real storage at relatively low cost. The aggregate transfer rate from memory to the processors is very high. When multiple processors are used effectively, the aggregate performance of a multi-processor system is very high.

Though the large memory size and aggregate performance are useful when implementing a main memory database, certain architectural constraints make an efficient implementation difficult. Although the memory is large, the access is non-uniform. A processor is able to access a subset of the system memory at a very high rate of speed while accessing the remaining memory is generally much slower. Databases need central control information which is used to find all of the records. On a non-uniform memory access (NUMA) architecture system, these central data structures may cause memory access contention (or a memory "hot spot") to occur when these values are referenced during every database operation. In this paper we present a design which minimizes the effect of these architectural constraints while exploiting the NUMA architecture to enhance performance.

1.1. Previous Work

Linear hashing as a search technique for databases was developed by W. Litwin [Lit80]. The objective of linear hashing is to allow dynamic reorganization of a hashed database as records are inserted or deleted while maintaining the ability to access records in a single bucket access. Litwin developed linear hashing for a disk based system running with a single processor.

A standard hash function is used to "scatter" the records of a database based on the key information in the record. Space will not be utilized effectively if the hash function scatters the records too much. If the records are not scattered enough there will be collisions which must be resolved using record chains or overflow buckets. These chains or overflow buckets will degrade performance as additional bucket accesses will be required.

Linear hashing allows the hash function to change gradually while the database is being modified. Each change to the hash function only affects a small portion of the database.

A solution for concurrent linear hashing was proposed by C. S. Ellis [Ellis87]. Concurrent linear hashing adds a locking protocol and extends the data structures allowing concurrent access to the entire linear hashed file by multiple processors or multiple processes on a single processor. Like linear hashing, concurrent linear hashing was intended for a disk environment.

As commercially available large scale parallel processors have become available, research into the use of these systems to implement database systems has started. The work in [BaFr86] involves implementing a relational database on the Hypercube architecture. The BBN Butterfly system was used to implement relational database operations in [RoJa87]. The use of various search structures for database queries in main memory was studied in [LeC86]. The work in [PrDa88] proposes the use of a multi-directory scheme to minimize the number of key comparisons to find a record in a main memory database while maintaining storage utilization.

The work in [KiSn83] explores a memory sharing network topology for NUMA systems and predicts the performance of the networks. The performance of the Butterfly memory sharing network is measured in the presence of hot spots in [Thom86]. A general evaluation of the memory bandwidth on the Butterfly memory system was done in [Sev89].

1.2. Linear Hashing

In a simple hashed database, records are distributed into M buckets using a fixed hash function as follows:

$$bucket = record\_key \ mod \ M;$$

Where record_key is an integer key value created by applying a hash function to the actual key information in the original record. Beginning with an empty database, records are inserted into the buckets based on the value for M. If a bucket has no space for a record to be inserted it is called a collision. When a collision occurs, the record is placed into an overflow bucket. When records are placed into the overflow buckets, two bucket accesses are required to retrieve the record. In Figure 1.1 bucket B_2 has overflowed requiring an overflow bucket.

Bucket chains can be of arbitrary length as the database becomes overloaded. As these chains grow, the performance of record searches begins to suffer because records are no longer accessed in a single bucket access. In a disk based system the number of

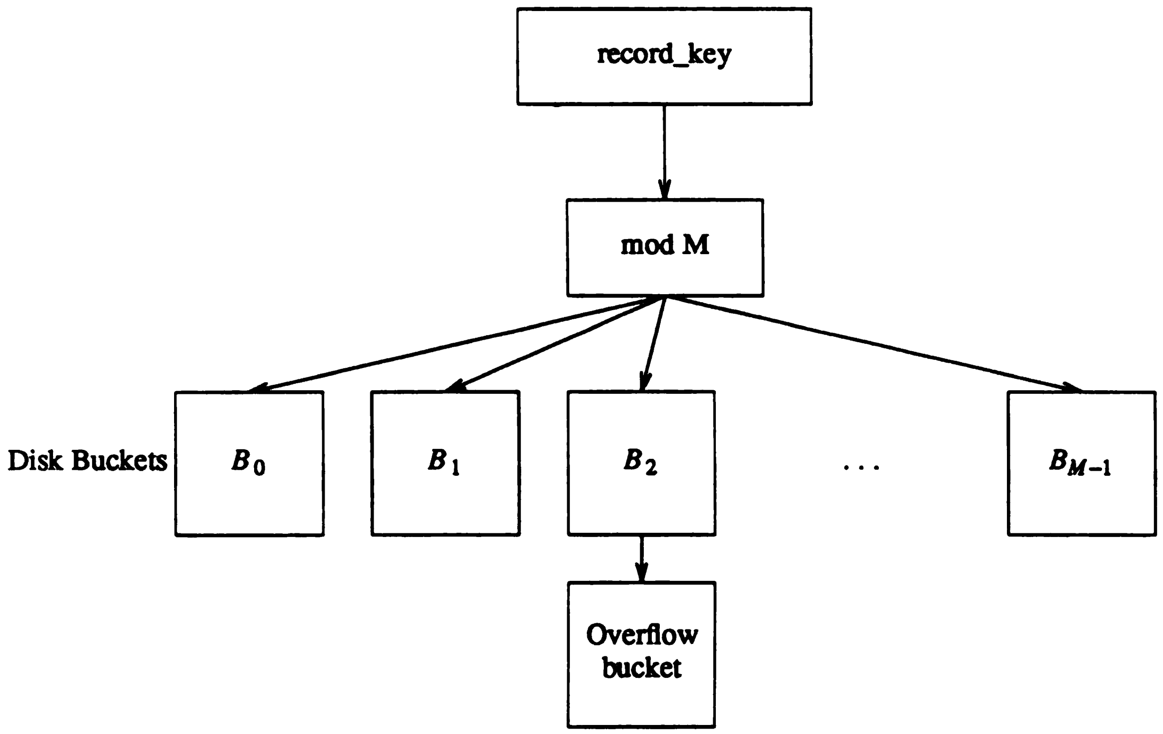


Figure 1.1 Simple hashed database with overflow buckets

bucket accesses are the primary performance limitation.

In order to solve the problem of overflow buckets, the database must be reorganized and the hashing function must be modified. A simple approach would be to increase the value for M and make another copy of the information using the increased value for M as the hash function for the new database. The disadvantage of this technique is that the entire database is inaccessible during this reorganization.

Linear hashing solves this problem by allowing the database size to grow dynamically. Instead of using M buckets until they all overflow and then reorganizing the database all at the same time, Linear hashing adds buckets one at a time in a linear fashion beginning at bucket B_M.

As buckets in the database overflow, buckets starting at bucket B_0 are "split". To split bucket B_i, bucket B_{i+M} is created. Then the records in bucket B_i are scanned and re-hashed using the hash function:

$$newbucket = record\_key \ mod \ (2*M)$$

Since the records being scanned are all in bucket B_i, They satisfy the condition:

$$i := record\_key \ mod \ M$$

The new bucket which is computed using the $2*M$ hash value is either bucket B_i or bucket B_{i+M} because:

$k \ mod \ M = i \Rightarrow k = nM+i$ for some integer n

If n is even then

$n = 2r$ for some integer r

This implies

$k = 2rM+i$
$k = r(2M)+i$

So

$k \ mod \ 2M = i$

If n is odd then

$n = 2r+1$ for some integer r

This implies

$k = (2r+1)M+i$
$k = r(2M)+(M+i)$

So

$k \ mod \ 2m = i+m$

Assuming that the record_key values are randomly distributed, approximately half of the records will end up in the B_i bucket and half of the records will end up in the B_{i+M} bucket.

In order to keep track of the splitting and allow the hash function to properly locate the records in the buckets which have been split, a new variable is used. The variable P indicates the next bucket to be split. Blocks less than P have been split and buckets P and above have yet to be split. In order to properly locate the records in split buckets, the hash function uses P to determine if the bucket has been split.

The dynamic hash function using P is as follows:

```
bucket := record_key mod M
if bucket < P
  bucket := record_key mod (2 * M)
```

The initial hash calculation will only compute the correct bucket if the bucket has not yet been split at the current value for M. If the bucket has been split the result of the initial hash calculation will be less than M. Since the records in the split buckets have been hashed using $2*M$ it is necessary to re-compute the proper bucket using $2*M$ to find the proper bucket. The bucket as a result of this second computation may be the same as the bucket computed in the first computation.

As records are inserted the database is re-organized by adding buckets in a linear fashion and P is updated. The hash function adapts to the changing size of the database. As records are deleted from the database and bucket loading falls under some criterion the buckets are merged. Merging buckets "undoes" the effects of a split. The records from buckets B_i and B_{i+M} are rehashed using M and placed into bucket B_i.

Note that although the hash function performs two computations on the hashed key value to find the proper bucket for the key, the algorithm never accesses the wrong bucket. Since the bucket accesses are the major cost in a disk based system, the additional computation has no significant impact on the performance.

The variable P is incremented each time the B_P bucket is split unless P is equal to $M-1$. If P is equal to $M-1$ then P is set to zero and M is doubled. When the B_P and the B_{P+M} bucket are merged, P is decremented unless P is zero. If P is zero, M is halved and P is set to the new value of $M-1$.

With this approach, it is still possible that overflow buckets will be required since only bucket B_P is allowed to split. When a collision occurs in a bucket which is not the B_P bucket an overflow bucket is created and the B_P bucket is split.

Figure 1.2 shows an example database after records containing record_key values of 1 through 11 have been inserted. Each bucket has space for three entries and all the buckets are full.

When a record with a hash_record_key value of twelve is to be inserted, the bucket computation indicates that the record belongs in bucket B_0. Since bucket B_0 already has three records and P is zero, it will be split. Keys 0, 4, 8, and 12 are rehashed using 2*M in the hash computation and distributed between buckets B_0 and B_4. Figure 1.3 shows the database after the split has been completed.

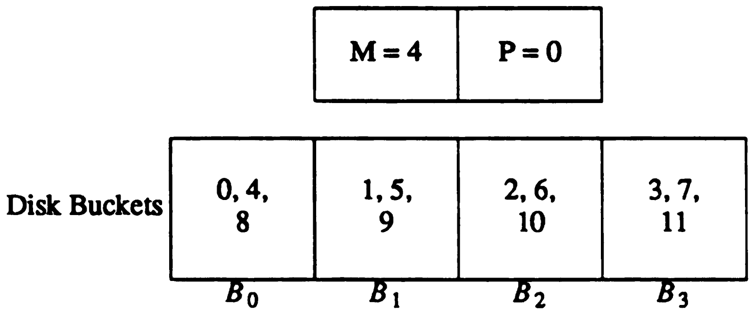


Figure 1.2 Example linear hashed database

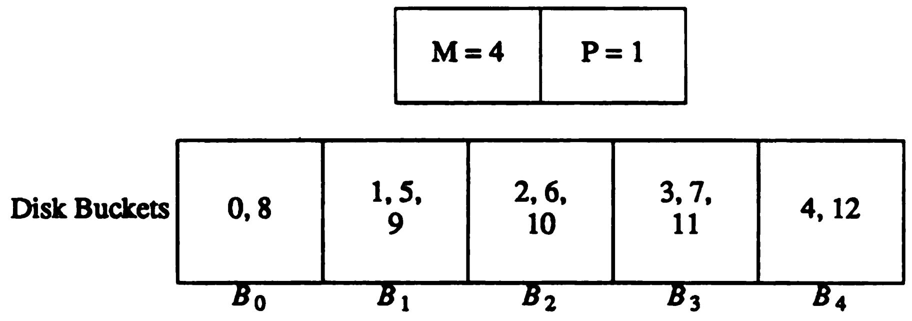


Figure 1.3 Example linear hashed database after split

As a result of the split, an overflow chain has been avoided. The next bucket to be split will be B_1. P has been updated to reflect the next bucket to be split.

In Figure 1.4, a record with a hash_record_key of 14 is inserted into bucket B_2 which cannot be split. An overflow bucket is created for the record and bucket B_1 is split. In Figure 1.5 a record with a hash_record_key of 18 is inserted into B_2. B_2 is now eligible to be split because P has advanced. B_2 is split into B_2 and B_6 removing the overflow bucket.

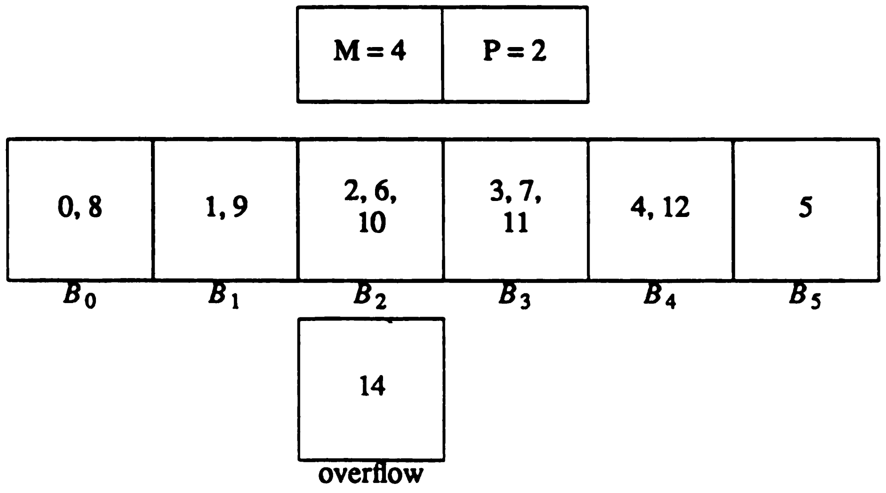


Figure 1.4 Linear hashed database with overflow bucket

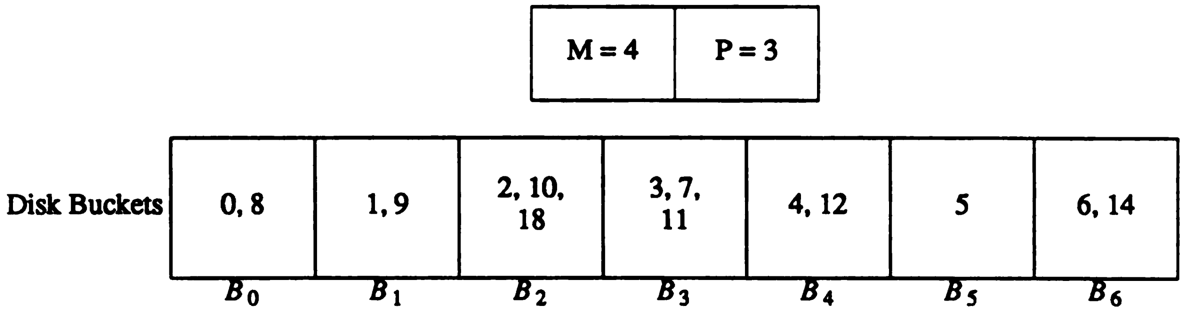


Figure 1.5 Linear hashed database with overflow bucket removed

This shows how the database re-organization can solve the problems of overflow buckets. A hash function will never be perfectly random so overflow buckets will always be necessary. The objective of linear hashing is to access records in as close to a single bucket as possible without wasting a great deal of memory.

According to Litwin, linear hashing can provide access to records with an average number of 1.03 bucket accesses while maintaining a storage utilization of 60%. To accomplish a storage utilization of 90% the average number of bucket accesses per record access rises up to 1.35 [Lit80].

Litwin proposed several algorithms for determining when to split the B_P bucket. The simplest algorithm shown above is called "uncontrolled splitting". Uncontrolled splitting only splits the P bucket when a bucket has completely filled up. "Controlled splitting" will split buckets based on a load factor. For example, a split might be triggered when a record was inserted into a bucket which caused the bucket to be more than 75% full. Another approach [Scholl79] suggests that splitting should be done every fixed number of insertions.

1.3. Description of Concurrent Linear Hashing

The original linear hashing design did not have data structures or locking protocols required to allow concurrent access by multiple processes on the same database.

C. S. Ellis proposed extensions to linear hashing which provide a high degree of concurrency among processes executing find, insert, and delete. [Ellis87]

The primary extensions to the data structures were to add a lock to each bucket and to add a level value to each bucket. M is related to the level value using the following formula:

$$M = Initial\_M * 2^{(level-1)}$$

Level indicates the number of times a particular bucket has been split. Level is initially set to one in all of the buckets. Level is incremented as buckets are split and decremented as buckets are merged. The data structure for concurrent linear hashing is shown in Figure 1.6.

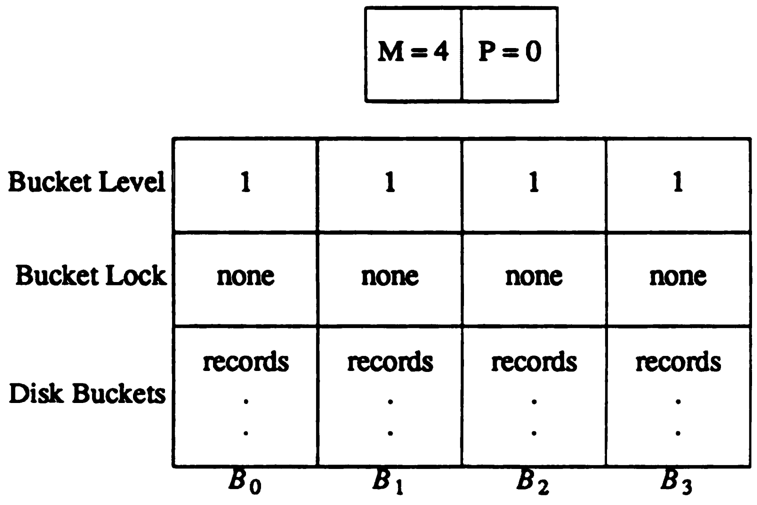


Figure 1.6 Concurrent linear hashed database

When there are overflow buckets, the lock in the first bucket in the chain acts as the lock for the entire chain.

Both the lock on the *M* and *P* variables and the locks in each bucket chain provide read, write, and exclusive locking. The compatibility of locks is shown in Figure 1.7.

Figure 1.8 shows the locks which are required by each type of database operation on both the bucket being affected and the *M* and *P* variables. Concurrency between the find, insert, delete, and split depends on the fact that each database operation makes a separate copy of the bucket from disk. A process could have a read lock on a bucket when performing a find operation and another process could obtain a write lock to perform a delete operation. This works because the first process has a separate memory copy of the bucket which is not affected by the second process.

	Existing Lock		
Lock request	Read	Write	Exclusive
Read Lock	Yes	Yes	No
Write Lock	Yes	No	No
Exclusive Lock	No	No	No

Figure 1.7 Bucket lock compatibility

Database Transaction	Bucket Chain Lock Needed	*M* and *P* Lock Needed
Insert	Write	Read
Delete	Write	Read
Find	Read	Read
Split	Write	Write
Merge	Exclusive	Exclusive

Figure 1.8 Lock requirements for database transactions

Splits and merge operations are single threaded because the split operation requires a write lock on *P* and *M* and the merge operation requires an exclusive lock on these variables. These two locks cannot be held simultaneously so only one process can be splitting *or* merging at one time.

The level value in each bucket is used to recover from a situation where a database reorganization occurs between the time a process reads the values for *P* and *M* and the time a process obtains the lock on the bucket.

Figure 1.9 shows a time sequence where two processors are attempting to insert records into the same bucket at the same time without using the level values in concurrent linear hashing.

The record will be inserted into the wrong bucket at *time*=13 because *P* was changed by processor A at *time*=10 while Processor B was waiting for the lock on the bucket. This is solved by incrementing the level value in the bucket as the split is completed.

Time	M	P	Processor A	Processor B
1	4	1	Insert rec 17	Insert rec 21
2			Read P, M	Read P, M
3			Compute bucket=1	Compute Bucket=1
4			Lock bucket=1	
5				Wait for lock
6			Insert Record	
7			Bucket has overflowed	
8			Split Bucket into	
9			buckets 1 and 5	
10	4	2	P = 2	
11			Unlock bucket=1	
12				Get lock bucket=1
13				Insert record(error)
14				Unlock bucket

Figure 1.9 Inserting a record into an incorrect bucket

When a bucket is locked the initial hash computation must be checked using the level value stored in the bucket. If the second computation indicates the incorrect bucket has been locked the hash computation must be redone. This step is referred to as the "rehashing loop".

The algorithm for the proper locking of a bucket chain is shown in Figure 1.10.

Figure 1.11 shows the time sequence which failed without the level value operating properly.

Record 21 will properly be placed in bucket B_5 because the level in bucket B_1 indicates the record belongs in B_5. It is not possible to solve this problem by holding a lock on *M* and *P* until a bucket lock was obtained because this would result in a deadlock situation. Processor A would be waiting for the lock on the variables to update *P* at *time*=10 while Processor B would be holding the read lock on the variables waiting for the bucket lock at *time*=5 which would not be available until Processor A finished the insert operation.

```
bucket := record_key mod M;
if bucket < P
  bucket = record_key mod (2 * M )

Lock_Bucket(bucket);

newbucket := record_key mod Initial_M * ( 2 ** (level - 1) )

while ( newbucket <> bucket ) do begin
  Unlock_Bucket(bucket);
  bucket := newbucket;
  Lock_Bucket(bucket);
  newbucket := record_key mod bucket_m
end;
```

Figure 1.10 Rehashing loop

Time	M	P	Processor A	Processor B
1	4	1	Insert rec 17	Insert rec 21
2			Read P, M	Read P, M
3			Compute bucket=1	Compute Bucket=1
4			Lock bucket=1	
5				Wait for lock
6			Insert Record	
7			Bucket has overflowed	
8			Split Bucket into	
9			buckets 1 and 5	
10	4	2	P = 2	
11			level[1] = 2	
12			level[5] = 2	
13			Unlock bucket=1	
14				Get lock bucket=1
15				Compute newbucket=5
16				Unlock bucket=1
17				Lock bucket=5
18				Compute newbucket=5
19				Insert
20				Unlock bucket=5

Figure 1.11 Inserting a record correctly using rehashing

Another solution to this problem is possible without using bucket levels. Instead the processor must re-compute the bucket value using a fresh copy of *P* and *M* after the bucket has been locked. [SeRoPr88]

The following section describes some of the problems with implementing concurrent linear hashing in a NUMA architecture system.

1.4. Motivation

The work by Ellis and Litwin is adequate for a disk based system but there are a number of problems with the approaches which occur when linear hashing is implemented as a main memory database on a multi-processor NUMA system. The first problem is the cost of accessing central variables such as *M* and *P* and the central locks associated with those variables. In a multi-processor NUMA environment, access to a central data structure may become a hot spot causing the performance gains to be minimized as the degree of parallelism is increased.

Database reorganization is single threaded in previous implementations of linear hashing. This is a problem when a parallel system is continuously inserting records into a database faster than the database can be reorganized. Even if a processor continuously splits buckets, the database can never be re-organized fast enough to keep the load factor within a reasonable bound. If database reorganization is single threaded the rate at which the database can be reorganized is fixed regardless of the number of processors performing database operations. Since record insertion is multi-threaded the aggregate rate at which records are inserted will increase as processors are added. At some point the aggregate rate of insertions will be higher than the rate at which the database can be reorganized and the reorganization will continue to fall behind.

This thesis presents two new approaches to solve these problems. The problem of access conflicts for central control variables is minimized through "retry logic" and distributed data structures. Retry logic is an extension to the rehashing loop as proposed by Ellis. The general concept of retry logic is to substitute costly central variables and locks with distributed control variables and locks. Additional logic must be added to handle the case where the database control variables are in an inconsistent state. Distributed variables are nearly always consistent and in the case where the database control variables are inconsistent the retry logic detects the inconsistency.

The second technique is multi-threaded reorganization for linear hashed databases. Extensions to the data structures and algorithms allow the bucket splitting and bucket merging to operate in a multi-threaded manner. This way it is possible to maintain the database load factor regardless of the number of processors performing continuous inserts or deletes.

Linear hashing with retry logic and multi-threaded reorganization is presented as a solution to the need for a high performance hash based main memory database which can handle very high continuous transaction loads. This type of database system is ideal for use as temporary files in a relational database implementation or a database with dynamic files.

Chapter 2 presents an overview of the GP-1000 memory system and shows performance results for the NUMA memory system which relate to the design of a main memory database on the GP-1000. Chapter 3 describes distributed linear hashing. Chapter 4 describes the performance of the system as implemented on the BBN Butterfly GP-1000. Chapter 5 presents the conclusion and points to areas of additional research.

CHAPTER 2

MEMORY PERFORMANCE OF A NUMA SYSTEM

2. Memory Performance of a NUMA System

In implementing a main memory database system it is important to understand the underlying performance of the memory system. When using a Non-Uniform Memory Access (NUMA) architecture it is even more important to understand the performance characteristics of the various types of memory.

The BBN GP-1000 is a NUMA architecture system with up to 256 nodes connected by a memory sharing network. Each processor can access the local memory on the processor board or the memory on the boards of other processors. References to memory on another processor are automatically routed through the memory sharing network. Memory accesses are routed directly to the proper node without passing through any intermediate nodes. Remote memory can be accessed as characters, words, or multiword DMA transfers. Each node is a 68020 processor with up to 4M of RAM. Local memory access is about 10 times faster than remote memory accesses.

The logical layout of an individual GP-1000 node is shown in Figure 2.1.

Memory requests from the central processing unit are handled by the memory control unit. Requests for addresses in the local memory are fetched from the local on board memory directly. Requests for addresses which are outside the board are routed to the network interface unit. The network interface unit transmits the request to the correct processor and waits for the reply. The network interface unit takes the information received from the network and passes it back to the memory management unit to satisfy the original request [BBPI87].

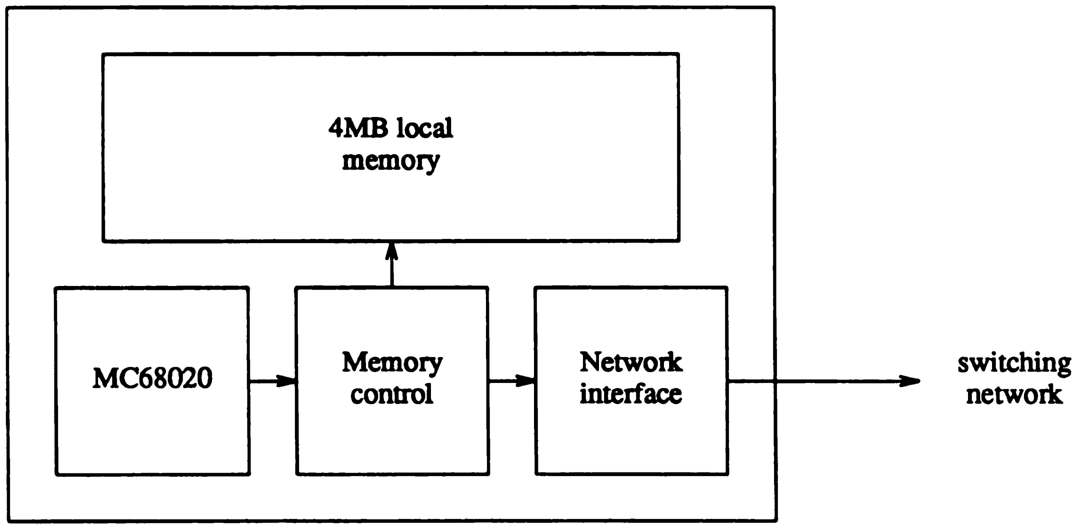


Figure 2.1 Layout of GP-1000 node

Data requests from other processors are received by the network interface unit which retrieves the requested information from memory and replies with the requested information over the network.

The switch provides a data path from each node to every other node. Memory references and data do not flow through any intermediate processors as in hypercube architectures. Each reference flows directly across the switch from the source processor to the destination processor. The memory sharing network is a multi-stage network of 4 by 4 crossbar routers [BBTu88]. Memory references are routed through the memory sharing network from crossbar to crossbar. Figure 2.2 shows a number of processors interconnected using the switch. The average time for a remote memory read is 7 microseconds. Of the 7 microseconds only 2 microseconds are used for the request to cross the switch and the response to return across the switch. The remaining time is the processing time for the request in the originating and destination nodes. The fact that the average request causes switch traffic for only about 28% of the total time of the request tends to reduce the overall switch contention [BBTu88].

The impact of unrelated switch activity on the performance of a transfer between two nodes has been shown to be negligible in [Thom86] and [Sev89].

The primary cause of delay which impacts the performance of an application occurs when two processors are accessing remote memory on the same node simultaneously as shown in Figure 2.2. Both CPU_0 and CPU_2 are simultaneously accessing the memory on CPU_1. Only one request will be processed. The request which fails will be retried by the original sending node after a delay. This memory contention occurs in the network interface regardless of the addresses of the memory in the destination node. Simultaneous accesses to different memory locations on the same processor will still cause a collision and a retry.

When two messages "collide" in the switch, one message is passed through

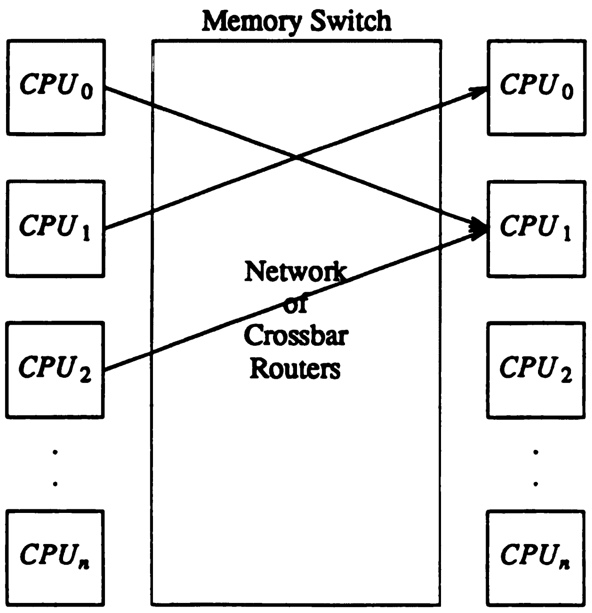


Figure 2.2 Logical layout of the GP-1000 switch

successfully and the other must be retried after a small delay. Two messages collide in the switch when both messages must be routed through the same output port of one of the crossbar switches. Figure 2.3 shows how two messages can pass through a network element successfully at the same time and when a collision occurs.

The rest of this chapter will cover the performance of the memory and memory sharing network on the GP-1000.

2.1. Experiment Details

A program was developed which allocated and referenced memory on the GP-1000 using different access patterns. The time and total memory transferred were recorded to compute the total system wide memory bandwidth. The bandwidth of the memory was measured on each processor and the total bandwidth for the entire system was accumulated at the end of the run.

The primary factors which affect performance are the location of the memory and the type of access being used. Memory location can be characterized as local memory, remote scattered memory and remote hot spot memory. Remote hot spot memory is when all of the processors are continuously accessing memory on one processor causing maximum memory contention. The remote hot spot memory has the worst performance

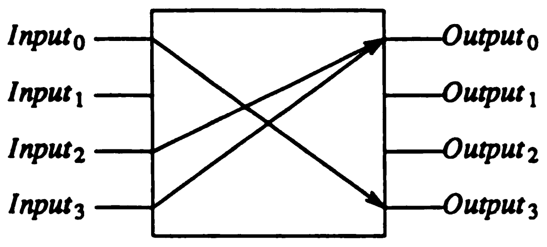


Figure 2.3 An individual crossbar in the switch network

because it has maximum memory contention. Remote scattered memory scatters the remote requests across all of the processors. Before each memory reference a random processor is chosen and the memory on that processor is accessed. Remote scattered memory should have low memory conflict.

A summary of the parameters which were tested is shown in Figure 2.4.

Each test was first run on a single processor and then on various numbers of processors.

2.2. Overall Memory Performance of the GP-1000

In the following performance figures the vertical axis is always the aggregate system bandwidth in megabytes transferred per second. The horizontal axis is the number of processors.

Figure 2.5 shows the effect of memory location on maximum memory bandwidth by using a copy loop. Figure 2.6 eliminates the local memory information and changes the scale to show more detail on the random and hot spot memory.

Test Parameters	
Parameter	Possible Value
Memory Type	Local Memory Remote with random references Remote with hot spot references
Transfer Style	Direct Memory Access (DMA) Copy loop
Block Length	1-1000 words
Iterations	Number of blocks to transfer for each processor
Read Rate	0 - 100% Read access remainder write access

Figure 2.4 Memory test parameters

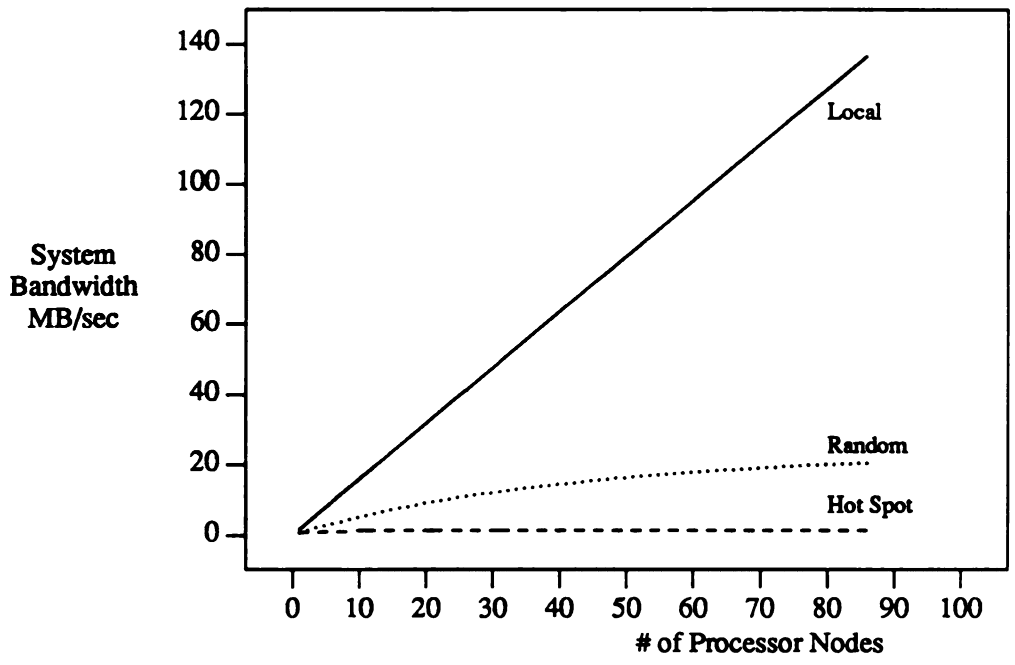


Figure 2.5 Effect of memory location on bandwidth

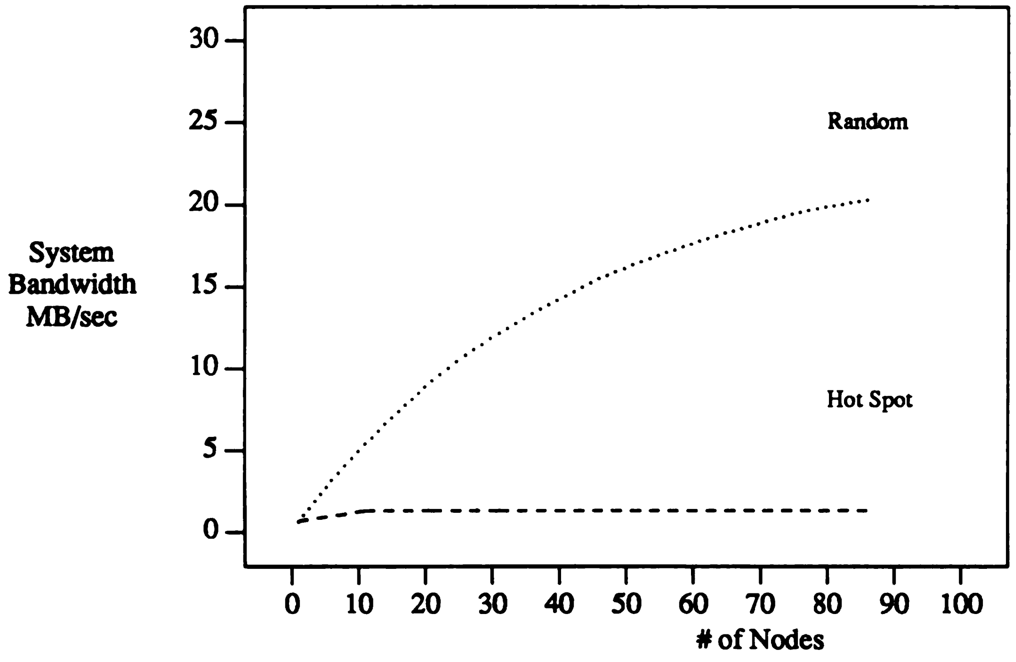


Figure 2.6 Effect of memory location on bandwidth (enlarged scale)

These figures show that the overall bandwidth of local memory is much greater than remote memory. Also hot spot memory has a fixed bandwidth of about 1.3 MB/sec which cannot be increased regardless of the number of processors which are trying to access the memory. This means that as more processors are trying to access the hot spot memory the additional processors simply spend most of their time waiting for the requests to be completed. This occurs because the network interface on the processor which contains the hot spot variable is running at its maximum capacity. [Mill89]

Figure 2.7 compares the effect of memory access style on the overall bandwidth. This was run with 64 byte transfers 50% read 50% write on random remote memory. DMA holds about a three to one advantage when transferring blocks 40 bytes or larger to and from remote memory. Below 40 byte blocks the DMA overhead causes the DMA transfers to have poorer performance than a simple copy loop [Sev89, Mill89].

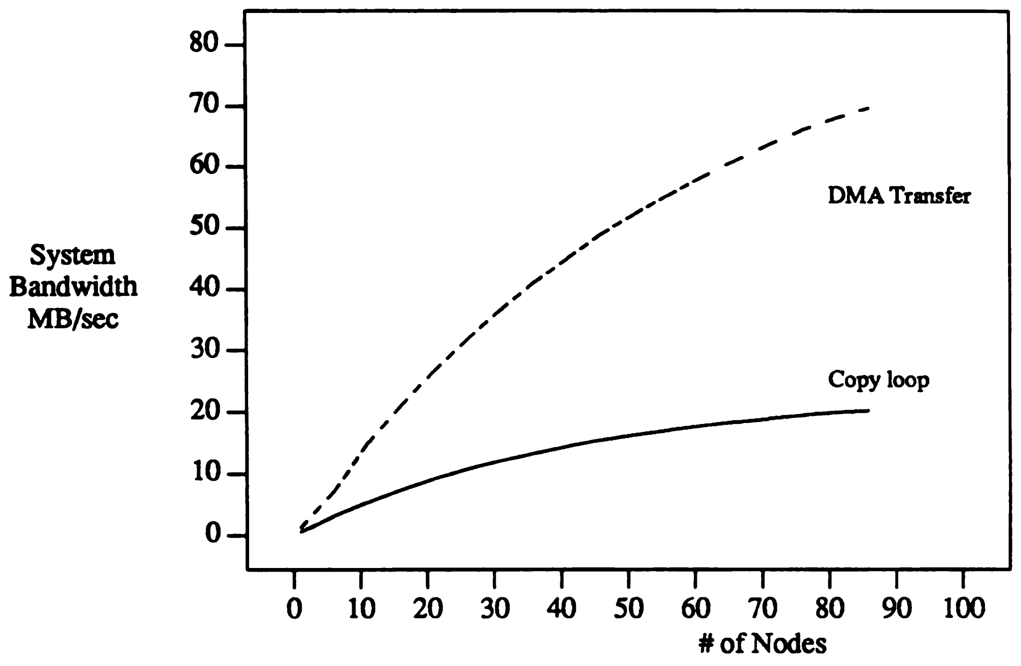


Figure 2.7 Effect of transfer style on overall bandwidth

The important results which affect the design are: (1) Hot spot memory has fixed bandwidth. As more processors access a hot spot the amount of time each processor must wait increases reducing overall performance. (2) The overall memory bandwidth of the local memory is very large. (3) The overall bandwidth for remote memory increases as processors are added long as references are random. (4) DMA is effective in retrieving blocks of data from remote processors.

In the implementation of a main memory database these results indicate that access to central variables which can cause hot spots should be avoided whenever possible. The use of local memory to cache global data structures will generally result in improved performance. Randomly distributed memory references such as hashed searches are handled well by the GP-1000 memory system.

CHAPTER 3

DISTRIBUTED LINEAR HASHING

3. Distributed Linear Hashing

The objective in distributed linear hashing is to provide control insuring database-wide consistency while minimizing the requirement for shared database wide variables and allowing multithreaded reorganization.

The linear hashing data structures are extended to improve performance in a main memory database implementation. In linear hashing the records are distributed into buckets which are normally stored on disk. In distributed linear hashing buckets are replaced by directories. A record key hashed to select the particular directory for a record and a second hash function is used to select the position within the directory. Figure 3.1 shows the structure of the directories and records. The pointers to all of the directories are cached in each processor. Each entry in the directory points to the head of a record chain. Collisions in a particular entry in the directory are resolved by adding the record to a linked list of records for the entry. Collisions are tracked to determine the average chain length for all of the chains which exist in the directory.

When a directory is split the data is not actually moved as in standard linear hashing. The records are re-linked into the new directories using the stored key information in each record. This reduces the memory contention during split operations.

Each directory is protected by a directory lock. The directory lock is a reader/writer lock. Multiple processes can simultaneously access a directory for reading but only one process can access a directory for writing. The directory locking protocol insures that a request for a write lock will eventually be satisfied even if there are continuous readers in

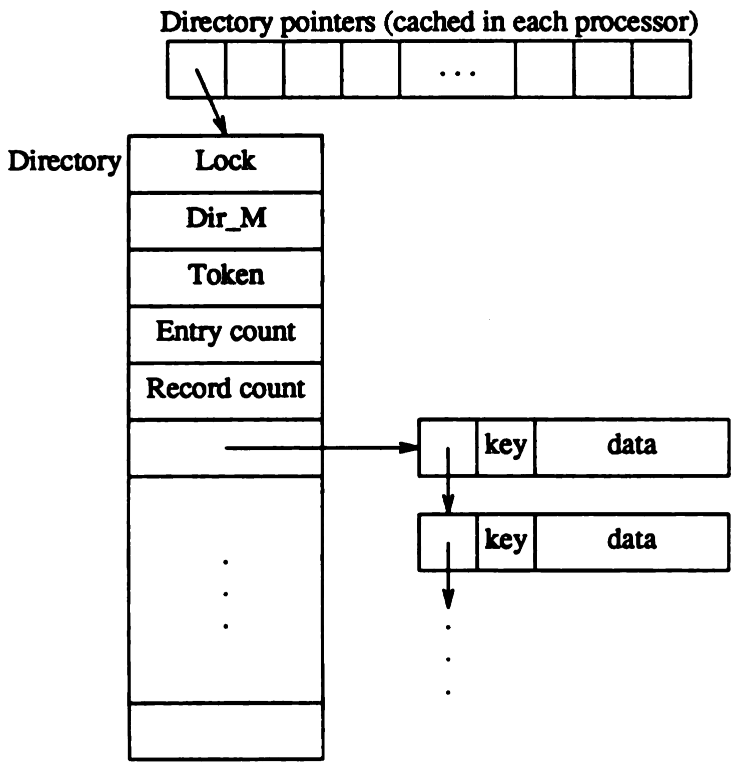


Figure 3.1 Main memory data structures

the directory. See Appendix B for details on the reader/writer locking operations.

The directories also contain the value for *M* which was used in the hash computation when placing the records into the directory during the most recent split or merge of the directory. Once a directory has been locked, this value can be used to determine if a particular record belongs in this directory regardless of the values of central control variables. This variable is equivalent to the level variable in concurrent linear hashing.

Dir_M is the only database-wide control variable which absolutely determines the proper directory for a particular record at a particular instant of time. To find and lock the proper bucket the central variables can be used as a "hint" or initial guess as to the proper bucket. After the bucket is locked Dir_M must be checked to insure that the calculation was correct.

3.1. Retry Logic

The algorithm which finds and locks the proper bucket for a given key compensating for inconsistent control variables is called "retry logic". Initially the directory is computed using the record key value and the current values of the control variables without locking the control variables. The directory indicated by the computation is locked. The proper directory is re-computed using the record key value and the value for Dir_M stored in the directory. If this indicates a different directory, the directory is unlocked and the second directory is locked. Once the second directory is locked the proper directory is again recomputed using the Dir_M value in the second directory.

This process is repeated until the directory indicated using the Dir_M stored in the directory is the directory which is currently locked.

When a process does not have a directory locked, any number of other database operations executing on other processors may change the structure of the database causing a database control variable to become invalid for use in a hash computation. Once a process has a directory locked, any number of database operations can execute changing the structure of the database *except* for the structure of the records in the directory which has been locked.

The distributed locks and distributed control variables stored in each directory substitute for centralized locks and control variables. The benefit of retry logic is that there are no central variables or locks to cause hot spots. The additional cost of retry logic is incurred when the incorrect directory is locked and additional directory locks are required. The performance impact of retry logic is compared to the performance impact of central variables in the performance analysis section.

Figure 3.2 shows the algorithm for locking a directory using retry logic to find the proper directory for a key. This retry logic allows a process to use database control

```
dir_number := linear_hash(key, P, M);
dir_pointer := dir_lock(dir_number);
new_dir_number := key mod dir_pointer.Dir_M

while (new_dir_number <> dir_number) do begin
  dir_unlock(dir_number);
  dir_number = new_dir_number;
  dir_pointer := lock_dir(dir_number);
  new_dir_number := key mod dir_pointer.Dir_M
  end
```

Figure 3.2 Pseudocode for finding the proper bucket

variables without locking the variables. The only requirement for retry logic to find the proper bucket is that the Dir_M values accurately reflect the contents of the bucket.

Because of the retry logic it is not necessary to split or merge directories in strict linear order. A number of directories can be split simultaneously as long as the value for Dir_M stored in each directory is maintained properly. With extensions to the definition of the split and merge operations multi-threaded database reorganization is possible.

3.2. Multi-threaded Reorganization

In concurrent linear hashing and sequential linear hashing, *P* always indicates the next bucket to be split, the bucket being split, or the bucket being merged. If a merge operation or a split operation was in progress, *P* always points to the directory being split or merged. Because *P* points to the directory being split or merged until the operation is complete database reorganization is single threaded. When a split or merge operation is required in distributed linear hashing *P* is moved before the merge or split operation is started.

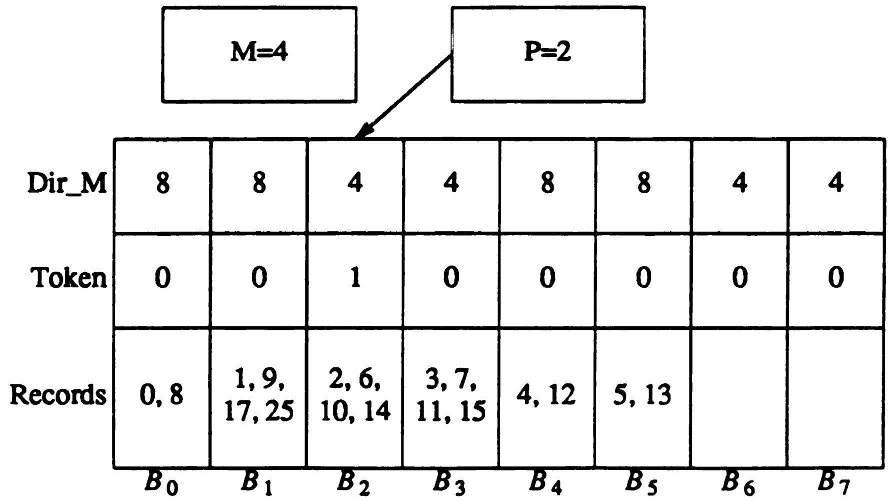
Moving *P* before the actual split or merge operation is started allows the basic data-base re-organization to be multi-threaded. *P* can point to a directory which has been split at the current level of *M* or a directory which has not been split at the current level of *M*.

The actual directory pointed to by *P* depends on whether or not the last operation was a split or merge.

Certain database operations such as doubling the value for *M* must have a lock on the *P* directory D_P before proceeding. A central lock to protect *P* which would cause memory contention. This implementation uses a distributed lock created by adding a single bit to each directory. This flag is called the TOKEN flag as shown in figure 3.3.

Only one directory has the TOKEN flag set at any one time. A process must have the directory lock in the directory with the TOKEN flag set to begin a directory split, directory merge, modify *P* or move the TOKEN.

Distributed linear hashing extends the split operations and merge operations to handle the case where a split or merge is not necessary. A process can detect if a directory has been split by comparing Dir_M in the directory to the current value for *M*. When a split is not necessary, the TOKEN is moved to the right but the directory is not split because it already is split at the current value for M. This provides some control over



Dir_M	8	8	4	4	8	8	4	4
Token	0	0	1	0	0	0	0	0
Records	0, 8	1, 9, 17, 25	2, 6, 10, 14	3, 7, 11, 15	4, 12	5, 13		
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7

Figure 3.3 Distributed linear hashed database

wasted splits and merges when record inserts are interspersed with record deletes. The TOKEN can be moved by successive directory underflows and overflows without actually performing any physical splits or merges.

3.3. Distributed Index Computation

The retry logic removes the need to provide locks on the database control variables. Without locks the database control variables are essentially "hints" rather than exact values. From a performance point of view the more accurate the database control variables are the less the retry logic will execute.

There are three techniques which can be used to maintain these database control variables. The simplest and most accurate is to simply have a single copy of the variable shared by all of the processors. Another technique is to have a copy of the variable on each processor with a global pointer to each local copy to allow the local copies to be updated. The third technique is to maintain the database control variable independently on each processor based on the information available to each processor. Combinations of these techniques can also be used. Figure 3.4 summarizes the advantages and disadvantages of each technique.

Technique	Advantages	Disadvantages
Central	Most accurate	Hot spot at high usage levels
Distributed	Access cost is low	Update cost is high
Local	Access cost is low Update cost is low	Least accurate

Figure 3.4 Comparison of techniques for database control variables

3.4. Maintaining M

The distributed technique was chosen for maintaining *M* as *M* is seldom updated. When *M* is doubled or halved during database reorganization, the processor changing *M* must go through each of the processors and update each processor's local copy of *M*. When a process intends to split the D_{M-1} directory, instead of moving *P* to the right one directory, *P* is set to zero and the *M* values are doubled. To double the *M* values a processor must hold the directory lock on the TOKEN directory and the TOKEN directory must be the D_{M-1} directory. To halve the values for *M* the process must have the lock on directory D_0 and the TOKEN flag must be set in the directory. Since only one directory is allowed to have the TOKEN flag set at one time, the update of the *M* values can only be performed by a single process. During the period when the *M* values are being updated, it is possible for a processor to use the incorrect value of *M*.

The data structure for *M* is shown in Figure 3.5. This technique for maintaining the values for *M* uses a very small amount of memory bandwidth compared to the memory requirements for a shared value for *M*. The values are only updated when the value for *M* is doubled or halved as the result of extensive database reorganization. In a typical run *M* is modified in one out of 50000 database operations.

3.5. Maintaining P

Choosing the technique for maintaining *P* is more complicated. Maintaining P as a distributed variable is not feasible because during periods of inserts and deletes *P* is changed quite frequently. It is possible to maintain an approximation to *P* independently in each of the local processors.

To maintain *P* independently, each processor examines the value for the Dir_M in each directory and compares it to the value for *M*. If the directory number is higher than the processor value for *P* and the Dir_M is 2**M* the directory has been split and the local

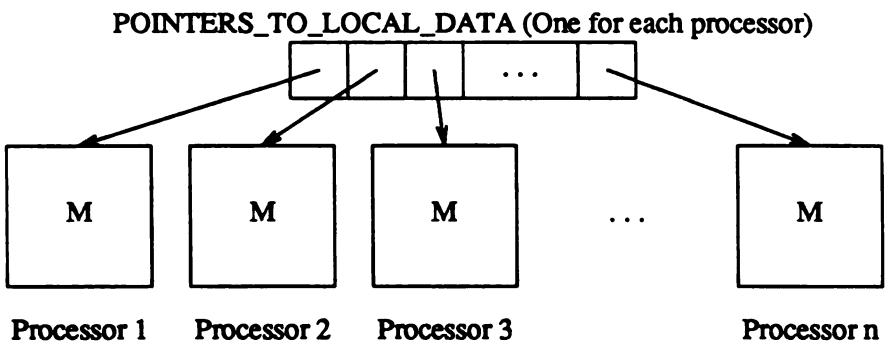


Figure 3.5 Data structure for distributed M

copy of P is too low and is set to the number of the current directory. If the directory number is lower than the processor value for P and the Dir_M is the same as M, then P is too high, the directory has been merged and the local copy of P is set to the number of the current directory.

In Figure 3.6 the local copy of P is too low because there have been a number of splits causing the global value for P to move to the right while the local copy of P was not updated. When the processor uses the local value for P there will only be a problem with keys which are hashed into buckets between the local P and the global P. In addition on the average only half of the records will be hashed improperly in the buckets between the local P and the global P. For records which hash into buckets below local P the hash computation would produce the same results if the local P was used in the computation or the global P was used. In the same way records in buckets from global P up to $M-1$ will be hashed properly using either value for P.

An important cost of maintaining P in each of the processors is the number of incorrect hashes which will occur because the local copies of P are out of date. In the next section the cost of this technique for maintaining P is compared with using a global

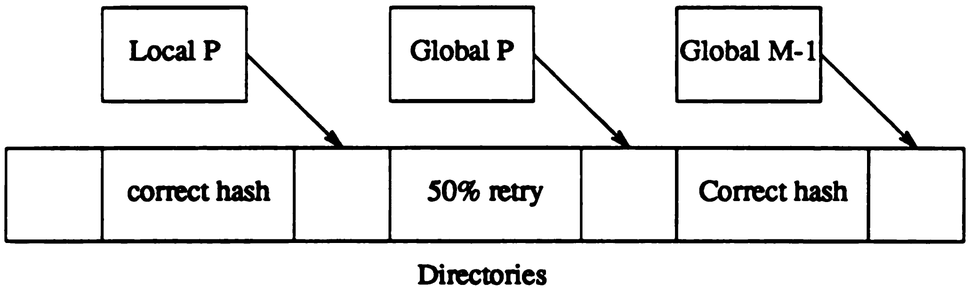


Figure 3.6 Maintaining local copies of P

value for *P* for each hash computation.

CHAPTER 4

PERFORMANCE ANALYSIS

4. Performance Analysis

This chapter discusses the performance of the distributed linear hashing system implemented on the BBN GP-1000 under the Chrysalis operating system.

The overall performance of the system at large numbers of processors is analyzed. The performance impacts of design decisions for database control variables and the effectiveness of the database organization are analyzed.

4.1. Experiment Details

The goal in measuring the performance of the implemented system was to measure the maximum continuous throughput of the database system. The performance measurement does not include (1) operating system overhead to initialize each processor, (2) time to open the database on each processor, (3) time to close the database on each processor or (4) operating system overhead when terminating the tasks on each processor.

The only segment which was measured was the time spent during continuous database operations as shown in Figure 4.1.

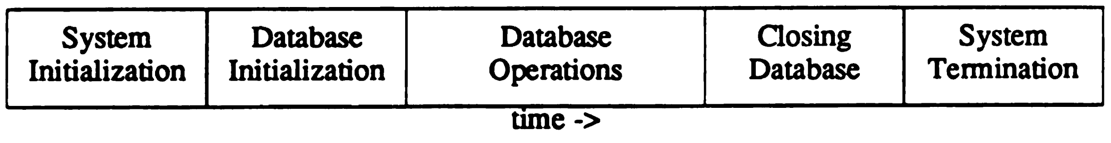


Figure 4.1 Experiment details

The total aggregate database operations per second were measured while varying the number of processors. Under ideal conditions, the total number of operations per second should increase linearly as the number of processors are increased. Total number of operations per second was chosen over speedup because it shows the speedup but also allows quantitative comparison between different types of runs on the same graph.

4.2. Performance impact of distributed control variables

This section examines the performance impact of the choice of storage for the database control variables as described in section 3.3. The following implementations are compared: (1) central *P* and central *M* protected by central locks, (2) distributed *M* with central *P* without any central locks and (3) distributed *M* with independent copies of *P* in each processor without any central locks. Implementations 2 and 3 use retry logic to substitute for the central lock.

The performance of the three implementations when performing continuous read operations is shown in Figure 4.2. This shows that the implementation using central data structures protected by central locks improves in overall performance until about 10 nodes. Then the overhead of the hot spots caused by these central data structures begins to dominate the computation and performance stops improving. The performance of another main memory file system on the Butterfly shows similar performance. The BBN-RAM file system [BBRf86] performance levels off after 15 nodes doing continuous activity.

The implementation using central *P* and distributed *M* without using locks to protect the values maintains linear performance up to 40 nodes. The elimination of the central locks and the central value for *M* reduces the hot spot memory accesses by 75%. In the implementation using locks each database operation accesses the lock twice and the values for *P* and *M*. In the implementation using central *P* there is only a single access per operation. However the performance is still impacted by the single hot spot access at high levels of parallelism.

The implementation using independent copies of P in each processor has the best performance at high levels of parallelism. This implementation never accesses a central variable during the read operations so it is only limited by the performance of the memory sharing network under random memory access patterns. The performance is still improving at 80 processors.

In the read performance the implementation using independent copies of *P* is the superior implementation. However read operations do not modify the database. Database reorganization requires access to central information and also causes the independent copies of *P* in each processor to become invalid causing retry logic.

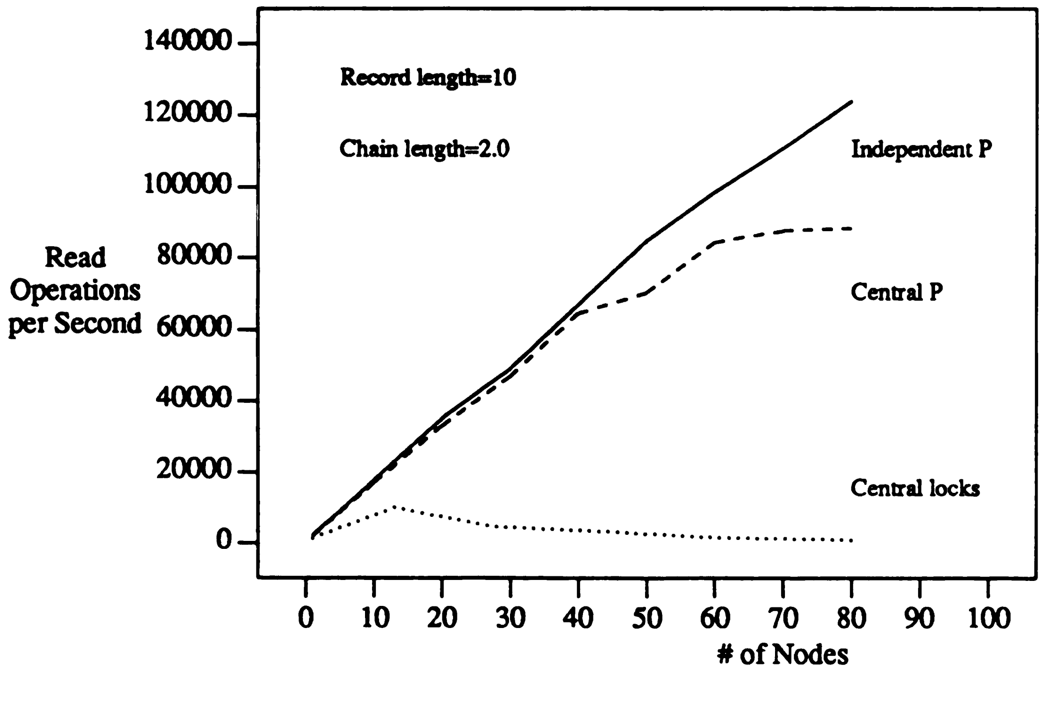


Figure 4.2 Read performance of different implementations

Figure 4.3 shows the performance of the different implementations when performing continuous insert operations and database reorganization. In this figure the implementation using central control variables and central locks maintains performance improvement until 15 processors where memory conflicts limit any additional performance improvement.

The performance is much closer for inserts than reads when comparing the performance of the implementation using central *P* with the implementation using independent *P*. At 50 nodes and below the implementation using central *P* without locks performs as well and at times performs better than the independent *P* implementation. Above 50 nodes the independent *P* implementation continues to increase performance while the central *P* implementation performance begins to level off.

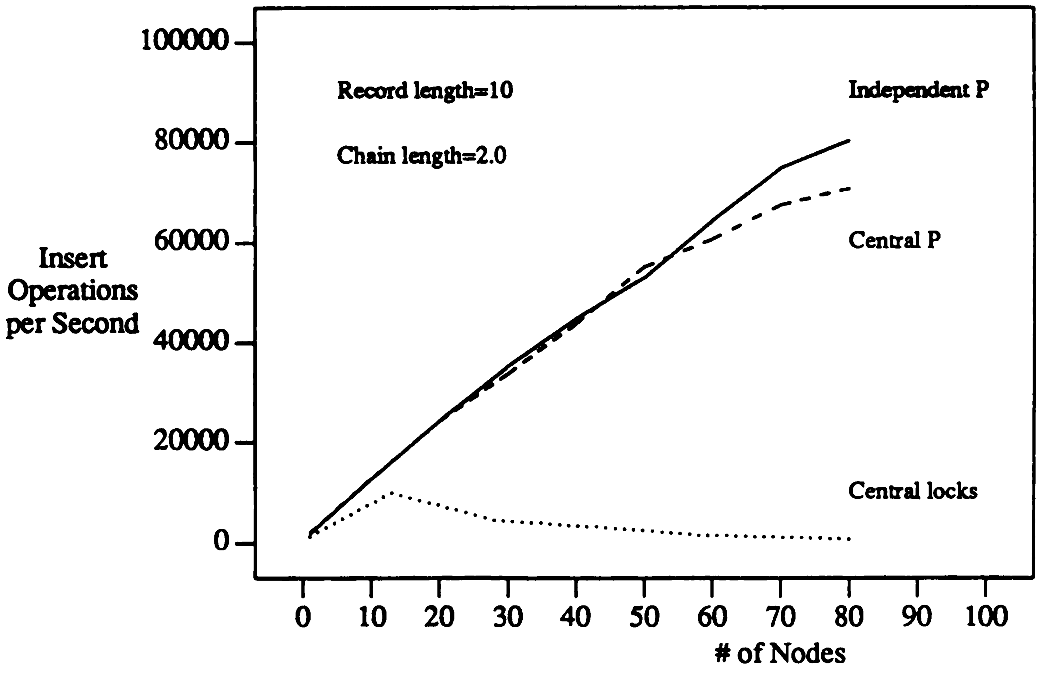


Figure 4.3 Insert performance

There are several reasons for this performance. Insert operations are inherently slower because they include database reorganization. This delays the point that memory contention becomes a performance bottleneck. The hot spot memory has fixed bandwidth regardless of the number of processors and if each processor is accessing the hot spot less often it will take more processors to reach the maximum bandwidth for the hot spot. The database reorganization makes use of the central variables to control the database reorganization. When performing inserts even the independent *P* implementation will have to access central data structures during reorganization causing some memory contention. Because the database is being reorganized the value for *P* is changing causing the independent copies of *P* to become inaccurate increasing the time spent in retry logic.

The percentage of database insert operations requiring an additional bucket lock because of retry logic is shown in Figure 4.4 for both the independent *P* implementation and the central *P* implementation. The implementation with the central locks is not shown because there are no retries required for the central lock implementation. The percentage of retries required when using a central value for *P* without locking *P* is very small. At 80 processors 3 out of 1000 bucket accesses will have to be retried because the value for *P* is inaccurate. The implementation using independent *P* has a retry percentage ranging from 4% to 7%. This is an additional overhead for each database operation on the average which will have an impact on the performance.

Unlike the overhead of the central data structures, the overhead of retry logic does not cause hot spot access. Since retry logic only uses the bucket locks and bucket locks are randomly distributed among the processors the retry logic causes additional *random* memory references. The bandwidth of random memory references increases as the number of processors are increased. The overhead of retry logic has an impact on each

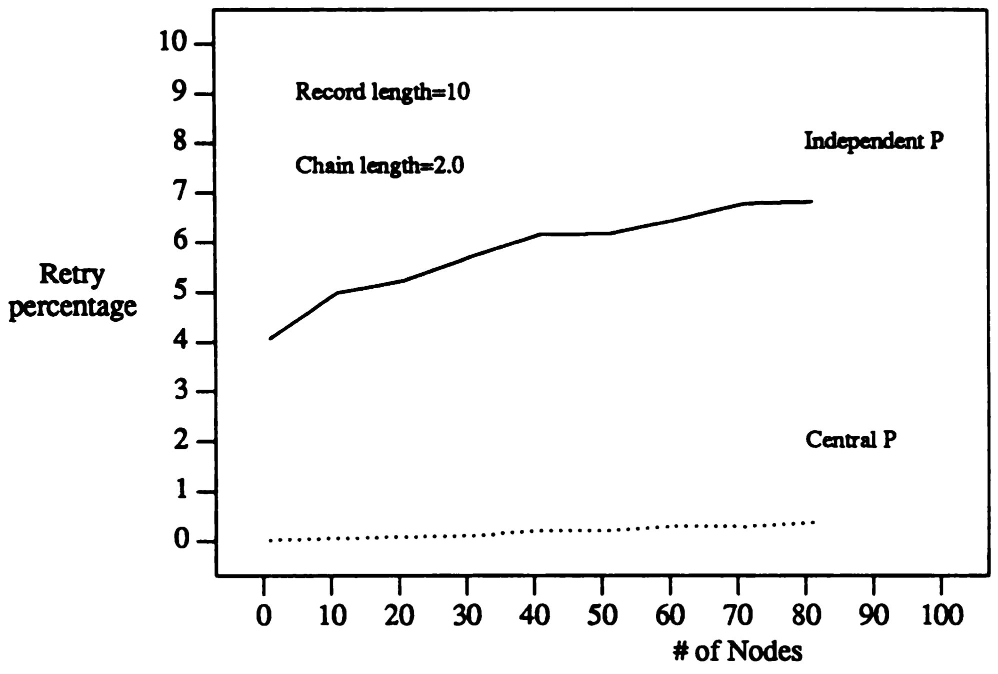


Figure 4.4 Retry percentage for various control strategies

processor but as the number of processors are increased the overhead per processor remains fixed and allows the overall system performance to improve as additional processors are added. The cost of central data increases for each processor as processors are added to the point where the access to the central data structure dominates the entire computation causing overall performance to level off.

The remaining performance figures are using the independent technique for maintaining *P*.

4.3. Performance of the overall system

The implemented system has been tested under continuous insert operations, continuous read operations, and continuous delete operations to determine the overall system capabilities for the independent *P* implementation.

Figure 4.5 shows that read only operations have the highest performance because no retry logic is required, no database reorganization is required and no memory allocation is required. Both insert and delete operations perform database reorganization. Delete is faster than insert because memory deallocation is simpler than memory allocation.

4.4. Performance of Database Reorganization

The performance figures include database reorganization for insert and delete operations. Under continuous delete or insert loads the database is being reorganized continuously to maintain the desired maximum allowed chain length. Distributed linear hashing allows this database reorganization to be performed in a parallel fashion. This allows the reorganization to keep up with the incoming operations. Figure 4.6 compares the actual chain length with the multi-threaded implementation with the chain length for single

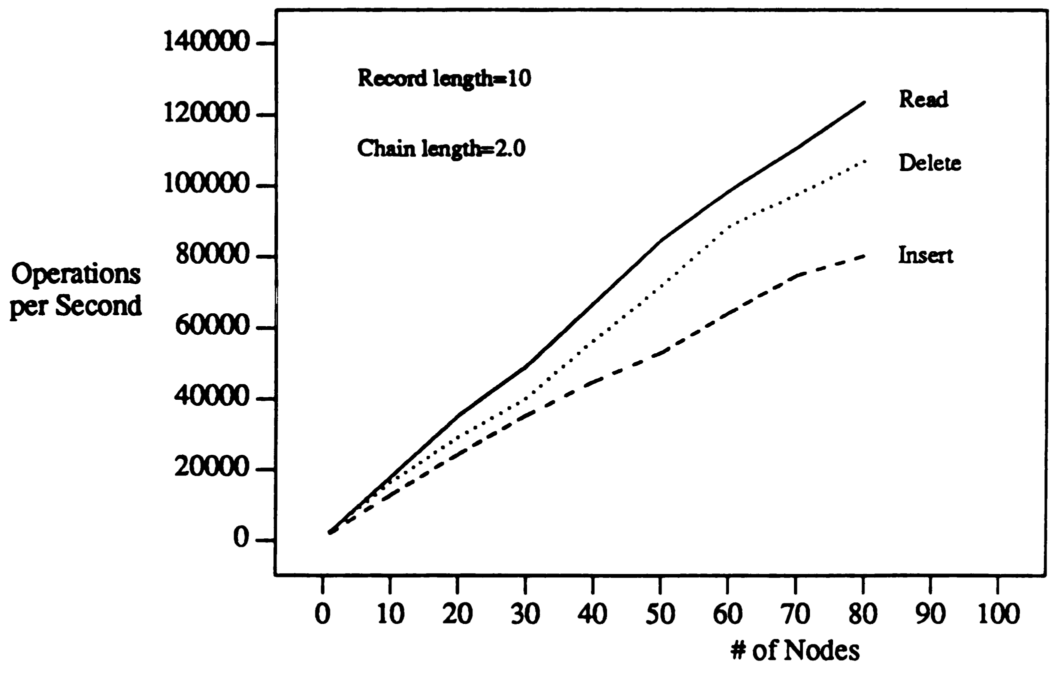


Figure 4.5 Comparison of inserts, deletes and reads

threaded continuous reorganization. Both implementations are trying to maintain a maximum chain length of 2.0 records.

The multi-threaded database reorganization is able to maintain the desired chain length regardless of the number of processors. The single threaded implementation cannot keep up with the required reorganization above 10 processors.

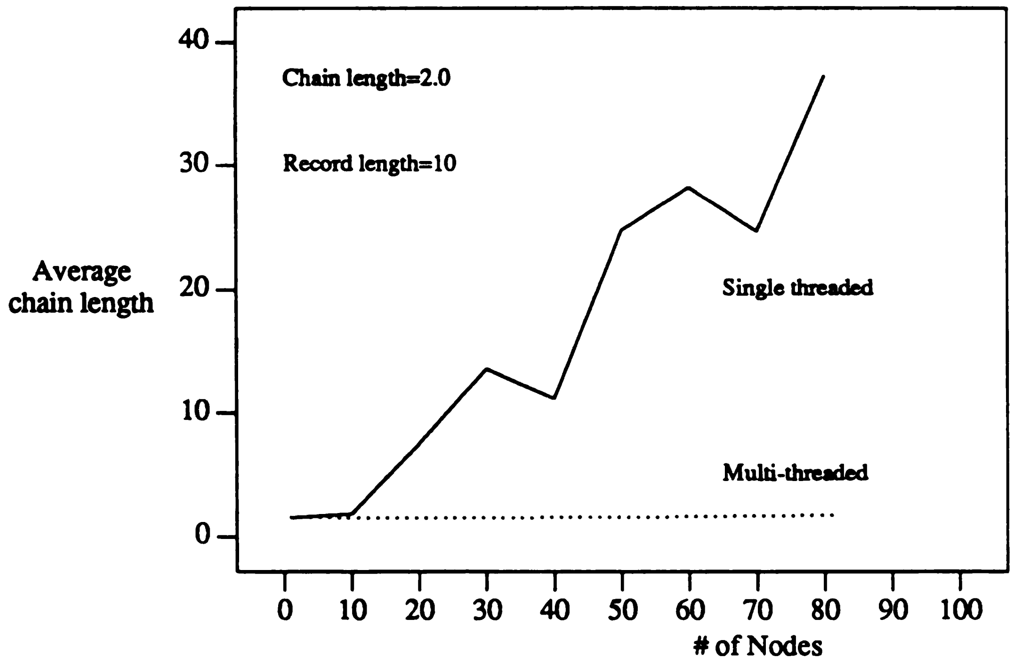


Figure 4.6 Chain length control using multi-threaded reorganization

CHAPTER 5

CONCLUSION

Distributed linear hashing is shown to be a useful tool for implementing a main memory database on a NUMA architecture system. Algorithms are developed which make minimal use of centralized variables or locks. Distributed techniques are used for protection, consistency, and reorganization. The additional fixed overhead of the retry logic for these distributed techniques is small compared to the cost of accessing central data structures at high levels of processors.

Multithreaded reorganization is introduced to handle database reorganization at high levels of continuous transactions without falling behind or requiring maintenance processes.

A practical main memory database system can be implemented on a general purpose NUMA architecture system providing excellent overall performance and good real time performance.

The process of taking a parallel processing database implementation designed for a disk environment and implementing the system in main memory required is not straightforward. The overall performance of the main memory system may be dominated by some portion of the operation which in a disk based system is considered to be negligible.

This work shows the KDL file system has excellent performance under continuous database operations. Higher level work has been done [Wolb89] using the KDL file system as a tool to implement parallel projection. The system has shown excellent performance in this application as well. It is expected that the file structure will perform well on selection, join, and non-key search operations based on the results of the projection implementation.

Distributed linear hashing has superior performance to previous implementations of parallel linear hashing [SeRoPr88] because of the increased use of distributed data structures.

FUTURE WORK

The performance of linear hashing database reorganization should be studied to determine the effect on chain length because only the *P* directory can be split. Extensible hashing allows the directory which is overflowing to split. This has the advantage of maintaining the chain length with less variation but the retry cost will be higher because *P* cannot be used to make the initial hash computation more accurate. Other forms of distributed hashing should be investigated.

Distributed linear hashing should be implemented on a variety of other multiprocessor systems such as the NCUBE, Hypercube, or Sequent. The design choices for the techniques used to maintain the database control variables may be quite different on each of these architectures.

Additional high level database operations should be implemented using the KDL file system. Selection and join should perform well using the KDL file system. The system has been designed to perform high speed non-key searches using all of the parallel resources of the system.

APPENDIX A

KDL-RAMFILE SYSTEM DOCUMENTATION

This documentation describes the application programmer interface to the KDL-RAMFILE system as implemented on the Butterfly. The first section is the calling sequences for the routines. The second section is a simple example of the use of the KDL-RAMFILE system.

SUBROUTINE SPECIFICATIONS

The following is a list of the routines in the KDL-database system in alphabetical order.

InitKDLSystem();

Must be called after the Uniform System has been initialized and before KDL_create is called.

KDL_close(kd_od);
*KD_ODES *kd_od;*

Closes the open KDL file. Must be called for each open descriptor when database activity is finished on a processor.

*KD_CDES *KDL_create(file);*
*char *file;*

Creates the global data structures for the KDL system. The parameter is ignored. This must be called once on a single processor to create each KDL file. The return value for KDL_create is the input to the KDL_open routine.

KDL_dumpout(kd_cd);

*KD_CDES *kd_cd;*

Dumps out the entire database. Should only be used on small databases.

KDL_insert(kd_od, rec_buf,keyval);

*KD_ODES *kd_od;*

*char *rec_buf;*

KEY keyval;

Inserts the specified record in the database. Returns non-zero if an error occurs during the insert such as duplicate key if duplicate key processing was requested. The keyval parameter contains the key for the record. The rec_buf parameter contains RECLEN data bytes for the record. The keyval is stored in the database along with the record information so it is not necessary to include the keyval in the rec_buf parameter.

*KD_ODES *KDL_open(kd_cd,prot);*

*KD_CDES *kd_cd;*

int prot;

Opens the KDL database for reading and inserting in a particular processor. The first parameter is the descriptor returned by the KDL_create routine. The prot parameter specifies if the file should be opened read-write or read only. Once one processor has opened a file readonly, no other processor can open the file read-write until all of the processors with the file open readonly have closed the file. The advantage of opening the database readonly is that the directory locks can be bypassed because no processor has write access to the file.

KDL_read(kd_od, rec_buf, keyval);

*KD_ODES *kd_od;*

*char *rec_buf;*

KEY keyval;

Returns the data stored in the record at keyval. Returns non-zero error if the record does not exist.

KDL_statprint(kd_cd);

*KD_CDES *kd_cd;*

Prints out internal statistics information on what has been done in the database since the KDL_create. This routine will print out the internal KDL performance information. Information included is the total time between the KDL_create and KDL_statprint, average number of inserts per unit time, average number of times reads had to traverse a record chain, etc.

This information can be used to tune the performance of the system.

SetKDLConfig(param, value);

int param;

int value;

Sets the tunable values as specified in Figure A.1. This routine must be called after the Uniform System has been initialized. This routine must be called prior to any KDL_create calls. Any calls to this routine after the KDL_create call will not affect the files which have already been created. InitKDLSystem will reset all values to their defaults. SetKDLConfig can also be called between KDL_create calls to set different parameter values for different KDL files.

KDL Parameters		
Parameter	Description	Default
INDBLOCK	Size of each directory. Should be a prime number.	61
SPLITVAL	The maximum average chain length allowed in a directory before splitting the P directory.	2.00
INITIALM	Initial number of directories.	128
MAXBLOCK	Maximum number of directories in database.	16K
DATBLOCK	As records are added, the KDL system requests a large block of memory from the Uniform System and puts the records into the block. This number must be larger than a record and should be from 10 to 1000 times the size of a record.	8192
RECLEN	The record size for fixed length records.	60
SUPDUPS	TRUE to suppress duplicate keys. This option causes a performance degradation because as records are inserted existing records must be searched. When SUPDUPS is TRUE, it implies SORTCHAIN.	FALSE
SORTCHAIN	TRUE if chains are to be sorted within each index entry. Primarily used with SUPDUPS.	FALSE

Figure A.1 Configurable Parameters

EXAMPLE USAGE

```
/* This is a simple demonstration of the use of the KDL RAMFILE.
  Charles Severance - Michigan State University.  */
#include <us.h>
#include "x5.h"
main() {
 KD_CDES *kd_cd;      /* Create descriptor */
 KD_ODES *kd_od;      /* Open descriptor */
 KEY hash_keyval;
 char rec_buf[50];
/* Initialize the Uniform System */
 InitializeUs() ;
/* Initialize the KDL System and change some defaults */
 InitKDLSystem();
 SetKDLConfig(RECLEN,30);
 SetKDLConfig(SPLITVAL,1.46);
/* Create and open a KDL Ramfile */
 kd_cd = KDL_create("testdatabase");
/* Optionally generate additional tasks on other nodes.  KDL_open
  may be called once on each node for each KDL file. */
 kd_od = KDL_open(kd_cd,KD_OPEN_RW);
/* Insert, read, and delete a record */
 hash_keyval = 1234;
 sprintf(rec_buf,"record information");
 err = KDL_insert(kd_od,rec_buf,hash_keyval);
 err = KDL_read(kd_od,rec_buf,hash_keyval);
 err = KDL_delete(kd_od,hash_keyval);
 KDL_close(kd_od);
}
```

APPENDIX B

PSEUDOCODE FOR KDL RAMFILE SYSTEM

This appendix shows the pseudocode for selected portions of the algorithm for distributed linear hashing.

In the implementation there are several important points to note. The only time global variables are accessed is in the split and merge routines. For a normal split or merge, the variable GLOBAL_P is accessed once and changed once. If M is doubled or halved, the processor doubling M must update all of the values for M in each processor. This is a time consuming activity but it is seldom performed and only after extensive database reorganization so updating the distributed copies of M is a very small part of the operation. The cost of updating the distributed copies of M is small compared with the cost of accessing a central variable for M.

If a processor cannot lock the *P* directory for a split or merge or "misses the token", the processor does not wait. This insures that the insert and delete operations will not be slowed down waiting for a split to complete. The split or merge will be deferred until some other transaction when the directory is not busy.

The merge operation is very similar to the split operation in terms of the data and computing requirements. Previous implementations required different levels of locking for merge and split operations giving preference to the split operation.

If a split (or merge) operation notices that a directory has already been split (or merged) the TOKEN is simply moved and no re-organization is done. This takes much less resources than a split or merge and will often occur if deletes are interspersed with inserts. This reduces the amount of database thrashing when there are interspersed inserts and deletes. The implementation of the directory locks insures that a process requesting an exclusive lock will eventually get access to the directory regardless of the number of readers attempting to lock the directory. This occurs because the process requesting the exclusive lock blocks other readers while waiting for the read locks to be cleared.

Pseudocode:

Global Shared Data

```
int NUMPROC;          /* Set to the number of processors */
int GLOBAL_P
int *GLOBAL_DISTM_POINTER[NUMPROC];
```

Local Data For each Processor

```
int DIST_M;
int LOCAL_P;

/***************************************************************************/

lock_dir_retry_logic(key,locktype)
int key;
{ /* Beginning of routine lock_dir_retry_login */
  dir_number = linear_hash(key,DIST_M,LOCAL_P);
  dir = lock_dir(bucket_number,locktype);
  update_LOCAL_P(dir_number,dir->DIR_M);
  newdir_number = key mod dir->DIR_M;
  while(newdir_number != dir_number ) {  /* Retry logic */
    unlock_dir(dir_number);
    dir_number = newdir_number;
    dir = lock_dir(dir_number,locktype);
    update_LOCAL_P(dir_number,dir->DIR_M);
    newdir_number = key mod dir->DIR_M;
  } /* End while.  Now we have the proper directory locked for the specified key */
} /* End of routine lock_dir_retry_logic */


/***************************************************************************/

/* This routine implements the local technique for maintaining P. */

update_LOCAL_P(dir_number,dir_number)
int dir_number, dir_number;
{ /* Beginning of routine update_LOCAL_P */
  if ( dir_number > LOCAL_P and dir_number >= (DIST_M * 2) ) {
    LOCAL_P = dir_number;
  }
  if ( dir_number < LOCAL_P and dir_number <= DIST_M ) {
    LOCAL_P = dir_number;
  }
} /* End of routine update_LOCAL_P */

/***************************************************************************/
```

```
lock_dir(dir_number,locktype)
int dir_number;
int locktype;
{ /* Beginning of routine lock_dir */
  dir = locate_dir(dir_number);
  Lock(dir->exclusive_lock);
  if ( locktype == exclusive ) {
    /* Wait until all read locks have been released */
  }
  return(dir);
} /* End of routine lock_dir */

/****************************************************************************/

insert(key,data)
int key;
char data[];
{ /* Beginning of routine to insert a record */
  dir = lock_dir_retry_logic(key,exclusive);
  /* Now the directory is searched and the record is inserted. While the
  lock is held in the directory, we have exclusive access to the directory. */
  unlock_dir(dir->number);
   if ( overflow ) split();
} /* End of routine to insert a record */

/****************************************************************************/

delete(key)
int key;
{ /* Beginning of routine delete */
  dir = lock_dir_retry_logic(key,exclusive);
  /* We have exclusive access to the directory. The directory is searched and
  the record is deleted.*/
  unlock_dir(dir->number);
  if ( underflow ) merge();
} /* End of routine delete */

/****************************************************************************/

read(key, data)
int key;
char data[];
{ /* Beginning of routine read */
  dir = lock_dir_retry_logic(key,read)
  dir->read_locks++; /* Set the read lock */
  unlock_dir(dir->number);
  /* We now have a read lock on the directory. No process can get exclusive access
  to the bucket but other processes can get additional read access. The
```

```
 directory is searched and the record is read if it exists. */
  dir->read_locks--;  /* Release the read lock */
} /* End of routine read */

/******************************************************************************/

split()
{ /* Beginning of routine split */
  dir = lock_dir_retry_logic_no_wait(GLOBAL_P,exclusive);
  if ( dir is busy ) return;
  if ( dir->TOKEN == FALSE ) { /* missed the TOKEN */
    unlock_block(dir->number);
    return;
  }
  /* Now we have the real TOKEN directory. One thing that is guaranteed  as long as
  we hold the lock on the TOKEN directory is that our value of DIST_M will not change.
  Before the directory is split we move the TOKEN to allow further splits to
  be multi-threaded. */
  dir->TOKEN = false;      /* First clear the TOKEN flag */
  old_DIST_M = DIST_M;
  if ( dir->number == (DIST_M - 1) ) {
    DIST_M = DIST_M * 2;
    /* Update each processor's copy of DIST_M (including the local processor)*/
    GLOBAL_P = 0;
  } else {
    GLOBAL_P++;
  }
  newdir = locate_dir(GLOBAL_P);
  newdir->TOKEN = TRUE;  /* TOKEN is moved so the next split can operate */'

  if ( dir->DIR_M == (old_DIST_M * 2) ) { /* Bucket is already split */
    unlock_dir(dir->number);
    return;
  }

  highdir = lock_dir(dir->number+dir->DIR_M,exclusive);
  /* Now we rehash all of the chains in the original directory and distribute them between
  the original directory and the high directory (empty). This distribution is accomplished by
  using dir->DIR_M * 2 to compute the proper directory for the record. */
  dir->DIR_M = (dir->DIR_M) * 2 ;
  newdir->DIR_M = dir->DIR_M;
  rehash(dir,newdir);
  unlock_dir(dir);
  unlock_dir(highdir);
} /* End of routine split */

/******************************************************************************/
```

```
/* The merge code is very similar to the split code.  The primary difference is in the
direction of the TOKEN and the use of M / 2 in place of M * 2. */

merge()
{ /* Beginning of routine merge */
  dir = lock_dir_retry_logic_no_wait(GLOBAL_P,exclusive);
  if ( dir is busy ) return;
  if ( dir->TOKEN == FALSE ) { /* missed the TOKEN */
    unlock_block(dir->number);
    return;
  }
  /* Now we have the real TOKEN directory.  One thing that is guaranteed  as long as
  we hold the lock on the TOKEN directory is that our value of DIST_M will not change.
  Before the directory is split we move the TOKEN to allow further splits to
  be multi-threaded. */
  dir->TOKEN = false;    /* First clear the TOKEN flag */
  old_DIST_M = DIST_M;
  if ( dir->number == 0 ) {
    DIST_M = DIST_M  / 2;
    /* Update each processor's copy of DIST_M (including the local processor)*/
    GLOBAL_P = DIST_M-1;
  } else {
    GLOBAL_P--;
  }
  newdir = locate_dir(GLOBAL_P);
  newdir->TOKEN = TRUE;   /* TOKEN is moved so the next merge can operate */'

  if ( dir->DIR_M == (old_DIST_M / 2) ) { /* Bucket is already merged */
    unlock_dir(dir->number);
    return;
  }

  highdir = lock_dir(dir->number+dir->DIR_M,exclusive);
  /* Now we rehash the chains from both directories and move them into the lower directory.
  This is accomplished by re-hashing the records using dir->DIR_M / 2
   to compute the proper directory for the record. */
  dir->DIR_M = (dir->DIR_M) / 2 ;
  newdir->DIR_M = dir->DIR_M;
  rehash(dir,newdir);
  unlock_dir(dir);
  unlock_dir(highdir);
} /* End of routine merge */
```

BIBLIOGRAPHY

[BaFr86] Baru C., Frieder O., "Implementing Relational Database Operations in a Cube-Connected Multicomputer", Computing Research Laboratory Report #CRL-TR-10-86, University of Michigan, May 1986.

[BBN86] Butterfly Parallel Computers, BBN Advanced Computers, Inc., 1986

[BBPl87] Inside the Butterfly Plus, BBN Advanced Computers, Inc., 1987.

[BBRf86] Butterfly Parallel Processor Chrysalis Programmers Manual. Version 4.0, BBN Advanced Computers, Inc., 1986.

[BBTu88] Butterfly GP1000 Tutorial, BBN Advanced Computers, Inc., 1988.

[Ellis87] Ellis C., "Concurrency in Linear Hashing", *ACM Transactions on Database Systems*, Vol. 12, No. 2, June 1987, pp. 195-217.

[KiSn83] Kruskal C. P., Snir M., "The Performance of Multiple Interconnection Networks for Multiprocessors", *IEEE Transactions on Computers*, Vol. C-32 No. 12, December 1983 pp. 1091-1098.

[LeC86] Lehman T. J., Carey M. J., "Query Processing in Main Memory Database Management Systems", *ACM Proceedings of SIGMOD '86*, International Conference on Management of Data, Vol. 15, no. 2 (June 1986), 239-250.

[Lit80] Litwin, W., "Linear Hashing: A New tool for File and Table Addressing", Proc. 6th VLDB Conference, pp. 212-223, 1980.

[Mill89] Milliken W., Personal Communications at the Butterfly User's Group Meeting, April 12, 1989, University of Rochester, NY.

[Ncu85] NCUBE/Ten An Overview, NCUBE Corp., Tempe, Az, 1985

[PrDa88] "Multi Directory Hashing", Technical Report, Computer Science Department, Michigan State University, Sept 88.

[RoJa87] Rosenau T., Jajodia S., "Parallel Relational Operations on the Butterfly Parallel Processor: Projection Results", Technical Report, Naval Research Laboratory, July 1987.

[Scholl79] Scholl M., "Performance Analysis of New File Organizations Based on Dynamic Hash-coding", Res. Rep. 347, I.R.I.A. - Lamboria, (Mar 1979), 28.

[SeRoPr] Severance C., Rosenau T., Pramanik S., "A High Speed KDL-RAM File System for Parallel Computers", Technical Report, Michigan State University Computer Science Department, August 1988.

[Sev89] Severance, C., "Evaluation of Butterfly Memory Conflict Performance", Unpublished Manuscript, April 16, 1989.

[Thom86] Thomas R. H., "Behavior of the Butterfly Parallel Processor in the Presence of Memory Hot Spots", *Proceedings of the 1896 International Conference on Parallel Processing*, pages 46-50. IEEE Computer Society Press, 1986.

[Wolb89] Wolberg P., "Implementing a Parallel Projection on the Butterfly", Unpublished manuscript, Nov 1989.