Naval Research Laboratory
Washington, DC 20375-5000

NRL Report 9259

A High-Speed KDL-RAM File System for Parallel Computers

C. SEVERANCE,* T. J. ROSENAU, AND S. PRAMANIK*

Integrated Warfare Technology Branch
Information Technology Division

**Computer Science Department*
Michigan State University

June 22, 1990

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 22, 1990	

4. TITLE AND SUBTITLE

A High-Speed KDL-RAM File System for Parallel Computers

5. FUNDING NUMBERS

PE - 63223C
PR - 552354CO
WU - DN155-097

6. AUTHOR(S)

C. Severance,* T. J. Rosenau, and S. Pramanik*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Code 5576
Washington, DC 20375-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL Report 9259

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Strategic Defense Initiative Office
1717 K Street, N.W.
Washington, DC 20006

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

*Computer Science Department, Michigan State University
This work was supported in part by grants from the National Science Foundation and the Naval Research Laboratory.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Here we present the design, implementation, and performance of a main memory file system. The implementation is based on a two-stage abstract parallel processing model whose objective is to maximize throughput and minimize response time. To maximize throughput, lock structures, access structures, and shared variables are distributed among the shared memories. A new, hash-based approach for parallel accesses is used. The effect of lock conflicts are minimized by an optimistic locking protocol. Analytical models are developed for hot-spot memory, distributed-data accesses, and space-vs-time trade-offs for fast accesses to records. Based on the performance results of these models, a high-speed KDL-RAM File System has been implemented on the Butterfly PLUS Parallel Processor and its performance results are given. We show that the performance improvement of this system is considerably better than Bolt, Beranek, and Newman, Inc.'s (BBN's) Butterfly RAMFile System on the Butterfly PLUS Parallel Processor.

Keywords:

14. SUBJECT TERMS		15. NUMBER OF PAGES
Parallel processing	Butterfly PLUS parallel processing	20
Relational database management systems → Hashing		16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

CONTENTS

A HIGH-SPEED KDL-RAM FILE SYSTEM FOR PARALLEL COMPUTERS

1.0 INTRODUCTION

A multiprocessor, main-memory database can be used to provide a high-speed database service for real-time applications. In these real-time applications, high throughput and average and maximum response times are important. Here we present a Key-accessed, Dynamically reconfigurable, distributed-Lock, Random-Access main Memory (KDL-RAM) file system that provides high throughput and fast response time with a guaranteed upper bound on access time. The high throughput and fast accesses are achieved through the parallel processing of databases and hash-based accessing of records. In main-memory databases, key comparison costs are a significant part of the response time. Thus as records are inserted and deleted, the file is dynamically reorganized to guarantee an upper bound on the overflow chain length.

1.1 Two-Stage, Parallel-Processing Model

In a shared-memory environment, parallel accesses to shared data cause memory-access conflicts. Figure 1 shows a processor memory interconnection architecture where several processors may access the same memory module at the same time. This can be a serious performance bottleneck for a high-throughput system. To avoid this problem, the data are distributed among the memories. In the KDL-RAM File System, we use hash-based data distribution to achieve high concurrency and throughput. Here, a key is hashed into a memory module number and a location within that module. Figure 2 illustrates an abstract model for the parallel processing of databases to achieve high throughput and fast response time. H_1 and H_2 are the data distribution and data construction stages. Q_i represents the parallel-processing nodes. After the data are distributed by H_1 between the parallel nodes, H_2 constructs the access structure within the node for fast accesses to a single record. Figure 2 also shows the implementation of H_1 and H_2 for the KDL-RAM File System where H_1 has been decomposed into H_{11} and H_{12}, where H_{11} maps key values to directories and H_{12} maps directories to memory devices. Note that the number of directories can be larger than the number of devices to reduce the lock conflict on the directories.

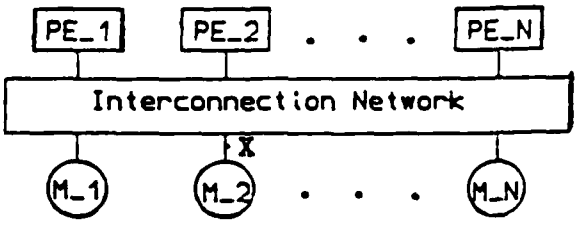

Fig. 1 — Parallel processing system model

Manuscript approved February 22, 1990.

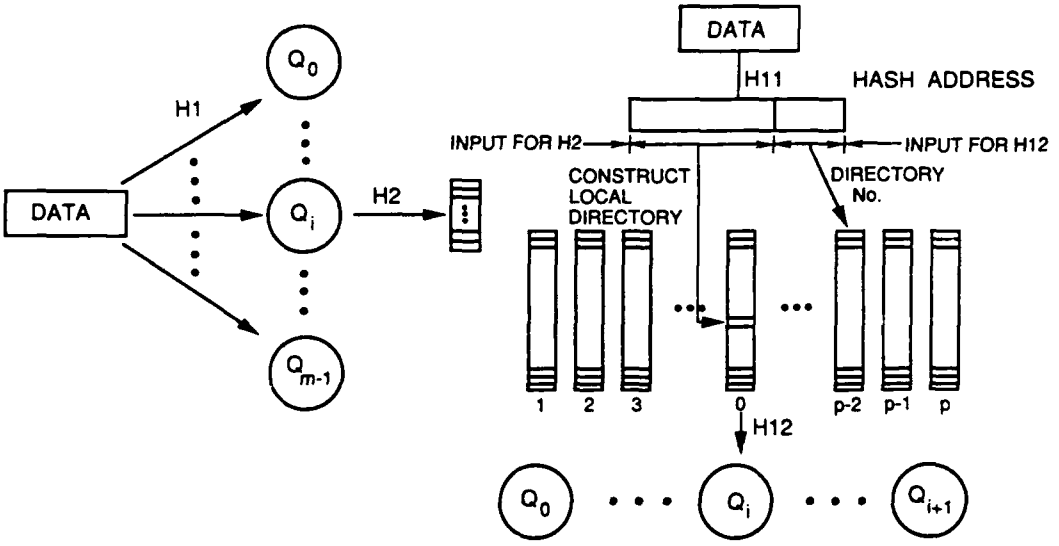

Fig. 2 — Abstract parallel-processing model of database sytems

There have been numerous hashing methods proposed for fast access to a single bucket [1–8], but none of them was designed to work in a parallel mode. The multidirectory hashing (MDH) methods proposed in Refs. 9 and 10 are designed for parallel operation and are based on the two-stage, abstract-parallel processing model. The KDL-RAM File System uses MDH for concurrent access to databases. The number of directories in a KDL-RAM file changes with the size of the file.

1.2 Previous Work

Much research has been done on the parallel processing of database management systems. The focus has been on database machines where parallelism in both I/O and CPU processing are exploited [11–20]. Commercial database machines have also been built that are used for special purpose applications [17,21]. However, it has been claimed that the performance of database machines would be limited by disk-to-memory transfer rates, and that unless the problem of the I/O bandwidth is tackled, the use of hundreds of query processors would not be justified [22,23]. Thus an efficient, general-purpose, transaction-oriented database machine is unlikely to be built by using existing I/O architectures. On the other hand, general-purpose, parallel-processing systems are becoming available commercially [24,25] and our objective is to develop databases for these systems. Little research has been done on the parallel processing of databases for general-purpose, parallel-processor systems [26,11]. The work in Ref. 11 involves relational databases for hypercube-type architectures. The KDL-RAM File System is designed for shared-memory parallel processors, such as the Butterfly PLUS Parallel Processor, for high-speed reads and writes.

1.3 Software/Hardware Platform for Parallel Processing

The KDL-RAM File System has been implemented on the Butterfly PLUS Parallel Processor, which is a Multiple Instruction, Multiple Data (MIMD) stream computer with shared memory. It can be configured with 1 to 256 processors, each of which is a Motorola 68020. Each node contains one processor capable of 2.5 million instructions per second (MIPS), a Motorola 68881 floating point coprocessor, a Motorola 68851 demand-paged, memory-management unit, and 4 Mbytes of main memory. A Butterfly

PLUS with 256 nodes can compute at 640 MIPS with a total memory of 1 Gbyte. The configuration available to us during this experiment was a 128-processor Butterfly PLUS with 512 Mbytes of total memory.

2.0 DATA DISTRIBUTION TO AVOID MEMORY-ACCESS BOTTLENECK

The first problem with high-speed reading and writing in shared-memory parallel computers is the memory-access bottleneck caused because several processors attempt to access the same memory location at the same time. The probability of this access conflict depends on such parameters as the read and/or write request rate to a memory unit and the memory access time. In general, data request rates are application dependent, and the probability of a memory access conflict is high for high-speed reading and writing to the same memory. For example, a serious memory-access bottleneck will occur for high-speed reading and writing when the lock table is centralized, because the lock duration is significant compared to the data request rate; and while a processor has a lock, other processors will do a test and set on the lock bit in a short loop creating a high data-access rate. A second memory-access bottleneck is the access structure—the hash directory and index of the database; every data request has to access this centralized data structure. A third source of memory access bottlenecks is the file-size, dependent parameters, because the hash function changes with the file size for dynamic files, and the variable containing the file size needs to be locked before every access.

Thus data-distribution algorithms have to distribute the lock table, access structure, and the file-size dependent parameters to avoid the memory-access bottleneck. Distribution algorithms for these structures and performance improvements resulting from these distribution techniques are given in Section 4. In Sections 2.1 and 2.2, we develop probabilistic models for network and memory-access conflict caused by parallel accesses.

2.1 Network Contention for Randomly Distributed Accesses

We have developed a probabilistic model for network contention in the Butterfly switch. We assume that a processing node generates access requests to every node's memory with the same probability. The network of the Butterfly machine with N number of processors consists of $\log_4 N$ stages with $N/4$ (4×4) switches at each stage. When several remote memory-access requests in the same switch are directed to the same output line, only one request is forwarded and all others are rejected and tried again from the source processors.

Let r_n be the probability that there is some remote memory access request on any particular output line of the n^{th} stage. Then the recurrence equation, $r_{n+1} = 1 - (1 - r_n/k)^k$, can be formulated, where r_0 is the rate of remote memory access request-per-network cycle time for each processor and the network uses $k \times k$ switches [27]. The number of remote memory access retries because of network contention (per network cycle time) is $r_0(r_0/r_n - 1)$. The r_0/r_n term comes from the geometric behavior of access requests. Then $100r_0(r_0/r_n - 1)$ is the percentage of network contention overhead. Figure 3 shows this for various system configurations and remote memory request rates. The number of memory modules is assumed to be same as the number of processors.

The presence of alternate paths (optional in the Butterfly switch) to reduce network contention is not considered. Figure 3 gives a rather conservative result showing that the increase in network contention overhead is small with an increase in the number of processors. Even when r_0 is large, the probability of

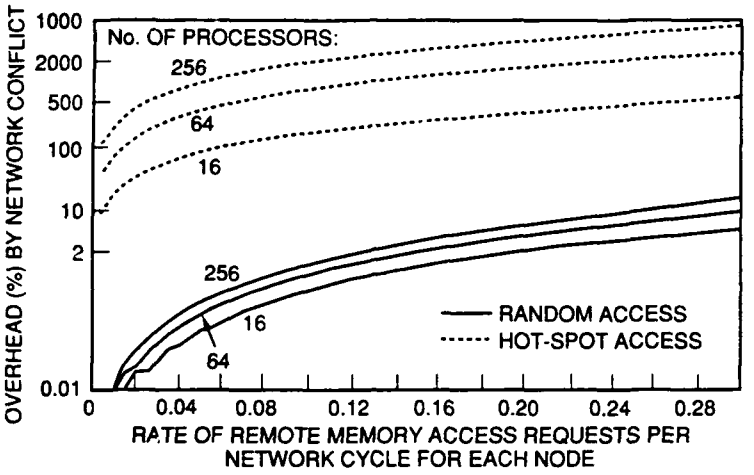

Fig. 3 — Percentage of increase in retries per network cycle

network contention overhead does not increase significantly with the number of processors. In Section 2.2, we see that data concentration does create serious access retry overhead. Thus data distribution helps in avoiding network access bottlenecks.

2.2 Access Conflict for Hot-Spot Memory

Let hot-spot memory (a single-memory location) be present and M processors compete to access this memory location. It is not difficult to see that the probability of each request being successful (or r_n) is $1/r_0M$, where r_0 is the hot-spot, access-request-rate-per-network cycle time. For each processor, the average number of hot-spot-access requests until success is r_0M, with the additional number of hot-spot access requests or network-contention overhead equaling $r_0M - 1$. Figure 3 shows the network contention overhead for hot-spot memory. We see that the number of retries per data request increases from 400% to 8000% when the number of processors increases from 16 to 256, and $r_0 = 0.3$. There is no problem, however, if r_0 is very small—say 0.01. Note that in the Butterfly machine, the r_0 value is generally very small because the programs are stored in local memory. For lock requests, on the other hand, the r_0 value is high because the test and set instruction is executed in a short loop.

We have also done performance studies on parallel memory accesses for the Butterfly machine, as shown in Figure 4. The performance is worst for hot-spot memory accesses but improves considerably for random accesses, which do not perform well when a large number of processors are used.

3.0 DATA CONSTRUCTION STRATEGIES

The objective of a data construction strategy is to achieve fast access to data records with a guaranteed upper bound on access time. A hash-based access with a bounded overflow chain length is used to achieve this. Though the response time depends on such parameters as lock contention, memory-access conflict, and hash-function computations, the number of key comparisons also contributes significantly to the response time in main-memory databases. Thus, when a hash collision occurs, we expand the directory instead of the overflow chain. The cost of directory expansion is minimized by an extended linear hashing scheme that is described in Section 4. In Section 3.1, we analyze storage use for directories when the chain length is at most one. This is the lower bound on access time. Larger directory sizes are needed to guarantee

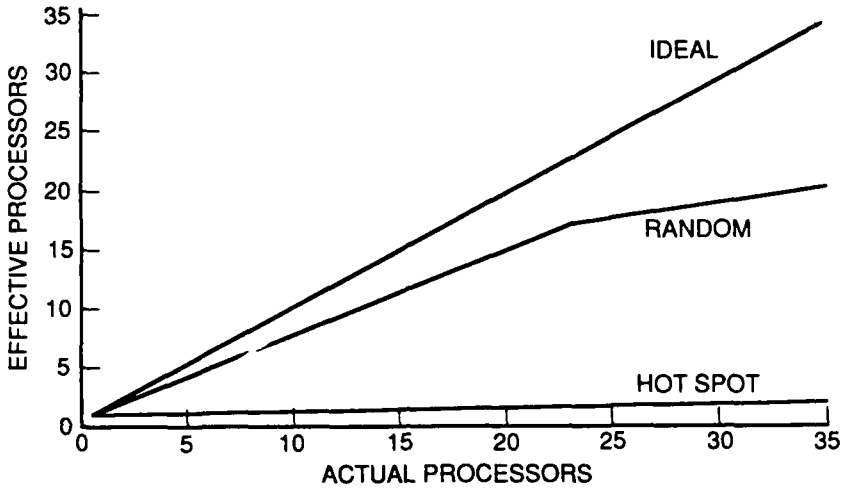

Fig. 4 — Performance of parallel-memory accesses in the Butterfly machine

this lower bound, because more hash-address bits are required to avoid collisions. However, the storage penalty is minimized by creating multiple hash directories.

3.1 Storage Use to Guarantee Lower Bound on Access Time

The average hash directory size for m records when no collision is allowed and a maximum of b bits are used for the hash address is given by

$$AVG(b, m) = \sum_{i=0}^{b} (2^i \times Prob(2^i, m)),$$

where $Prob(2^i, m)$ is the probability that the directory size is 2^i when m records are inserted without collision. We can express $Prob(2^n, m)$ as

$$Prob(2^n, m) = Prob(size \leq 2^n, m) - Prob(size \leq 2^{n-1}, m),$$

where $Prob(size < k, m)$ is the probability that the directory size is less than k when m records are inserted and where

$$Prob(size \leq 2^{-1}, m) = 0 \quad \text{For all } m.$$

$$Prob(size \leq 2^0, m) = 0 \quad \text{For any } m > 1.$$

$Prob(size < 2^n, m)$ is given by

$$Prob(size \leq 2^n, m) = \prod_{k=1}^{m-1} \frac{2^b - k2^{(b-n)}}{2^b - k}.$$

Here we have assumed that the directory sizes are powers of two, and they range between 2^0 and 2^0. The value of b is usually very large to guarantee enough hash address bits to avoid collision. Thus the above formula gives the average directory size with no collision when the value of b is very large. It is difficult to compute the recurrence relation for a large b; however we will use the result for small value of b to derive

the important theorem described in Section 3.2. Figure 5 gives the average directory size when $b = 9$. Nine bits will be able to resolve conflict for only a small number of records. The linear portion of the curve will be used for the average directory size with no conflict. Figure 6 shows the corresponding storage overhead ($f(m)/m$) obtained from Fig. 5, where $f(m)$ is the average directory size for m records and also shows that a file with 17 records will have the worst-case storage use.

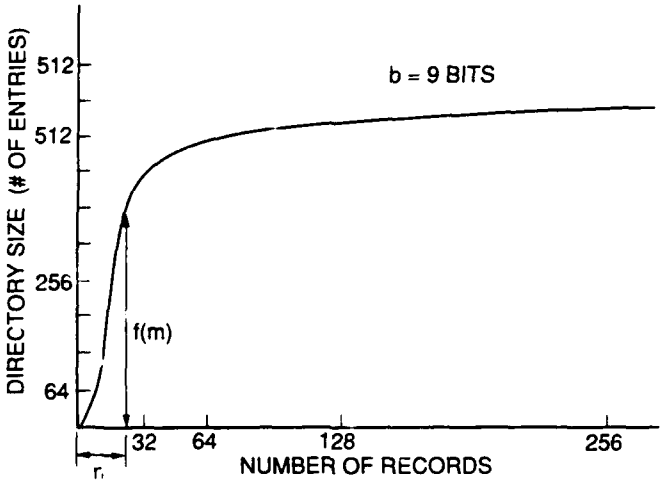

Fig. 5 — Average directory size for one access-per-record

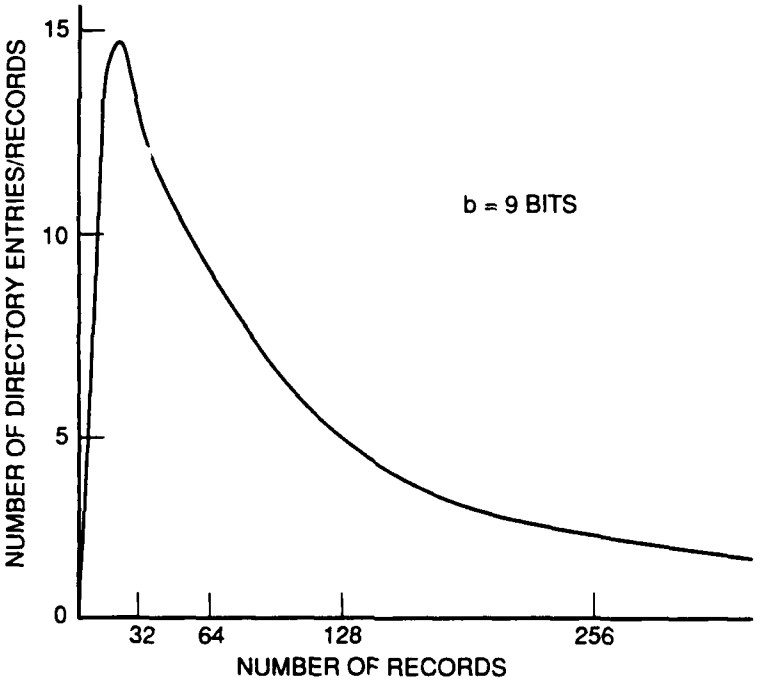

Fig. 6 — Storage overhead for one access-per-record

3.2 Improved Storage Use by Multidirectory Hashing

We see from Fig. 5 that for any number of records m, where $m \leq$ the worst case storage use, it is more economical to store the records in two independent directory structures than in one. The following theorem proves this claim.

THEOREM. The average size of any directory with m records, where $m \leq$ the worst-case point, is bigger than the sum of the average sizes of two directories with m_1 and m_2 records respectively and where $m = m_1 + m_2$ for a given b and $f(1) = 1$.

Proof: Let $f(i)$ be the average directory size for a file with i records. Then the following expression calculates the total directory size for two directories with m_1 and m_2 records:

$$total\_dir\_size\_2 = f(m1) + f(m2) = f(m1) + f(m - m1)$$

$$= f(m) + \{f(m1) - \{f(m) - f(m - m1)\}\}.$$

As Fig. 5 shows, the straight line between the origin and the point $(m,f(m))$ is always above $f(i)$ for any i between 0 and m. Hence, the minimum value for $f(m) - f(m - m_1)$ is $< (m1)$, which makes the whole term $\{f(m_1) - \{f(m) - f(m - m_1)\}\}$ less than zero. Thus,

$$total\_dir\_size\_2 = f(m1) + f(m2) < f(m).$$

By applying the Theorem recursively, we get the optimal set of directories with one record each; however, this is not practical, because the theorem ignores the overhead of keeping a set of pointers to locate the directories.

We constructed multiple directories based on the hash-based data distribution of Fig. 2. Extendable and linear hashings were used for constructing each directory. Figure 7 gives the directory size for multidirectory hashing when one record-per-directory entry at most is allowed; the experiment has been performed by inserting 5000 unique key values to the database. The y-axis of Fig. 7 represents the total size of all the directories and shows that the total directory size decreases considerably with an increasing number of directories.

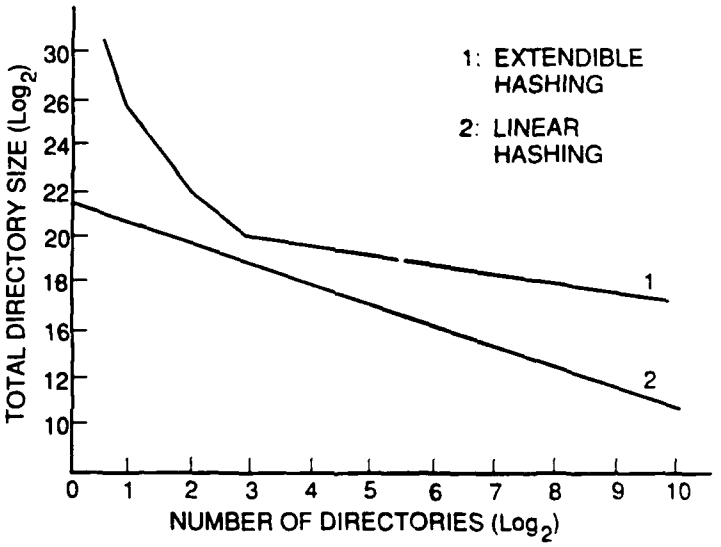

Fig. 7 — Total directory size in multidirectory hashing

3.3 Bounding Average Access Time

We have also analyzed storage use when chain length is greater than one. In this approach, we bound the average chain length to some specified value. For example, our experiment shows that by allowing an

average chain length of 1.01 in a single directory, we reduce the directory size over the case when the chain length is 1.0. The storage penalty is minimized in a single directory case when some collisions are allowed. However, bounding the access time to one is also a practical proposition in a parallel-processing environment because multiple directories, which are useful for concurrent accesses, reduce the total directory size.

4.0 AN IMPLEMENTATION OF THE KDL-RAM FILE SYSTEM ON A BUTTERFLY PLUS MACHINE

We have implemented a KDL-RAM File System on the Butterfly PLUS Parallel Processor based on the two-stage abstract model presented in Section 1. Now we highlight the design, the details of which can be found in Ref. 28 and which include information on how to use the KDL-RAM File System.

Figure 8 shows the primary data structure of the KDL-RAM File System. Here the file is accessed through multiple hash directories that are distributed among the various memories. A directory map table (DMT) is used to locate the directories. Since every data request in the KDL-RAM File System has to access the DMT, the DMT can be a source of hot-spot problems. To avoid these problems, the DMT is copied into the memory of every processor. The data consistency of multiple copies is guaranteed by maintaining a master copy of the DMT. When a processor accesses an entry in its local DMT and finds it empty, only then does the master DMT have to be accessed—otherwise all DMT accesses are local. Note that in the Butterfly machine, remote memory-access times are about ten times longer than local memory accesses. For each processor, the master copy will need to be accessed, at the most, as many times as there are number of directories. Even though the master DMT is shared, it need not be locked, because a DMT entry is updated only when the corresponding directory block is split, and a processor has to lock the directory block before it can be split.

The hash directories are dynamically created as the file size grows. When the average chain length in a directory exceeds the upper bound specified by the user, the directory is split into two by using a scheme discussed in Section 4.1. The objective is to keep the average chain length very small even though a smaller chain length requires more directories. This classical space vs time trade off is analyzed in Section 3. For example, to achieve the lower bound on access time, a chain length of one, at most, is required. To achieve this, many sparse directories may be needed.

4.1 Directory Growth by a Concurrent Linear-Hashing Scheme

The number of directories increases in the KDL-RAM File System as the file size grows. Growth is achieved by storing each directory in a fixed-size bucket of a linear hashed file. The buckets are distributed among the various memories to enhance concurrency. As in linear hashing [2], two variables P and M are used for adding buckets to a file; P is used to point to the bucket to be split next, and M is used to compute the hash address. Initially, M number of buckets, numbered 0 ... M - 1, are created, and P points to the first bucket. The location of a record within the bucket is found by using part of the original hash address bits as an offset in the directory in that bucket. Records for each entry of the directory are chained.

4.1.1 Bounding Chain Lengths Under High-Speed Concurrent Record Insertions

The insertion rate of records increases significantly with the number of nodes. As a result, the rate of bucket splits falls behind the rate of record insertions when the number of nodes is large. This condition can

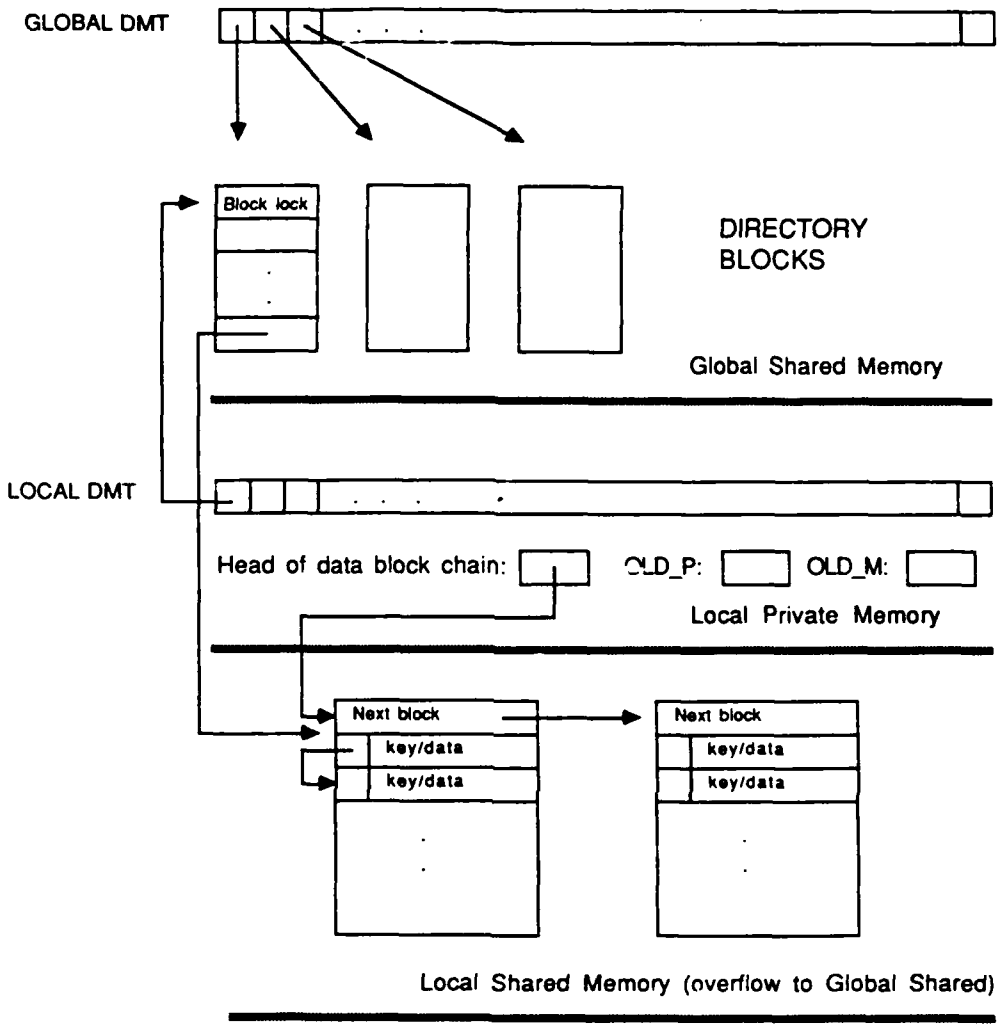

Fig. 8 — Primary data structure of the KDL-RAM File System

result in a long, average-chain length; to avoid this, we split a bucket as soon as it overflows without having P point to it. This requires concurrent splitting of the buckets. If a bucket is split, and it is not pointed by P, then it is marked as such; this information is needed to locate a record. When too many buckets are marked split, the KDL-RAM File System forces P to propagate all the way to the end to guarantee the desired bound on the average chain length.

4.2 Hot Spot for P, M Locks

P and M are the potential sources for hot-spot problems, because their values are needed by every data access. Therefore, they need to be kept locked for a significant amount of time until an insertion is complete. However, the values of P and M change only periodically. Thus we will use an optimistic locking approach in accessing P and M; a retry logic is employed to achieve this. If the values of P and M are changed during the time they are used, the operations are retried. In Section 5, we present the performance improvement of this optimistic approach as used in the KDL-RAM File System over straight-forward (P, M) locks.

5. LOCK-IMPLEMENTATION STRATEGIES IN THE KDL-RAM FILE SYSTEM

In a multiprocessor, real-time environment, database reorganization must occur with minimal impact on other database activities. For this reason, a single, file-wide lock for reorganization is not acceptable. A locking scheme is necessary to ensure overall database consistency.

5.1 Optimistic Locking Protocol for P and M Accesses

A fundamental race condition, which can cause database inconsistency, occurs when one processor reorganizes a directory block and another processor uses P and M to calculate the block for a record to be inserted. Once the block is calculated, the first processor finishes the reorganization, changing P and/or M in such a way that the record being inserted will end up in the incorrect block. A possible solution to this problem would be to lock all accesses to P and M during the block split. This technique has two pitfalls: (1) it affects the entire database causing nonaffected access blocks to be delayed needlessly, and (2) memory conflict for the lock could cause hot-spot overhead.

A more efficient solution can be implemented by an optimistic locking protocol where we assume that the probability of a race condition is low, causing a record to be hashed into the wrong block. The solution is to calculate the directory block without locking P and M, locking the block, then recalculating the directory block and comparing the results of the second computation with the actual block locked. If the results match, the proper block was locked. If there is a mismatch, the block is unlocked and the entire lock operation is retried. This solution has the additional cost of a block unlock and retry for the new block lock, but it has the advantage that only the block that is reorganized is affected during the reorganization phase. There is no delay to unaffected portions of the database. In measured experiments of continuous insertion loads with up to 50 processors, the retry logic was used less than once in 5000 inserts on the average.

Another race condition has to do with the simultaneous updating of P and M. Normally, only P is updated when a block is split and when the hardware-provided atomic operation ensures the consistency of P. When P reaches M - 1 and the M - 1 block splits, M is doubled and P is set to zero. As these are two separate operations, a potential race condition exists where a processor can get completely incorrect values for P and M. A simple solution to this problem is to lock P and M together when updating M and then check the lock on each access to P or M. A problem arises because the lock would be locked and unlocked twice for each database read or write causing at least six memory accesses (two each for test, set, and write). This lock causes a hot-spot problem that has a significant impact on performance as the number of processors is increased.

Once again, an optimistic approach allows most lock accesses to be eliminated by using retry logic. The M updates are still locked. The code to accomplish this is as follows:

```
newm := M * 2;
LOCK(pmlock);
M := 0;
P := 0;
M := newm;
UNLOCK(pmlock);
```

M is set to zero before setting P to zero so that processes can tell if the P, M combination is invalid. During the entire time P and M are invalid, M is set to zero.

When each process starts, it makes a consistent copy of P and M using the lock. These values are stored in the local memory and are used to determine if M has changed. When a process accesses P and M, first the access is performed without the lock. If M is not zero and has not changed since the last access, then P and M are consistent. If M is zero or has changed, then P and M are inconsistent, and the process must use the lock to access the values to ensure consistent values, as shown below:

```
loc_p := P;
loc_m := M;
IF ( loc_m = 0 or loc_m < > old_m ) THEN BEGIN
  LOCK(pmlock);
  loc_m := M;
  loc_p := P;
  UNLOCK(pmlock);
END;

old_m := loc_m; /* save for next access */
```

In this way, the lock is only accessed once for each process when M is updated thereby eliminating over 99% of the lock accesses and solving the severe hot-spot problem.

A comparison of results with and without retry logic is given in Fig. 9, which shows the effect of hot-spot memory contention on the overall performance of an application and a significant amount of performance improvement as a result of the optimistic locking protocol.

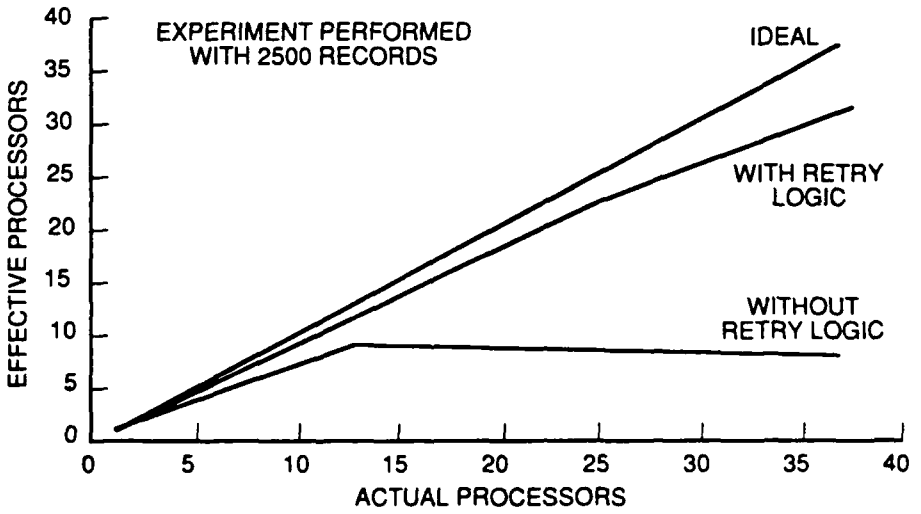

Fig. 9 — Cost of P, M lock with and without retry logic

6.0 PERFORMANCE OF THE KDL-RAM FILE SYSTEM

Figure 10 compares the performance of the KDL-RAM File and the Butterfly RAMFile Systems for inserting 100,000 records in rapid succession by the processors. (The Appendix gives a brief description of the Butterfly RAMFile System.) In this experiment, each processor writes almost an equal number of records.

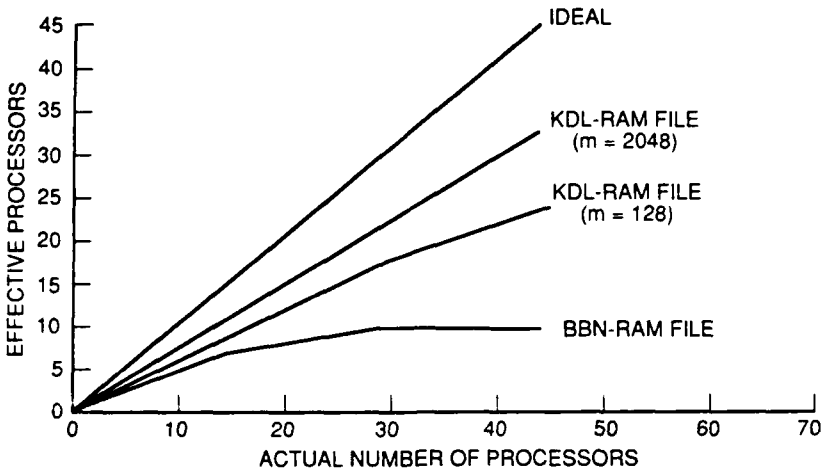

Fig. 10 — Performance comparison of the KDL-RAM File and Butterfly RAMFile Systems

Figure 11 shows that the performance improvement of the KDL-RAM File System is considerably better than the existing Butterfly RAMFile System, which levels off after only 15 processors, while the KDL-RAM File System has about 73% effective processors when 50 real processors are used. The m_0 value corresponds to the number of directory blocks initially allocated. To make this performance improvement fair, we used various parameter adjustments during the experiment. For example, we made sure for large databases that all the records reside within a single node's memory when a single processor is used. Figure 11 also compares the average record insertion time for the KDL-RAM File and Butterfly RAMFile Systems. For the KDL-RAM File System, the insertion time is almost constant.

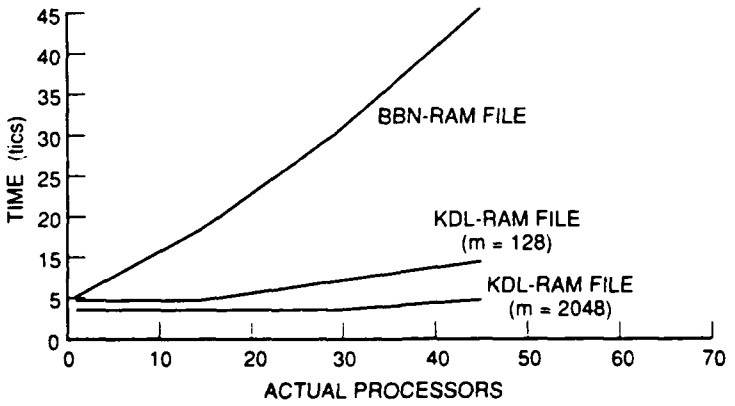

Fig. 11 — Average insertion time for a record

We analyzed the performance of the KDL-RAM File System for parallel reads, as shown in Fig. 12. The performance is optimal for up to 80 processors after which the performance begins to degrade because of conflict between the effect of memory and network access.

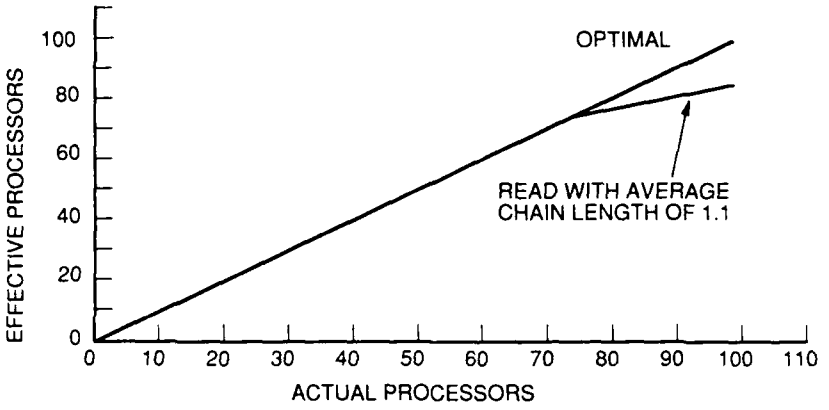

Fig. 12 — Performance of parallel reads in the KDL-RAM File System

7.0 CONCLUSIONS

The design, implementation, and performance issues of parallel, main-memory databases are presented. Performance improvements for up to 50 processors are given. We are currently investigating various strategies to make these performance curves more linear. Performance analysis for a larger number of processors (~ 100) is under way. We are also working to extend this design for a disk-based virtual memory system, where the proposed main memory system may be viewed as processing a part of the database in a large main memory buffer. To accomplish this, various fundamental issues (such as parallel I/O, working set models, and processing time vs disk speed) need to be investigated. We show that the KDL-RAM File System is an appropriate data-access structure for implementing parallel relational database systems. We are now implementing database functions such as projection, restriction, and hash-based joins by using the KDL-RAM File System.

8.0 REFERENCES

1. R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files," ACM *Transactions on Database System* **4**(3), 315-344 (1979).

2. W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing," Proceedings of the 6th Very Large Database (VLDB) Conference, Montreal, Quebec, 1980, pp 212-223.

3. P.A. Larson, "Dynamic Hashing," *BIT* **18**(2), 184-201 (1978).

4. G. Martin, "Spiral Storage, Incrementally Augmentable Hash Addressed Storage," Theory of Computing Report, University of Warwick, Coventry, England, Mar. 1979.

5. D.B. Lomet, "Bounded Index Exponential Hashing," ACM *Transaction on Database Systems* **8**(1), 136-165 (1983).

6. S.P. Ghosh, *Database Organization for Data Management*, 2nd ed. (Academic Press, Inc., San Diego, 1986).

7. R. Sprugnoli, "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets," *Communications of the ACM* **20**(11), 841-850 (1977).

8. G. Wiederhold, *File Organization for Database Design* (McGraw-Hill, New York, 1987).

9. S. Pramanik and H. Davies, "Multi-Directory Hashing," Technical Report, Computer Science Department, Michigan State University, Sept. 1988.

10. S. Pramanik and M. Kim, "Generalized Parallel Processing Model for Database Systems," Proceedings of the International Conference on Parallel Processing, University Park, PA., Aug. 1988, pp. 76–83.

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

12. D. Bitton, H. Boral, D. DeWitt, and W. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," ACM *Transactions on Database Systems* 8(3) (1983).

13. E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," ACM *Transactions on Database Systems* **4**(1) (1979).

14. H. Garcia-Molina, R. Lipton, and J. Valdes, "A Massive Memory Machine," *IEEE Trans. Comput.* **C-33**(5) (1984).

15. P. Hawthorne and D. DeWitt, "A Performance Evaluation of Database Machine Architectures," International Conference of Very Large Databases (VLDB), Cannes, France, Sept. 1981, pp. 199-213.

16. S. Su, L. Nguyen, A. Eman, and G. Lipovski, "The Architectural Features and Implementation Techniques of the Multicell CASSM," *IEEE Trans. Comput.* **C-28**(6), 430–445 (1979).

17. "IDM Product Description," Britton-Lee, Inc., Los Gatos, CA, 1986.

18. S. Pramanik and F. Fotouhi, "Index Database Machines," *Computer J.* **29**(5) (1986).

19. S. Pramanik and M. Kim, "HCB_tree: A B_tree Structure for Parallel Processing," *Information Processing Lett.* **29**(4), 213-220 (1988).

20. S. Pramanik, "Performance Analysis of a Database Filter Search Hardware," *IEEE Trans. Comput.* (Dec. 1986).

21. Teradata Corporation, description of architecture appeared in *Computer Decision* (1986).

22. H. Boral and D. DeWitt, "Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines," *Database Machines*, (Springer-Verlag, 1983).

23. R. Agrawal and D. DeWitt, "Whither Hundreds of Processors in a Database Machine," Proceedings of the International Workshop on High-Level Language Computer Architecture, 1984.

24. Butterfly Parallel Processor Overview, BBN Advanced Computers Inc. (1986).

25. "NCUBE/Ten, An Overview," NCUBE Corp., Tempe, AZ, 1985.

26. T. Rosenau and S. Jajodia, "Basic Database Operations on the Butterfly Parallel Processor: Experiment Results," NRL Memorandum Report 6173, Mar. 1988.

27. C. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Trans. Comput.* **C-32**(12), 1091-1098 (1983).

28. C. Severance and S. Pramanik, "KDL-RAM File System Users' Manual," Computer Science Department, Michigan State University, Oct. 1988.

Appendix A
THE BUTTERFLY RAMFILE SYSTEM

A multiprocessor that allocates individual memory segments with each processor has the potential to store complete files in main memory by scattering them across the distinct memories of all the processors. The Butterfly computer allows you to load complete files into main memory by using the Butterfly RAMFile System. File size should not be a problem, since the available main memory can approach 1 Gbyte (with 256 processors each with 4 Mbytes of memory). Using computers with large amounts of main memory will have a beneficial effect on database operations by allowing several files to be kept resident in memory while different queries progress. Instead of different portions of a file in and out from a external disk, you can access the file immediately from main memory, thereby saving on disk accesses. If there is enough main memory, you can simulate a disk drive on each processor. This method will store the file in main memory and will distribute the file across the different processor's memory.

The RAMFile system is a utility that allows files to be loaded into main memory and accessed like a UNIX* file. The size of the file is constrained by the amount of main memory available, which can approach 1 Gbyte, depending on the configuration. This is an upper limit, because the operating system and other application programs also will have to be resident in the same memory. To load a file into and out of the Butterfly's main memory, the application program must call a routine that will transfer the file from or to the host computer over the network by using a streams server. This can be a slow, serial process depending on the size of the file being loaded. The method appears to be a temporary solution until the Butterfly can handle disk drives attached directly to it. The file will be distributed across the memory of all available processors in a round-robin fashion, allocating blocks of the file to the next available processor with enough memory to hold that size block. It will continue "dealing" equal-size segments of the file until the complete file has been dispersed. Each segment of a file can range in size from 256 bytes to 64 kbytes in increments of powers of 2.

Access methods are similar to UNIX system calls except that the prefix "RF_" is inserted in front of all system calls (for example, RF_create, RF_open, RF_seek, RF_read, RF_write, RF_close). These calls are used like their UNIX counterparts except for possible parameter differences. These primitives allow individual processes to access the RAMFile without worrying about consistency and other related problems. By using the RF_seek command (similar to the lseek command), you have the ability to address specific locations within a file. Upon opening a file with RF_open, all processes that must access the file can do so with all locking and mapping hidden from the user.

One problem preventing maximum parallelization is "hot spots" (or contention). These cause locking if accesses are made to the same segment of a RAMFile and if attempts are made to obtain data from the same remote processor at the same time. The RAMFile System causes contention by locking a file on a segment basis. This means that if two processors try to access two different (or the same) memory locations that happen to lie within the same segment, only one processor will be able to proceed, while the other will have to wait until the first unlocks the particular segment in the segment lock table. If two processors are

*UNIX is a trademark of Bell Laboratories.

trying to obtain data from the same remote processor through the Butterfly switch at the same time, one will have to wait for the first data transmission to complete, causing a similar hardware-related contention problem.