Transaction Support in Read Optimized and Write Optimized File Systems†

Margo Seltzer
Michael Stonebraker

Department of Electrical Engineering and Computer Science
University of California Berkeley, CA 94720

Abstract

This paper provides a comparative analysis of five implementations of transaction support. The first of the methods is the traditional approach of implementing transaction processing within a data manager on top of a read optimized file system. The second also assumes a traditional file system but embeds transaction support inside the file system. The third model considers a traditional data manager on top of a write optimized file system. The last two models both embed transaction support inside a write optimized file system, each using a different logging mechanism.

Our results show that in a transaction processing environment, a write optimized file system often yields better performance than one optimized for reads. In addition, we show that file system embedded transaction managers can perform as well as data managers when transaction throughput is limited by I/O bandwidth. Finally, even when the CPU is the critical resource, the difference in performance between a data manager and an embedded system is much smaller than previous work has shown.

1. Introduction

During the past decade several attempts have been made to provide transaction support as part of the operating system. Embedded support provides concurrency control and crash recovery to all applications rather than just to the clients of the data manager, and there is a single paradigm for application recovery and a single implementation of this paradigm. Such systems are described in [WALK83], [MUEL83], [PU86], and [MITC82]. All of these systems assume a traditional, or read optimized, file system and have met with much skepticism. The shortcomings inherent to these systems are discussed in [STON81], [TRAI82], and [STON85].

Camelot's distributed transaction processing system [SPE88A] provides a set of Mach processes which provide support for nested transaction management, locking, recoverable storage allocation, and system configuration. Atomic transactions may be implemented by means of the recoverable storage, but requests to read and write such storage are not locked automatically. Thus a data server must make requests of the disk manager to lock these regions[SPE88B]. In this way, we see that transaction support under Camelot may be viewed as a hybrid between a data manager and an embedded operating system transaction manager. It is similar to a data manager in that a user process (data server) is required to coordinate locking. On the other hand, it is also similar to an embedded transaction manager in that generic locking and transaction capabilities are provided.

Kumar, [KUM87], shows that an operating system embedded transaction manager provides substantially worse performance than the traditional data manager. He cites the lack of semantic information, the system call locking overhead, and the size of the log as primary causes for a 30% difference in performance between the two systems. In [KUM89], by introducing hardware assisted locking and better locking protocols, he finds that the difference in performance may be reduced to 7-10%.

By changing Kumar's simulation model in two fundamental ways, the locking granularity and the buffer management, we find that even without special hardware support, embedded transaction managers can equal the performance of data manager transaction support. Specifically, in disk bound configurations, performance is dominated by the cost of reading random data blocks. Since both the data manager and embedded systems perform the same number of data reads, performance is virtually the same in both models. In CPU bound configurations, the performance difference between operating system embedded support and a data manager

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

† This research was sponsored by the National Science Foundation under contract MIP 8715235.

may be expressed simply as a function of the system call overhead and in our simulations is less than 20%, even without the locking techniques found in [KUMAR89]. Under high contention, using subpage locking or variable page sizes, the embedded models can come within 5% of the data manager models. By using special locking techniques, the embedded systems can actually provide better performance than the data manager. Finally, under either model (data manager or embedded), a write optimized file system outperforms a read optimized one.

The purpose of this study is twofold. First, we wish to characterize the performance of operating system embedded transaction support. If such embedded systems perform as well as their database counterparts, applications other than the data manager could reliably access shared data. Secondly, we seek to understand the tradeoffs between using a read optimized file system and a write optimized one. The simulations described in this paper isolate each of the critical resources and stress all five models in each dimension, enabling a characterization of the performance of each model across a wide range of configurations.

In the next section, our simulation model is presented. This is followed by a description of a write optimized file system and the other models being examined. Then the simulation results of our study are presented.

2. The Simulation Model

We used a stochastically generated workload to compare the different models of transaction support. The database was defined to consist of a single data file with a variable number of indices stored as B-trees. Its size and fill factor (the fraction of each page containing valid data) may be varied through simulation parameters.

A transaction is defined to be a sequence of retrieve, update, insert, and delete operations. Each retrieve and update operation affects a single data page and a search path through a single index. A search path consists of an access to one page in each level of the B-tree, culminating with a leaf page. An insert or delete operation affects a single data page and a search path through each index, since it is presumed that a key must be inserted/deleted into/out of each index.

At any time during a simulation there are at most M active transactions where M defines the degree of multiprogramming. At initialization, M transactions are created, and each time a transaction commits or aborts, a new transaction is created. A number of operations, O, uniformly distributed over $(l - .25l, l + .25l)$, where l is the average transaction length, is generated. A second parameter, F determines what percent of the O operations modify the database (as opposed to performing only reads). Finally, a third parameter f identifies what percentage of the modify operations are inserts or deletes (as opposed to updates). Thus, a transaction may be defined as:

O total operations composed of:

$(1-F)O$	retrieves
fFO	inserts/deletes
$(1-f)FO$	updates

Each operation of a transaction is processed in the following manner. A data page is selected from a distribution described by two parameters d and a. The parameter d indicates what percent of the database gets a percent of the accesses. For example, d=20 and a=80 means that 80% of the accesses go to 20% of the database. Once a data page is selected, it is locked, read from disk or the buffer pool, and left locked until transaction commit time. To simulate index traversal, using the same distribution as was used for the data file, one page is selected from each level of the B-tree. These pages are locked, read, and unlocked either at completion of the operation (for data manager models) or at transaction commit time (for embedded models). As soon as one operation completes, the next operation begins. When all the operations have completed, a synchronous write forces the log to disk.

Processor speed may be varied by simulation parameters, but the number of instructions required to perform each operation is fixed. These numbers are summarized in table 1. The instruction count for locking includes both the lock and unlock actions. In the embedded models, we assume that a system call is required to obtain a lock, so the actual cost of a lock is a function of the number of instructions for both a system call and a lock (we assume that all unlocking may be performed by a single system call at transaction commit time). Periodically, the deadlock detector runs aborting transactions which have exceeded the parametrized timeout interval. Another parameter defines how frequently a checkpoint is taken. At checkpoint time, all dirty pages are forced to disk and creation of new transactions is inhibited until all active transactions have committed.

The total database size is derived from the dbsize, pagesize, and fillfactor parameters respectively. Dbsize defines the size (in megabytes) of the data file. Using the fillfactor parameter, which defines how much data is on each page, we determine the number of records in the database. Then, using the number of records, the fillfactor, the pagesize, and the size of a key (16 bytes), we determine how many index pages are required, using the

operation	number of instructions
lock	1000
syscall	500
retrieve	7000
update	12000
insert/delete	18000

Table 1: CPU cost of each operation.

formulas below.

$$L = \frac{R * K}{F * P} \quad \text{and} \quad L_i = \frac{L_{i+1} * K}{F * P}$$

where
L is the number of leaf pages
L_i is the number of B-tree pages at level i
R is the number of records in the data file
K is the key size
F is the fillfactor
P is the pagesize

Once the size of each index has been calculated by summing the L_i above, we multiply by the number of indices and add the data file size to yield the total database size. Finally, we define the buffer pool size to be 10% of this total database size. The buffer pool uses an LRU replacement algorithm and flushes dirty blocks to disk asynchronously. In both [KUMAR87] and [KUMAR89], the buffer pool is sized in terms of a number of pages. This penalizes simulations with a smaller page size by providing them less main memory. By keeping the amount of main memory constant, we find that reducing the page size can improve performance by more than 25% in high contention environments. Table 2 summarizes the simulation parameters available and their default values.

Statistical Parameters		
parameter	description	default
runlen	Transactions per run	10000
nruns	Runs per data point	5
Workload Characteristics		
parameter	description	default
l	Avg ops per transaction	16
F	% update operations	.25
f	% insert/delete	.50
d/a	Request distribution	50/50
I	Number of indices	5
dbsize	Mbytes in the data file	1024 (1G)
bufsize	Buffer pool size	10% of db
fillfactor	Valid fraction of page	.70
System Parameters		
parameter	description	default
cpu_speed	Processor speed (in MIPS)	10
disks	Number of disks	10
users	Degree multiprogramming	20
pagesize	Page size (in bytes)	4096
spagesize	Subpage size	128 bytes
deadlock	Deadlock detector interval	5 sec
chkpt	Checkpoint interval	5 min

Table 2: Simulation Parameters

3. Transaction Processing Models

This analysis considers five models of transaction processing. The first is a traditional data manager on a read optimized file system. The second puts the same data manager on a write optimized file system. The third supports embedded transactions in a read optimized file system. The fourth embeds transactions in a write optimized file system, and the last also embeds transactions in a write optimized file system, but takes advantage of the file system's "no-overwrite" policy to reduce the size of the log.

3.1. The Data Manager Model

In the data manager model we assume detailed knowledge of the structure of the database. For example, logging may be performed at a logical, rather than a physical level, allowing the logging of only the record being modified instead of an entire page. In this model, all records are assumed to be self contained, that is, their index values are stored in the records themselves, and index changes need not be logged explicitly. Using special concurrency control protocols facilitating high degrees of parallelism [BAYER77], the data manager needs only hold index locks during the physical manipulation of the index page (on the order of a few thousand instructions), providing superior performance in environments with high lock contention.

The sequence of events for accessing a random record in the database is as follows. First, a keyed lookup is performed. This requires traversing the non-leaf pages of a B-tree by obtaining a read lock on each page, finding the next page to access, and releasing the read lock. When a leaf page is reached, the data page is locked and accessed. In the case of an update (an update is defined to change both the record and one index) a write-lock is obtained on the leaf page of the B-tree. The index page and data page are modified and the update is logged. The change is logged by recording both a before and after image of the record, and the index locks are released.

A transaction may be decomposed into operations whose cost may be expressed as a combination of logging, I/O, and locking costs. In the data manager models, the logging cost is proportional to the record size. On the read optimized file system logically contiguous blocks of a file are stored contiguously on disk so that a sequential read of a file may be performed at the sequential disk speeds. However, since the files are not being read sequentially in this model, there is no benefit derived from this optimization. Therefore, even on the read optimized file system, the I/O costs are proportional to the random read time of the disks. On a write optimized file system, the reads are also performed randomly, but the writes are all performed sequentially (section 3.3 provides a detailed description of how this is achieved). Finally, as it is assumed that the data manager maintains its own lock manager, the locking cost is strictly a function of the number of locks and independent of any system call overhead.

3.2. The Operating System Model

As the operating system knows nothing about the internal structure of files, it cannot distinguish between data and index updates. In order to guarantee serializability it must perform strict two phase locking [GRAY76] on all accessed and modified subpages. Assuming that there are S subpages per page, in traversing the B-tree, $\log_2 S$ subpages, selected uniformly from the filled subpages within the page, are selected for locking to model the search for a key within the page.[1] To modify a leaf page, one of the selected subpages is write-locked, and all subpages after it on the page are also write-locked, to allow the shuffling of keys within the page. This requires the operating system to obtain multiple write locks (on average half the number of filled subpages per page) for each B-tree page modified as compared to the data manager's one lock. Furthermore, since we are assuming that all the transaction support is provided in the operating system, each lock request requires a system call. Although we add the cost of a system call to each lock request in all the embedded models, this is an artificially high penalty since the data manager will also incur a system call each time a page, which is not resident in the buffer pool, is requested.

The final difference between the data manager and the operating system embedded model is the amount of logging information. Since the operating system may not perform logical logging, we resort to physical logging and save both before and after images of each subpage that is modified. In the case of inserts and deletes, this number may become quite large since multiple subpages per index page are modified.

We decompose the transaction costs into logging, I/O, and locking. This time, the logging cost is proportional to the size of a subpage, the I/O costs are proportional to the random disk access time, and the locking cost is a function of the number of locks, the number of subpages per page, and the system call overhead.

3.3. The Log Structured File System Model

A log-structured file system (LFS) uses the disk system as a continuously wrapping log. Rather than modifying files in place, newly written data pages and their describing meta data are written sequentially to the disk log. A large number of dirty data pages, their describing metadata, and a summary block, identifying the file and logical block number of each dirty page, are written sequentially in a single unit referred to as a segment. Figure 1 shows the allocation of three files in a log-structured file system.

[1] The $\log_2$ assumes a binary search is used to locate the correct subpage.

Traditional logs usually provide only sequential access for reading, but a log-structured file system builds its meta data into the log itself, so that random retrieval is also possible. Its structure is very similar to a UNIX file system [MCKU84]. In a UNIX file system, the location of the blocks of a file are stored in index structures which reside on fixed places on disk. These index structures are the meta data written to the log in LFS. In an LFS, the location of this meta data is not fixed. All the meta data is stored in a single file, called the map file. There is one special index structure, called the super block, which describes where the blocks of the map file reside in the LFS. The super block may be cached in main memory and appended to the log at checkpoint time.

This structure makes all writes sequential while retaining the ability to perform random retrieval. To locate a block of a file, the super block (which is cached in main memory) is accessed to find the location of the file's index structure. This index structure is then read to determine where the blocks of that file reside. Each time a file is written, both the newly written data and a new version of its index structure are written, and the superblock is updated to reflect the new location of the file's index structure.

Recovering a log-structured file system is similar to standard database recovery [HAER83]. The file system

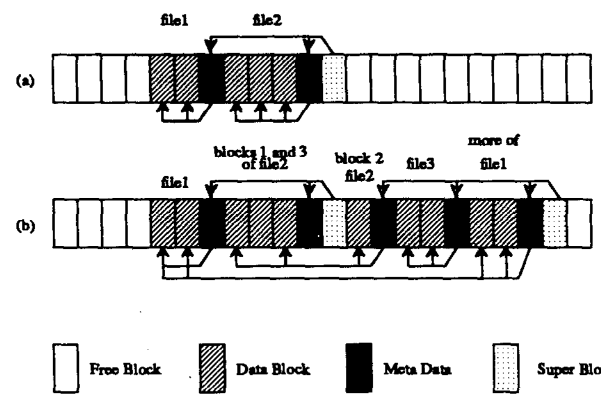

Figure 1: A Log Structured File System. In figure (a), two files have been written, file1 and file2. Each has an index structure in the meta data block which is allocated after it on disk. These meta data blocks are referenced via the superblock which is also appended to the log. In figure (b), the middle block of file2 has been modified. A new version of it is added to the log, as well as a new version of its meta data. Then file3 is created, causing its blocks and meta data to be appended to the log. Next, file1 has two more blocks appended to it. These two blocks and a new version of file1's meta data are appended to the log. Finally, another copy of the super block is appended to the log.

is read backwards until the last copy of the super block is found. Then, the summary block for each segment is read, and the super block is updated to reflect the new blocks written since the last checkpoint.

At some point, the disk system fills, requiring the log to wrap. At this time some blocks in the log will be "dead", that is, they will have been superceded by new versions of the block or the file will have been deleted. A cleaning process reclaims regions of contiguous space from the log by reading the tail of the log, discarding "dead" blocks, appending "live" blocks back into the log, and updating the meta-data appropriately. Space is continually reclaimed from the tail of the log and allocated to the head of the log [ROSE89]. The cost of log wrapping has not been taken into account in our simulations.

There are two characteristics of a log-structured file system that make it desirable for transaction processing. First, a large number of dirty pages are written to a single, large (several megabytes), contiguous region of the disk. Since only a single seek is performed to write out a large number of dirty blocks, the "per write" overhead is much closer to that of a sequential disk access rather than that of a random disk access. Secondly, since data is written using a "no overwrite" policy, before images of updated pages exist elsewhere in the file system.

In the LOG model, we take advantage of the sequential nature of writes, but not the "no-overwrite" policy of the log-structured file system. The logging and locking information is identical to that for the operating system model. However, the I/O component of the transaction cost is proportional to the random disk access time for reading, and the sequential disk access time for writing.

In LOG2, we take advantage of both the sequential nature of writes and the "no-overwrite" policy of the log-structured file system. Instead of logging before and after images of the subpages being modified, all dirty pages are forced to disk at commit time. However, since a single page may contain subpages modified by more than one transaction, subpages for uncommitted transaction may get written to disk. In order to guarantee the ability to abort any uncommitted transactions, we require a log which records the location of the previous and current versions of all dirty subpages. This logging information must be forced to disk before the dirty pages themselves. The difference between the LOG and LOG2 models is that in LOG2, log records are very small (16 bytes) and the total logging overhead is proportional only to the number of subpages modified rather than to the size of the subpages.

3.4. Model Summary

In the discussion that follows, DM refers to the data manager on a read optimized file system, DML refers to the data manager on a log-structured file system, OS refers to transaction support embedded in a read optimized file system, LOG refers to embedded support in a log-structured file system using a full, and LOG2 refers to embedded support in a log-structured file system, using the file system in place of a traditional log. For each component of transaction cost, logging, I/O, and locking, table 3 indicates the parameters on which this component of the cost is dependent for each model.

	logging	I/O (read)	I/O (write)	locking
DM	num updates record size	random	random	num locks
DML	num updates record size	random	seq	num locks syscall cost
OS	num updates subpage size	random	random	num locks syscall cost subpages/page
LOG	num updates subpage size	random	seq	num locks syscall cost subpages/page
LOG2	num updates	random	seq	num locks syscall cost subpages/page

Table 3: A Comparison of Five Transaction Processing Models.

4. Simulation Results

The three potential performance bottlenecks are the CPU, the disk system, and lock contention. By isolating each of these resources and we can stress all five systems in each dimension, resulting in a characterization of the performance of each model across a wide range of configurations. The simulation configuration consisted of a 1G data file (implying a total database size, with indices, of 2.2G) and 10 disks. By varying the CPU speed and the locality of accesses, CPU bound, disk bound, and lock contention bound configurations were analyzed. Each of the data points represents 5 runs of 10000 transactions each. The variance across the 5 runs was approximately 1% of the average and yielded 95% confidence intervals of approximately 2%.

4.1. CPU and Disk Boundedness

With a large data file (1G), 10 disks, and uniform distribution of accesses, a CPU bound environment was created by setting the CPU speed to 1 MIPS[2]. Since the

[2] This processor speed is excessively slow since we are ignoring a great deal of software overhead such as query processing, communication, and operating system overhead. The goal is to focus on those costs which differ in the models rather than on those which are the same.

access pattern is uniform, there is little contention on either indices or the data file (the probability of conflict between any two transactions is approximately 5%), and there is no need to set the locking granularity (or subpage size) any smaller than the page size for the embedded models. This differs from [KUMAR87] in that he always assumes that the embedded models must perform subpage locking, and subpages are 128 bytes. Figure 2 shows the results of varying the degree of multiprogramming until the CPU becomes saturated. In this configuration the two data manager models provide better performance than any of the embedded systems. Whereas Kumar found this difference to be 30% or more in a CPU bound configuration, we see that at a degree of multiprogramming of 20, the difference in throughput between the DM and the OS results is approximately 17% ($\frac{T_{DM}-T_{OS}}{T_{DM}}$), and the difference between DML and either LOG or LOG2 is 20% ($\frac{T_{DML}-T_{LOG}}{T_{DML}}$). This difference is explained by the fact that we require only 1 lock per level of the B-tree while Kumar required 4. Therefore, we find that the difference in performance is due only to the system calls required by the file system based models.

This difference in performance, in terms of throughput, may be expressed by the function:

$$T_{non-dm} = \left[\frac{LN + C}{N(L+S) + C} \right] T_{dm} \quad OR$$

$$T_{dm} = \left[1 + \frac{NS}{LN + C} \right] T_{non-dm}$$

where

N is the number of locks required,
L is the CPU overhead for acquiring a lock,
C is the CPU overhead per transaction not associated with locking, and
S is the cost of a system call (in ms).

For the workload simulated, T_{dm} is $1.2T_{non-dm}$ or $(1+.4S)T_{non-dm}$. It is apparent that the cost of a system call will have a tremendous impact on the performance. Figure 3 graphically depicts this difference in performance as the cost of a system call is varied. In the preceding simulation a .5 ms overhead for system calls was used, yielding a 17-20% difference in performance between the data manager and embedded models. At .25 ms (250 instructions), the difference between the data manager and embedded models drops to 12%.

Although this configuration is CPU bound, the write optimized file system based models provide better performance than the read optimized file system based models. Comparing the DML performance with the DM performance, we see a gap of nearly 12% (4.1 tps vs. 3.6

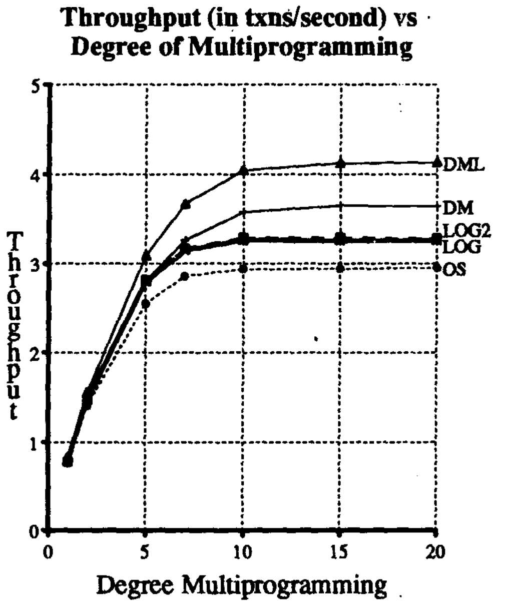

Figure 2: CPU Bounding under Low Contention. The degree of multiprogramming is varied in a low contention configuration. Due to the system call required per lock in the OS, LOG, and LOG2 models, the data managers provide the best performance.

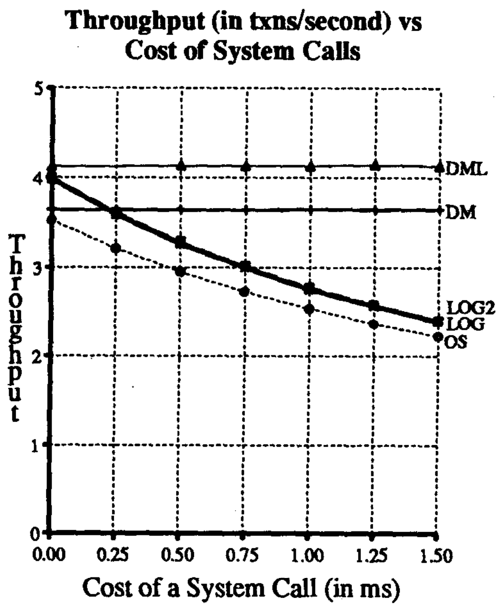

Figure 3: Effect of the Cost of System Calls. As system calls become more costly (in terms of CPU cycles), the difference in performance between the data manager and the embedded models widens.

tps), and comparing LOG2 with OS, we see a gap of 10% (3.3 tps vs. 3.0 tps). In each of these situations, the CPU cost for both configurations is the same. Since the configuration is not disk bound, the better performance of the log-structured file system is unexpected. Upon closer inspection, it becomes clear that although the disks are under utilized, the read optimized file systems are running at approximately 50% disk utilization while the write optimized models are running at approximately 33% utilization.

To understand how this effects throughput, we need to examine what happens as dirty blocks are flushed to disk. In the read optimized models, flushing a dirty block keeps the disk busy for the time of a random access (approximately 28.3 ms)[3]. When dirty data blocks are written, an incoming read request may be delayed for up to 28.3 ms. Even if these writes are attempted during idle disk cycles, subsequent read requests may still queue up behind the writes and be delayed. On the other hand, when the log based models flush their dirty blocks, they write a large number of blocks at sequential speed (1.99 ms). Therefore, the potential delay incurred per block flushed is much less. Even when the disks are not the bottleneck, the difference in write performance of the disk systems does impact the resulting throughput.

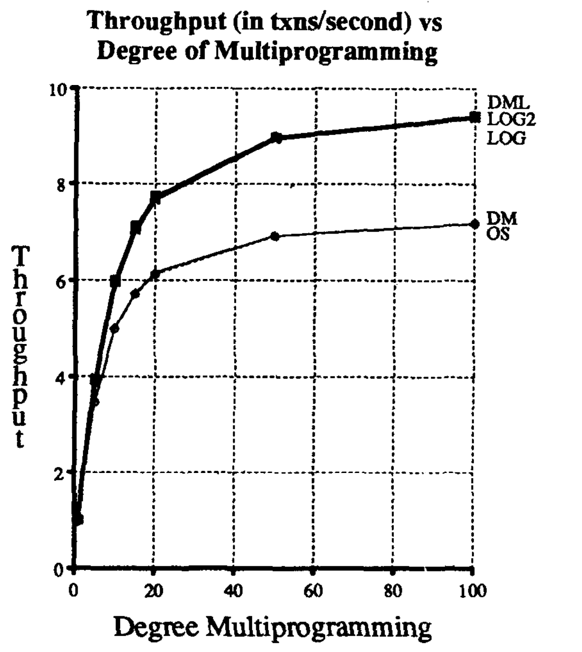

Figure 4: Disk Bounding under Low Contention. Since there is sufficient CPU power to support the more expensive embedded systems, the file system is the determiner of performance, and we see the write optimized file systems providing the best performance.

[3] All disk times are based on the performance specification of the Fujitsu Eagle M2361A [FUJI84].

By increasing the processor speed to 10 MIPS, the configuration becomes disk bound. Once again, the degree of multiprogramming was varied to determine a saturation point. These results are shown in figure 4, and as expected, the log-structured models provide the best performance, by approximately 23% (9.4 tps for DML and 7.2 tps for DM). Furthermore, although the configuration is disk bound, the size of the log does not have a significant impact on performance. Both the data manager and operating system models exhibit nearly identical performance, even though the operating system maintains a much larger log. As in the CPU bound case, these results differ dramatically from [KUMAR87]. He found that in disk bound configurations the data manger out performed the operating system embedded model and attributed this difference to the size of the log. Although the OS keeps a much larger log than DM, their performance is nearly identical as shown by the overlapping lines in figure 4. Similarly, DML, LOG and LOG2 exhibit nearly identical performance although these models have different sized logs as well. Since logging occurs at sequential speed in all the models, it is clear that performance is dominated by the random disk access times.

Having analyzed the extremes of disk boundedness and CPU boundedness, we now analyze the region in between. By varying the processor speed, the results shown in figure 5 were obtained. Between any two models, there are two factors which contribute to the performance differential: the file system or size of the log and the location of transaction support (data manager or embedded). At 1 MIPS, the CPU bound configuration, the file system component accounts for 10-12% of the differential (the difference between DML and DM or OS and LOG) and the location component 19-20% (the difference between OS and DM or LOG and DML). By 2 MIPS, that emphasis has shifted so that the file system component is 19-21% and the location component is 15-17% Finally, by the disk bound point, 3 MIPS, the location component is 0 (the DM and OS lines overlap, as do the DML, LOG, and LOG2) and the file system accounts for a 22% difference in performance. However, at any point along the curves, the best performance is provided by supporting transactions within the data manager on top of a log-structured file system.

As was observed in the disk bound configuration, the size of the log does not contribute significantly to the performance of the systems. The difference in I/O costs between DML and LOG is that DML is able to perform logical logging (proportional to the record size) while LOG performs physical logging (proportional to page size). The logging difference between LOG and LOG2 is that the LOG2 model requires a log even smaller than DML (16 bytes per modification rather than 2 records). Since logging is always performed at sequential speeds,

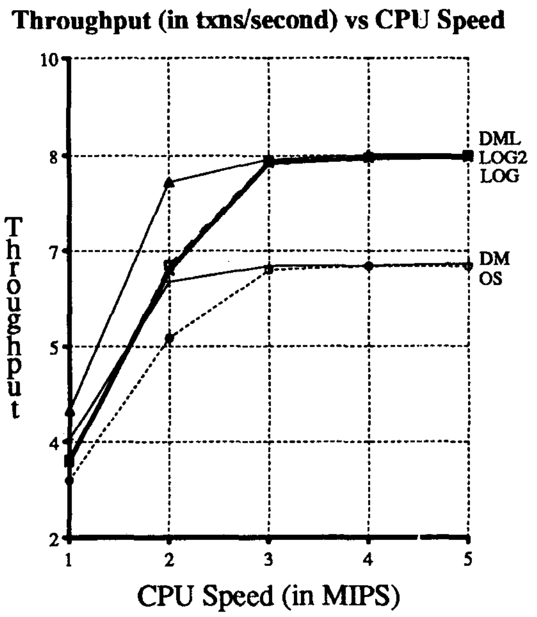

Figure 5: Effect of CPU Speed on Transaction Throughput. Increasing CPU speed moves the configuration from a state of CPU boundedness to disk boundedness. Even before the systems become completely disk bound (at 3 MIPS), the major factor contributing to the performance differential is the file system and not the location of transaction support.

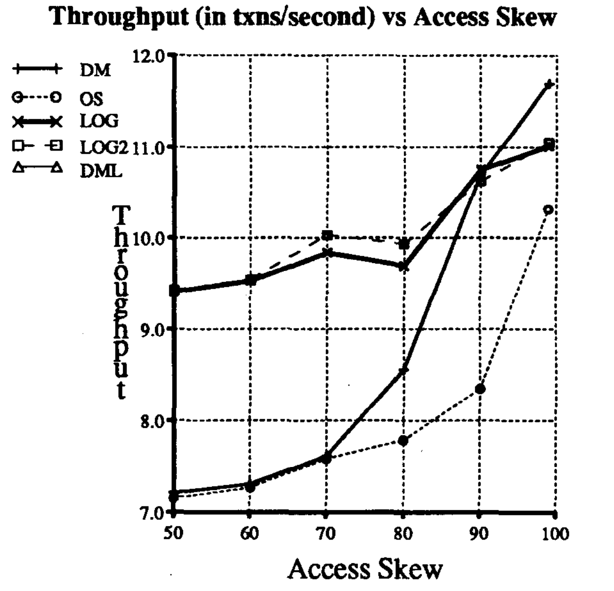

Figure 6: Effect of Skewed Access Distribution on Throughput. Contention begins to impact performance when the skew reaches greater than 70/30. The embedded models diverge from their data manager counterparts at this point.

the total time required to log a transaction is still a small part of the total I/O time (under 1%), and the resulting performance is the same for all three systems. Therefore, the primary benefit of the log-structured file system implementation is its superior write performance, not its no overwrite policy.

4.2. Lock Contention

All the preceding tests were run with a uniform access pattern over a 1G data file. The next issue we investigated was the effect of lock contention on these results. To induce contention, the distribution of accesses to the database was skewed. The multiprogramming level was set to 100, the number of disks to 10, the CPU speed to 10 MIPS, and the distribution was varied from uniform (50/50; 50% of the accesses to 50% of the database) to extremely contention bound (99/1; 99% of the accesses to 1% of the database). Figure 6 shows these results.

There are two factors at work here. First, since the configuration is initially disk bound, the skewing of the access patterns results in a higher buffer cache hit ratio and therefore improved performance. Secondly, the skewing of the access patterns induces hotspots in the database, and the contention for locks degrades performance. At the 70/30 skew point, the DM and OS lines diverge as do the DML and LOG/LOG2 lines. The data manager based models continue to take advantage of the improved buffer cache hit ratio and their performance climbs steadily. The OS model also exhibits improved performance, but not as much as the data managers since it is starting to suffer from contention on the indices (since index locks are held until transaction commit time in the embedded models). The LOG and LOG2 models actually suffer performance degradation as a result of the increased skewing and resulting contention.

Figure 7, which shows the number of aborts for each of the models as a function of this skewing, indicates that the embedded models exhibit higher abort rates than the data manager models from the 70/30 point until the 95/1 point. Since many more transactions are aborting, the resulting throughput is lower, therefore, in a contention bound environment the coarse grain page locking employed by the embedded models is unsatisfactory.

We investigated three techniques to reduce the effect of lock contention in the embedded models. First, subpage locking, as described in section 3.2, was used. Next, the page size was reduced and locking was performed on full pages. Finally, a modified subpage locking technique similar to that described in [KUMAR89] was used.

Subpage locking reduces the locking granularity, and as a result, the degree of contention, but not as much as expected. Since updates to a B-tree page require shuffling around the entries on a page, multiple subpages get locked for each update. In addition, the pages locked always follow the subpage on which the insert is

Number of Aborts ($\log_{10}$ / 10000 successful) vs Access Skew

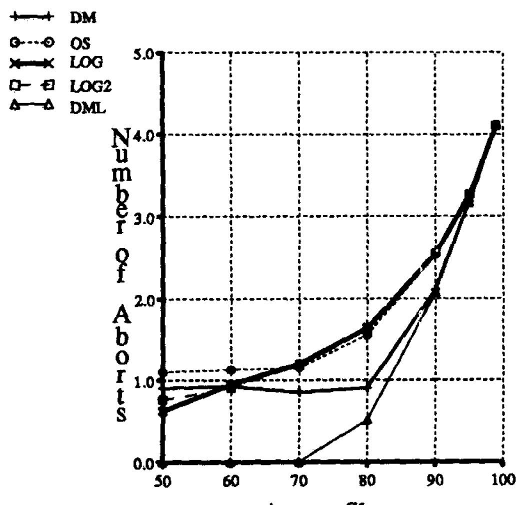

Figure 7: Effect of Access Skewing on the Number of Aborted Transactions. The abort rate begins climbing at a 70/30 skew for the embedded systems, but at an 80/20 skew for the data manager.

Throughput (in txns/second) vs Access Skew (Subpage Locking)

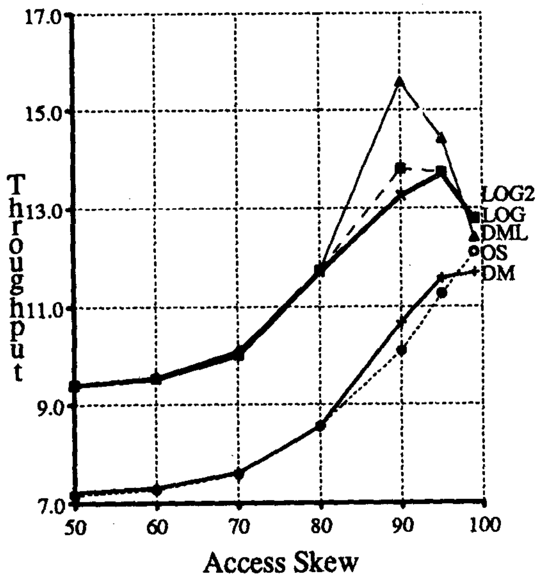

Figure 8: Impact of Access Skewing with Subpage Locking. By reducing the locking granularity, the embedded models can regain some of the performance lost to contention.

being made. As a result, the distribution of write locked subpages is skewed towards those at the end of the page. In addition, the CPU cost per level of the B-tree is higher since multiple subpage locks are required to find the correct subpage. Therefore, if a high contention environment is CPU bound, changing the locking granularity will not improve performance. If the CPU is not the bottleneck, some of the performance lost to contention may be regained.

Figure 8 shows the same contention bound environment as figure 6, but uses subpage locking for the embedded models. In the region between 70/30 and 95/5 the embedded models come much closer to equaling their data manager counterparts. In the case of the read optimized file systems (DM and OS), the difference is at most 6% (at the 90/10 point). For the log-structured file system, the largest gap is under 12% (also at 90/10). At the most contention bound point, even the data manager models exhibit extreme contention due to locking conflicts on the data file. Since the embedded models are able to lock subpages of the data file, at the 99/1 point, the embedded models exhibit better performance.

The next technique to reduce contention was to decrease the page size and lock on full pages. While decreasing the page size does reduce contention, it may also increase the depth of the B-tree. Increasing the depth of the B-tree adds extra I/O to each operation as well as adding an additional lock request to each traversal. As a result, we find that reducing the page size is beneficial only if it does not increase the depth of the B-tree. In the case of the contention bound simulations, reducing

Throughput (in txns/second) vs Access Skew (Optimal Page size)

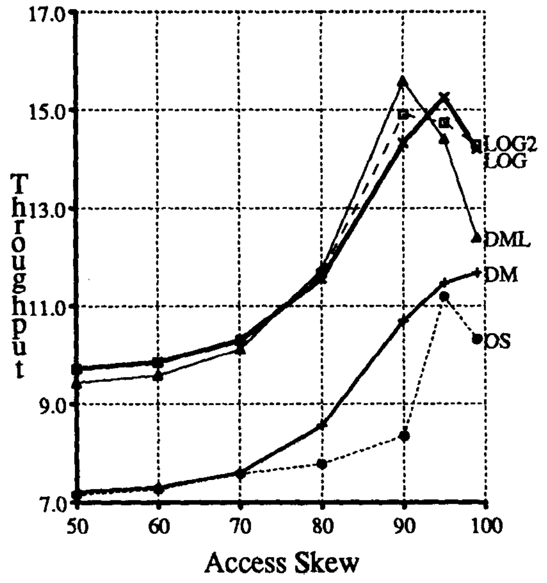

Figure 9: Impact of Access Skewing with Variable Page Size. Varying the page size compensates for some of the penalty from contention in the embedded systems.

the page size from 4K bytes to 2K bytes does not increase the depth of the B-tree. The results in figure 9 show the same contention bound environment using page locking and selecting the optimal page size for each model (all the embedded models used 4096 byte pages for skews of 50, 60, 70; 128 byte pages for 80, and 512 byte pages for 90, 95, and 99). For the read optimized file system, using page size to reduce contention is less effective than using subpages, since the largest OS/DM differential is 22% at the 90/10 point. On the other hand, the differential for the write optimized file system has gone from 12% at the 90/10 point in figure 8 to 4% at the 90/10 point in figure 9. Furthermore, the write optimized embedded models actually surpass the write optimized data manager at 95/5 rather than 99/1 as before. Depending on the file system, varying either the subpage size or the page size is an effective mechanism for handling lock contention.

The last approach reverts to subpage locking, but avoids the overhead of multiple locks per level and the skewing of locked subpages. This is similar to the proposal in [KUM89], but has lower CPU costs. In both Kumar's and our model, each subpage is treated as an independent bucket of entries. In the normal case an operation requires locating the correct subpage and modifying only that subpage. In Kumar's method, this requires keeping the smallest keys for each subpage on the first subpage of the page. To avoid bottlenecking on this first page, we perform a non locking binary search across subpages to select the correct page.

Another difference in the two algorithms is that Kumar keeps entries in each bucket chained in a link list, requiring linear time to search each bucket, while we keep entries within a subpage sorted maintaining the $O(log)$ search time. Therefore, normal operations require no more time using this structure than they do in the regular full page structure.

In the case of an overflow of a subpage or a change to the smallest key on a subpage, we lock all the subpages within a page and reorganize. Since, on average, we expect to move half the entries on a page, this reorganization costs the same as a normal page oriented key insert. Whereas Kumar assumes that reorganization is required every 600 updates, we assume reorganization is required once in every 10 updates since we require reorganization or full page locking both when buckets fill as well as when the first key on a page is modified.

In the next set of simulations, we used this modified subpage locking. The subpage size was set at 512 bytes (8 subpages per page and approximately 22 keys per page). These results are shown in figure 10. Since this locking protocol offers the smaller locking granularity of subpage locking without the extra CPU overhead of

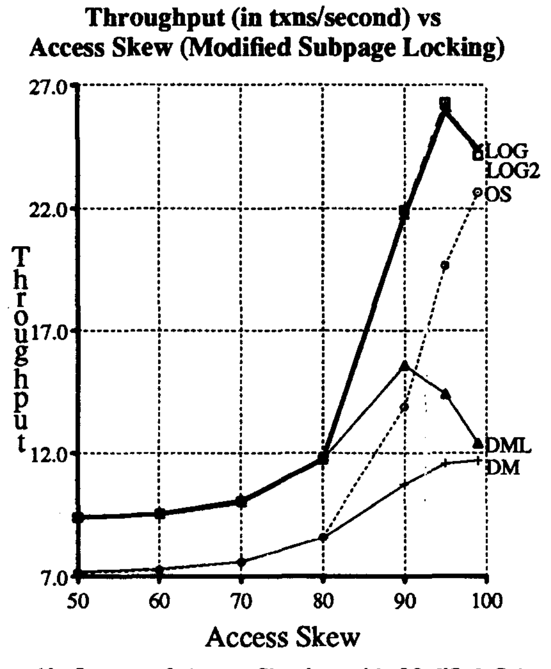

Figure 10: Impact of Access Skewing with Modified Subpage Locking. By reducing the locking granularity, the embedded systems are able to surpass the data manager in an environment with extremely high contention.

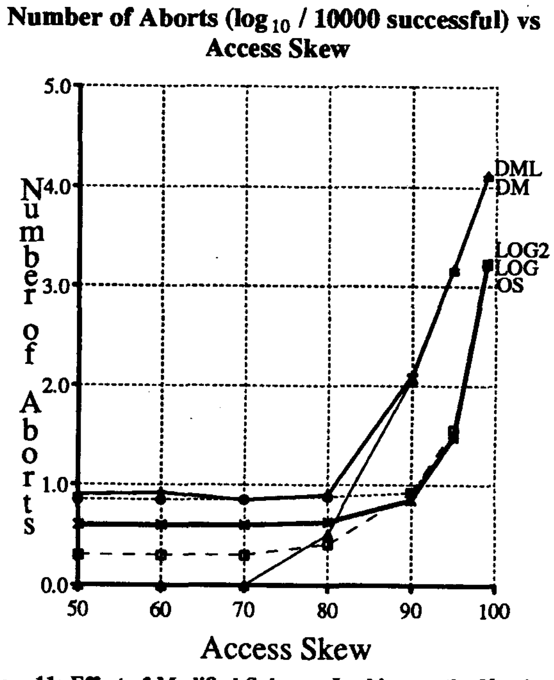

Figure 11: Effect of Modified Subpage Locking on the Number of Aborted Transactions. The new locking mechanism reduces the number of aborts by a factor of 10, thus allowing the high throughput rates observed in figure 10.

multiple locks per update, its performance is even better than the data manager's, and we see the performance difference between the data manager and embedded models in excess of 45% at the 95/5 point. Looking at the number of aborts for the embedded models shown in figure 11, we see that lock contention is virtually eliminated until the 90/10 point, and beyond that point, the number of aborts in the embedded models is an order of magnitude smaller than for the data managers.

5. Conclusions

Independent of whether transaction support is embedded in the file system or implemented in the data manager, the log structured file system offers better performance than the traditional read optimized file system. Its major benefit is its improved write performance, not its no overwrite policy. In fact, as we see from the results in disk bound configurations, the size of the log has very little impact on the resulting performance. This is explained by the fact that logging always occurs at sequential speeds and is a very small fraction of the total I/O time.

Since logging is not an important factor, we find that embedded transaction support performs as well as the data manager support in disk bound configurations. Whether we use a read optimized or write optimized file system, we find that the data manager and embedded models offer nearly identical performance. As a result, supporting transactions within the file system is a feasible solution, when the system is I/O bound.

As Kumar concluded, when the CPU is the bottleneck, there is a penalty in embedding transaction support in a file system. However, when lock contention is not a factor, there is no need to perform subpage locking, and the difference in performance is directly proportional to the cost of a system call and is usually under 20%. Therefore, the feasibility of an embedded transaction manager is strictly dependent on the system call overhead.

Finally, as lock contention becomes a factor in limiting performance, all models experience some degradation, but the data manager suffers the least due to its use of semantic information for B-tree locking. The embedded models may recoup most of this performance loss through variable subpage and page size. In some cases, where the CPU is not a critical resource, embedded systems with modified subpage locking not only recoup this loss, but provide better throughput than the data manager.

Except in the most CPU bound environments, there is virtually no penalty incurred in embedding transaction support in the operating system. It does, however, require careful and defensive design to avoid index contention as well as operating system flexibility to vary the page and subpage sizes as needed.

There are several areas which warrant further investigation. We have not accounted for the cost of log wrapping in the log-structured file system. This will reduce the benefit of the log-structured file system, but it is not clear how great this impact will be. In addition, the use of RAID devices [PATT88] will penalize the small writes that occur in a read optimized file system. These issues will be examined in later research.

6. References

[BAYER77] Bayer, R., Scholnick, M., "Concurrency Operations on B-Trees," *Acta Informatica*, 1977.

[FUJI84] M2361A Mini-Disk Drive Engineering Specifications, Fujitsu Limited, 198 4.

[GRAY76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., "Granularity of locks and degrees of consistency in a large shared data base.", *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, pp. 365-394.

[HAER83] Haerder, T. Reuter, A. "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, 15(4), 237-318, 1983.

[KUM87] Kumar, A., Stonebraker, M., "Performance Evaluation of an Operating System Transaction Manager", *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, England, 1987.

[KUM89] Kumar, A., Stonebraker, M., "Performance Considerations for an Operating System Transaction Manager", *IEEE Transactions on Software Engineering*, 15(6), June 1989.

[MCKU84] Marshall Kirk McKusick, William Joy, Sam Leffler, and R. S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.

[MITC82] Mitchell, J., Dion, J., "A Comparison off Two Network-Based File Servers", *Communications of the ACM*, 25(4), April 1982.

[MUEL83] Mueller, E. etc al., "A Nested Transaction Mechanism for LOCUS", *Proceedings 9th Symposium on Operating System Principles*, October 1983.

[OUST88] Ousterhout, J., Douglis, F., "Beating the I/O Bottleneck: A Case for Log Structured File Systems", Computer Science Division (EECS), University of California, Berkeley, UCB/CSD 88/467, October 1988.

[PU86] Pu, C., Noe, J., "Design of Nested Transactions in Eden", Technical Report 85-12-03, Dept. of

Computer Science, Univ of Washington, Seattle, WA, 1986.

[PATT88] Patterson, D. et. al., "RAID: Redundant Arrays of Inexpensive Disks," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Ill., June 1988.

[ROSE89] Rosenblum, M., "The Design of LFS", Technical Report, Computer Science Department, University of California, Berkeley, July, 1989.

[SPE88A] Spector, Rausch, Bruell, "Camelot: A Flexible, Distributed Transaction Processing System", *Proceedings of Spring COMPCON 1988*, February 1988.

[SPE88B] Spector, A, Swedlow, K., *Guide to the Camelot Distributed Transaction Facility*, Computer Science Department, Carnegie-Mellon University, Release 1, edition 0.98(51), May 1988.

[STON81] Stonebraker, M., "Operating System Support of Data Managers", *Communications of the ACM*, 24(7), July 1981.

[STON85] Stonebraker, M., "Problems in Supporting Data Base Transactions in an Operating System Transaction Manager", *Operating System Review*, 19(1), January 1985.

[TRA82] Traiger, I., "Virtual Memory Management for Data Base Systems", *Operating System Review*, 16(4), October 1982.

[WALK83] Walker, Popek, English, Kline, Thie, "The LOCUS Distributed Operating System", *Proceedings 9th Symposium on Operating System Principles*, October 1983.