LIBTP: Portable, Modular Transactions for UNIX[1]

Margo Seltzer
Michael Olson
University of California, Berkeley

Abstract

Transactions provide a useful programming paradigm for maintaining logical consistency, arbitrating concurrent access, and managing recovery. In traditional UNIX systems, the only easy way of using transactions is to purchase a database system. Such systems are often slow, costly, and may not provide the exact functionality desired. This paper presents the design, implementation, and performance of LIBTP, a simple, non-proprietary transaction library using the 4.4BSD database access routines (**db**(3)). On a conventional transaction processing style benchmark, its performance is approximately 85% that of the database access routines without transaction protection, 200% that of using **fsync**(2) to commit modifications to disk, and 125% that of a commercial relational database system.

1. Introduction

Transactions are used in database systems to enable concurrent users to apply multi-operation updates without violating the integrity of the database. They provide the properties of atomicity, consistency, isolation, and durability. By atomicity, we mean that the set of updates comprising a transaction must be applied as a single unit; that is, they must either all be applied to the database or all be absent. Consistency requires that a transaction take the database from one logically consistent state to another. The property of isolation requires that concurrent transactions yield results which are indistinguishable from the results which would be obtained by running the transactions sequentially. Finally, durability requires that once transactions have been committed, their results must be preserved across system failures [TPCB90].

Although these properties are most frequently discussed in the context of databases, they are useful programming paradigms for more general purpose applications. There are several different situations where transactions can be used to replace current ad-hoc mechanisms.

One situation is when multiple files or parts of files need to be updated in an atomic fashion. For example, the traditional UNIX file system uses ordering constraints to achieve recoverability in the face of crashes. When a new file is created, its inode is written to disk before the new file is added to the directory structure. This guarantees that, if the system crashes between the two I/O's, the directory does not contain a reference to an invalid inode. In actuality, the desired effect is that these two updates have the transactional property of atomicity (either both writes are visible or neither is). Rather than building special purpose recovery mechanisms into the file system or related tools (*e.g.* **fsck**(8)), one could use general purpose transaction recovery protocols after system failure. Any application that needs to keep multiple, related files (or directories) consistent should do so using transactions. Source code control systems, such as RCS and SCCS, should use transaction semantics to allow the ''checking in'' of groups of related files. In this way, if the ''check-in'' fails, the transaction may be aborted, backing out the partial ''check-in'' leaving the source repository in a consistent state.

A second situation where transactions can be used to replace current ad-hoc mechanisms is in applications where concurrent updates to a shared file are desired, but there is logical consistency of the data which needs to be preserved. For example, when the password file is updated, file locking is used to disallow concurrent access. Transaction semantics on the password files would allow concurrent updates, while preserving the logical consistency of the password database. Similarly, UNIX utilities which rewrite files face a potential race condition between their rewriting a file and another process reading the file. For example, the compiler (more precisely, the assembler) may

[1] To appear in the *Proceedings of the 1992 Winter Usenix*, San Francisco, CA, January 1992.

have to rewrite a file to which it has write permission in a directory to which it does not have write permission. While the ‘‘.o’’ file is being written, another utility such as **nm**(1) or **ar**(1) may read the file and produce invalid results since the file has not been completely written. Currently, some utilities use special purpose code to handle such cases while others ignore the problem and force users to live with the consequences.

In this paper, we present a simple library which provides transaction semantics (atomicity, consistency, isolation, and durability). The 4.4BSD database access methods have been modified to use this library, optionally providing shared buffer management between applications, locking, and transaction semantics. Any UNIX program may transaction protect its data by requesting transaction protection with the **db**(3) library or by adding appropriate calls to the transaction manager, buffer manager, lock manager, and log manager. The library routines may be linked into the host application and called by subroutine interface, or they may reside in a separate server process. The server architecture provides for network access and better protection mechanisms.

2. Related Work

There has been much discussion in recent years about new transaction models and architectures [SPEC88][NODI90][CHEN91][MOHA91]. Much of this work focuses on new ways to model transactions and the interactions between them, while the work presented here focuses on the implementation and performance of traditional transaction techniques (write-ahead logging and two-phase locking) on a standard operating system (UNIX).

Such traditional operating systems are often criticized for their inability to perform transaction processing adequately. [STON81] cites three main areas of inadequate support: buffer management, the file system, and the process structure. These arguments are summarized in table one. Fortunately, much has changed since 1981. In the area of buffer management, most UNIX systems provide the ability to memory map files, thus obviating the need for a copy between kernel and user space. If a database system is going to use the file system buffer cache, then a system call is required. However, if buffering is provided at user level using shared memory, as in LIBTP, buffer management is only as slow as access to shared memory and any replacement algorithm may be used. Since multiple processes can access the shared data, prefetching may be accomplished by separate processes or threads whose sole purpose is to prefetch pages and wait on them. There is still no way to enforce write ordering other than keeping pages in user memory and using the **fsync**(3) system call to perform synchronous writes.

In the area of file systems, the fast file system (FFS) [MCKU84] allows allocation in units up to 64KBytes as opposed to the 4KByte and 8KByte figures quoted in [STON81]. The measurements in this paper were taken from an 8KByte FFS, but as LIBTP runs exclusively in user space, there is nothing to prevent it from being run on other UNIX compatible file systems (e.g. log-structured [ROSE91], extent-based, or multi-block [SELT91]).

Finally, with regard to the process structure, neither context switch time nor scheduling around semaphores seems to affect the system performance. However, the implementation of semaphores can impact performance tremendously. This is discussed in more detail in section 4.3.

The Tuxedo system from AT&T is a transaction manager which coordinates distributed transaction commit from a variety of different local transaction managers. At this time, LIBTP does not have its own mechanism for distributed commit processing, but could be used as a local transaction agent by systems such as Tuxedo [ANDR89].

Buffer Management	• Data must be copied between kernel space and user space. • Buffer pool access is too slow. • There is no way to request prefetch. • Replacement is usually LRU which may be suboptimal for databases. • There is no way to guarantee write ordering.
File System	• Allocation is done in small blocks (usually 4K or 8K). • Logical organization of files is redundantly expressed.
Process Structure	• Context switching and message passing are too slow. • A process may be descheduled while holding a semaphore.

Table One: Shortcomings of UNIX transaction support cited in [STON81].

The transaction architecture presented in [YOUN91] is very similar to that implemented in the LIBTP. While [YOUN91] presents a model for providing transaction services, this paper focuses on the implementation and performance of a particular system. In addition, we provide detailed comparisons with alternative solutions: traditional UNIX services and commercial database management systems.

3. Architecture

The library is designed to provide well defined interfaces to the services required for transaction processing. These services are recovery, concurrency control, and the management of shared data. First we will discuss the design tradeoffs in the selection of recovery, concurrency control, and buffer management implementations, and then we will present the overall library architecture and module descriptions.

3.1. Design Tradeoffs

3.1.1. Crash Recovery

The recovery protocol is responsible for providing the transaction semantics discussed earlier. There are a wide range of recovery protocols available [HAER83], but we can crudely divide them into two main categories. The first category records all modifications to the database in a separate file, and uses this file (log) to back out or reapply these modifications if a transaction aborts or the system crashes. We call this set the **logging protocols**. The second category avoids the use of a log by carefully controlling when data are written to disk. We call this set the **non-logging protocols**.

Non-logging protocols hold dirty buffers in main memory or temporary files until commit and then force these pages to disk at transaction commit. While we can use temporary files to hold dirty pages that may need to be evicted from memory during a long-running transaction, the only user-level mechanism to force pages to disk is the **fsync**(2) system call. Unfortunately, **fsync**(2) is an expensive system call in that it forces all pages of a file to disk, and transactions that manage more than one file must issue one call per file.

In addition, **fsync**(2) provides no way to control the order in which dirty pages are written to disk. Since non-logging protocols must sometimes order writes carefully [SULL92], they are difficult to implement on Unix systems. As a result, we have chosen to implement a logging protocol.

Logging protocols may be categorized based on how information is logged (physically or logically) and how much is logged (before images, after images or both). In **physical logging**, images of complete physical units (pages or buffers) are recorded, while in **logical logging** a description of the operation is recorded. Therefore, while we may record entire pages in a physical log, we need only record the records being modified in a logical log. In fact, physical logging can be thought of as a special case of logical logging, since the ‘‘records’’ that we log in logical logging might be physical pages. Since logical logging is both more space-efficient and more general, we have chosen it for our logging protocol.

In **before-image logging**, we log a copy of the data before the update, while in **after-image logging**, we log a copy of the data after the update. If we log only before-images, then there is sufficient information in the log to allow us to **undo** the transaction (go back to the state represented by the before-image). However, if the system crashes and a committed transaction’s changes have not reached the disk, we have no means to **redo** the transaction (reapply the updates). Therefore, logging only before-images necessitates forcing dirty pages at commit time. As mentioned above, forcing pages at commit is considered too costly.

If we log only after-images, then there is sufficient information in the log to allow us to redo the transaction (go forward to the state represented by the after-image), but we do not have the information required to undo transactions which aborted after dirty pages were written to disk. Therefore, logging only after-images necessitates holding all dirty buffers in main memory until commit or writing them to a temporary file.

Since neither constraint (forcing pages on commit or buffering pages until commit) was feasible, we chose to log both before and after images. The only remaining consideration is when changes get written to disk. Changes affect both data pages and the log. If the changed data page is written before the log page, and the system crashes before the log page is written, the log will contain insufficient information to undo the change. This violates transaction semantics, since some changed data pages may not have been written, and the database cannot be restored to its pre-transaction state.

The log record describing an update must be written to stable storage before the modified page. This is **write-ahead logging**. If log records are safely written to disk, data pages may be written at any time afterwards. This means that the only file that ever needs to be forced to disk is the log. Since the log is append-only, modified pages always appear at the end and may be written to disk efficiently in any file system that favors sequential ordering (*e.g.*, FFS, log-structured file system, or an extent-based system).

3.1.2. Concurrency Control

The concurrency control protocol is responsible for maintaining consistency in the presence of multiple accesses. There are several alternative solutions such as locking, optimistic concurrency control [KUNG81], and timestamp ordering [BERN80]. Since optimistic methods and timestamp ordering are generally more complex and restrict concurrency without eliminating starvation or deadlocks, we chose two-phase locking (2PL). Strict 2PL is suboptimal for certain data structures such as B-trees because it can limit concurrency, so we use a special locking protocol based on one described in [LEHM81].

The B-tree locking protocol we implemented releases locks at internal nodes in the tree as it descends. A lock on an internal page is always released before a lock on its child is obtained (that is, locks are not **coupled** [BAY77] during descent). When a leaf (or internal) page is split, a write lock is acquired on the parent before the lock on the just-split page is released (locks are **coupled** during ascent). Write locks on internal pages are released immediately after the page is updated, but locks on leaf pages are held until the end of the transaction.

Since locks are released during descent, the structure of the tree may change above a node being used by some process. If that process must later ascend the tree because of a page split, any such change must not cause confusion. We use the technique described in [LEHM81] which exploits the ordering of data on a B-tree page to guarantee that no process ever gets lost as a result of internal page updates made by other processes.

If a transaction that updates a B-tree aborts, the user-visible changes to the tree must be rolled back. However, changes to the internal nodes of the tree need not be rolled back, since these pages contain no user-visible data. When rolling back a transaction, we roll back all leaf page updates, but no internal insertions or page splits. In the worst case, this will leave a leaf page less than half full. This may cause poor space utilization, but does not lose user data.

Holding locks on leaf pages until transaction commit guarantees that no other process can insert or delete data that has been touched by this process. Rolling back insertions and deletions on leaf pages guarantees that no aborted updates are ever visible to other transactions. Leaving page splits intact permits us to release internal write locks early. Thus transaction semantics are preserved, and locks are held for shorter periods.

The extra complexity introduced by this locking protocol appears substantial, but it is important for multi-user execution. The benefits of non-two-phase locking on B-trees are well established in the database literature [BAY77], [LEHM81]. If a process held locks until it committed, then a long-running update could lock out all other transactions by preventing any other process from locking the root page of the tree. The B-tree locking protocol described above guarantees that locks on internal pages are held for extremely short periods, thereby increasing concurrency.

3.1.3. Management of Shared Data

Database systems permit many users to examine and update the same data concurrently. In order to provide this concurrent access and enforce the write-ahead logging protocol described in section 3.1.1, we use a shared memory buffer manager. Not only does this provide the guarantees we require, but a user-level buffer manager is frequently faster than using the file system buffer cache. Reads or writes involving the file system buffer cache often require copying data between user and kernel space while a user-level buffer manager can return pointers to data pages directly. Additionally, if more than one process uses the same page, then fewer copies may be required.

3.2. Module Architecture

The preceding sections described modules for managing the transaction log, locks, and a cache of shared buffers. In addition, we need to provide functionality for transaction *begin*, *commit*, and *abort* processing, necessitating a transaction manager. In order to arbitrate concurrent access to locks and buffers, we include a process management module which manages a collection of semaphores used to block and release processes. Finally, in order to provide a simple, standard interface we have modified the database access routines (**db**(3)). For the purposes of this paper we call the modified package the **Record Manager**. Figure one shows the main interfaces and architecture of LIBTP.

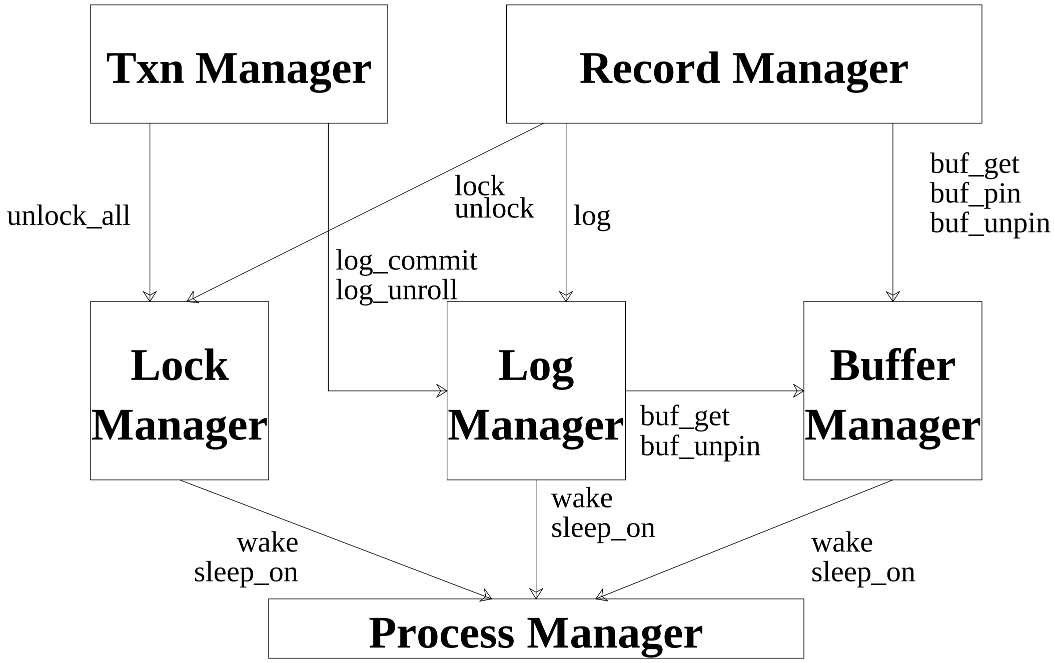

Figure 1: Library module interfaces.

3.2.1. The Log Manager

The **Log Manager** enforces the write-ahead logging protocol. Its primitive operations are *log*, *log_commit*, *log_read*, *log_roll* and *log_unroll*. The *log* call performs a buffered write of the specified log record and returns a unique log sequence number (LSN). This LSN may then be used to retrieve a record from the log using the *log_read* call. The *log* interface knows very little about the internal format of the log records it receives. Rather, all log records are referenced by a header structure, a log record type, and a character buffer containing the data to be logged. The log record type is used to call the appropriate redo and undo routines during *abort* and *commit* processing. While we have used the **Log Manager** to provide before and after image logging, it may also be used for any of the logging algorithms discussed.

The *log_commit* operation behaves exactly like the *log* operation but guarantees that the log has been forced to disk before returning. A discussion of our commit strategy appears in the implementation section (section 4.2). *Log_unroll* reads log records from the log, following backward transaction pointers and calling the appropriate undo routines to implement transaction abort. In a similar manner, *log_roll* reads log records sequentially forward, calling the appropriate redo routines to recover committed transactions after a system crash.

3.2.2. The Buffer Manager

The **Buffer Manager** uses a pool of shared memory to provide a least-recently-used (LRU) block cache. Although the current library provides an LRU cache, it would be simple to add alternate replacement policies as suggested by [CHOU85] or to provide multiple buffer pools with different policies. Transactions request pages from the buffer manager and keep them **pinned** to ensure that they are not written to disk while they are in a logically inconsistent state. When page replacement is necessary, the **Buffer Manager** finds an unpinned page and then checks with the **Log Manager** to ensure that the write-ahead protocol is enforced.

3.2.3. The Lock Manager

The **Lock Manager** supports general purpose locking (single writer, multiple readers) which is currently used to provide two-phase locking and high concurrency B-tree locking. However, the general purpose nature of the lock manager provides the ability to support a variety of locking protocols. Currently, all locks are issued at the granularity of a page (the size of a buffer in the buffer pool) which is identified by two 4-byte integers (a file id and page number). This provides the necessary information to extend the **Lock Manager** to perform hierarchical locking [GRAY76]. The current implementation does not support locks at other granularities and does not promote locks; these are obvious future additions to the system.

If an incoming lock request cannot be granted, the requesting process is queued for the lock and descheduled. When a lock is released, the wait queue is traversed and any newly compatible locks are granted. Locks are located via a file and page hash table and are chained both by object and by transaction, facilitating rapid traversal of the lock table during transaction commit and abort.

The primary interfaces to the lock manager are *lock*, *unlock*, and *lock_unlock_all*. *Lock* obtains a new lock for a specific object. There are also two variants of the *lock* request, *lock_upgrade* and *lock_downgrade*, which allow the caller to atomically trade a lock of one type for a lock of another. *Unlock* releases a specific mode of lock on a specific object. *Lock_unlock_all* releases all the locks associated with a specific transaction.

3.2.4. The Process Manager

The **Process Manager** acts as a user-level scheduler to make processes wait on unavailable locks and pending buffer cache I/O. For each process, a semaphore is maintained upon which that process waits when it needs to be descheduled. When a process needs to be run, its semaphore is cleared, and the operating system reschedules it. No sophisticated scheduling algorithm is applied; if the lock for which a process was waiting becomes available, the process is made runnable. It would have been possible to change the kernel's process scheduler to interact more efficiently with the lock manager, but doing so would have compromised our commitment to a user-level package.

3.2.5. The Transaction Manager

The **Transaction Manager** provides the standard interface of *txn_begin*, *txn_commit*, and *txn_abort*. It keeps track of all active transactions, assigns unique transaction identifiers, and directs the abort and commit processing. When a *txn_begin* is issued, the **Transaction Manager** assigns the next available transaction identifier, allocates a per-process transaction structure in shared memory, increments the count of active transactions, and returns the new transaction identifier to the calling process. The in-memory transaction structure contains a pointer into the lock table for locks held by this transaction, the last log sequence number, a transaction state (*idle*, *running*, *aborting*, or *committing)*, an error code, and a semaphore identifier.

At commit, the **Transaction Manager** calls *log_commit* to record the end of transaction and to flush the log. Then it directs the **Lock Manager** to release all locks associated with the given transaction. If a transaction aborts, the **Transaction Manager** calls on *log_unroll* to read the transaction's log records and undo any modifications to the database. As in the commit case, it then calls *lock_unlock_all* to release the transaction's locks.

3.2.6. The Record Manager

The **Record Manager** supports the abstraction of reading and writing records to a database. We have modified the the database access routines **db**(3) [BSD91] to call the log, lock, and buffer managers. In order to provide functionality to perform undo and redo, the **Record Manager** defines a collection of log record types and the associated undo and redo routines. The **Log Manager** performs a table lookup on the record type to call the appropriate routines. For example, the B-tree access method requires two log record types: insert and delete. A replace operation is implemented as a delete followed by an insert and is logged accordingly.

3.3. Application Architectures

The structure of LIBTP allows application designers to trade off performance and protection. Since a large portion of LIBTP's functionality is provided by managing structures in shared memory, its structures are subject to corruption by applications when the library is linked directly with the application. For this reason, LIBTP is designed to allow compilation into a separate server process which may be accessed via a socket interface. In this way LIBTP's data structures are protected from application code, but communication overhead is increased. When applications are trusted, LIBTP may be compiled directly into the application providing improved performance. Figures two and three show the two alternate application architectures.

There are potentially two modes in which one might use LIBTP in a server based architecture. In the first, the server would provide the capability to respond to requests to each of the low level modules (lock, log, buffer, and transaction managers). Unfortunately, the performance of such a system is likely to be blindingly slow since

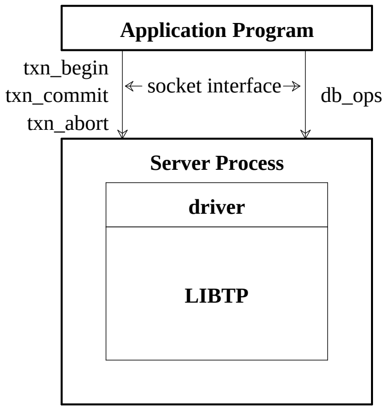

Figure 2: Server Architecture. In this configuration, the library is loaded into a server process which is accessed via a socket interface.

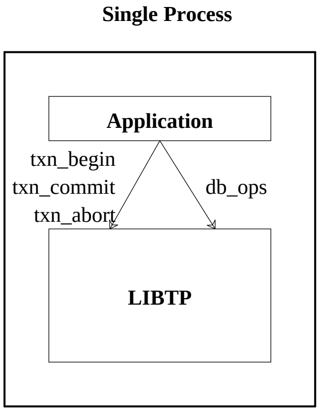

Figure 3: Single Process Architecture. In this configuration, the library routines are loaded as part of the application and accessed via a subroutine interface.

modifying a piece of data would require three or possibly four separate communications: one to lock the data, one to obtain the data, one to log the modification, and possibly one to transmit the modified data. Figure four shows the relative performance for retrieving a single record using the record level call versus using the lower level buffer management and locking calls. The 2:1 ratio observed in the single process case reflects the additional overhead of parsing eight commands rather than one while the 3:1 ratio observed in the client/server architecture reflects both the parsing and the communication overheard. Although there may be applications which could tolerate such performance, it seems far more feasible to support a higher level interface, such as that provided by a query language (*e.g.* SQL [SQL86]).

Although LIBTP does not have an SQL parser, we have built a server application using the toolkit command language (TCL) [OUST90]. The server supports a command line interface similar to the subroutine interface defined in **db**(3). Since it is based on TCL, it provides control structures as well.

4. Implementation

4.1. Locking and Deadlock Detection

LIBTP uses two-phase locking for user data. Strictly speaking, the two phases in two-phase locking are a **grow** phase, during which locks are acquired, and a **shrink** phase, during which locks are released. No lock may ever be acquired during the shrink phase. The grow phase lasts until the first release, which marks the start of the shrink phase. In practice, the grow phase lasts for the duration of a transaction in LIBTP and in commercial database systems. The shrink phase takes place during transaction commit or abort. This means that locks are acquired on demand during the lifetime of a transaction, and held until commit time, at which point all locks are released.

If multiple transactions are active concurrently, deadlocks can occur and must be detected and resolved. The lock table can be thought of as a representation of a directed graph. The nodes in the graph are transactions. Edges represent the **waits-for** relation between transactions; if transaction A is waiting for a lock held by transaction B, then a directed edge exists from A to B in the graph. A deadlock exists if a cycle appears in the graph. By convention, no transaction ever waits for a lock it already holds, so reflexive edges are impossible.

A distinguished process monitors the lock table, searching for cycles. The frequency with which this process runs is user-settable; for the multi-user tests discussed in section 5.1.2, it has been set to wake up every second, but more sophisticated schedules are certainly possible. When a cycle is detected, one of the transactions in the cycle is nominated and aborted. When the transaction aborts, it rolls back its changes and releases its locks, thereby breaking the cycle in the graph.

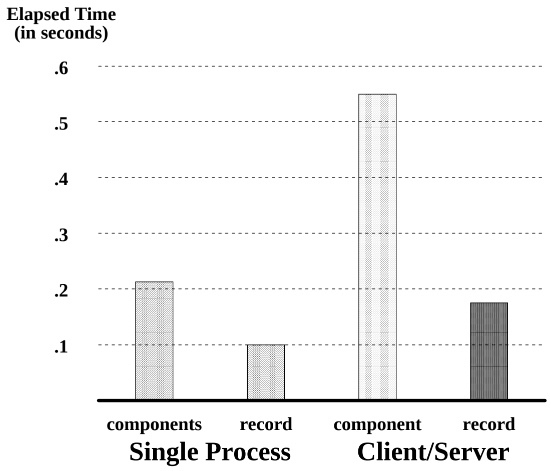

Figure 4: Comparison of High and Low Level Interfaces. Elapsed time in seconds to perform a single record retrieval from a command line (rather than a procedural interface) is shown on the y axis. The ''component'' numbers reflect the timings when the record is retrieved by separate calls to the lock manager and buffer manager while the ''record'' timings were obtained by using a single call to the record manager. The 2:1 ratio observed for the single process case is a reflection of the parsing overhead for executing eight separate commands rather than one. The additional factor of one reflected in the 3:1 ratio for the client/server architecture is due to the communication overhead. The true ratios are actually worse since the component timings do not reflect the search times within each page or the time required to transmit the page between the two processes.

4.2. Group Commit

Since the log must be flushed to disk at commit time, disk bandwidth fundamentally limits the rate at which transactions complete. Since most transactions write only a few small records to the log, the last page of the log will be flushed once by every transaction which writes to it. In the naive implementation, these flushes would happen serially.

LIBTP uses **group commit** [DEWI84] in order to amortize the cost of one synchronous disk write across multiple transactions. Group commit provides a way for a group of transactions to commit simultaneously. The first several transactions to commit write their changes to the in-memory log page, then sleep on a distinguished semaphore. Later, a committing transaction flushes the page to disk, and wakes up all its sleeping peers. The point at which changes are actually written is determined by three thresholds. The first is the *group threshold* and defines the minimum number of transactions which must be active in the system before transactions are forced to participate in a group commit. The second is the *wait threshold* which is expressed as the percentage of active transactions waiting to be committed. The last is the *logdelay threshold* which indicates how much unflushed log should be allowed to accumulate before a waiting transaction's commit record is flushed.

Group commit can substantially improve performance for high-concurrency environments. If only a few transactions are running, it is unlikely to improve things at all. The crossover point is the point at which the transaction commit rate is limited by the bandwidth of the device on which the log resides. If processes are trying to flush the log faster than the log disk can accept data, then group commit will increase the commit rate.

4.3. Kernel Intervention for Synchronization

Since LIBTP uses data in shared memory (*e.g.* the lock table and buffer pool) it must be possible for a process to acquire exclusive access to shared data in order to prevent corruption. In addition, the process manager must put processes to sleep when the lock or buffer they request is in use by some other process. In the LIBTP implementation under Ultrix 4.0[2], we use System V semaphores to provide this synchronization. Semaphores implemented in this fashion turn out to be an expensive choice for synchronization, because each access traps to the kernel and

[2] Ultrix and DEC are trademarks of Digital Equipment Corporation.

executes atomically there.

On architectures that support atomic test-and-set, a much better choice would be to attempt to obtain a spinlock with a test-and-set, and issue a system call only if the spinlock is unavailable. Since virtually all semaphores in LIBTP are uncontested and are held for very short periods of time, this would improve performance. For example, processes must acquire exclusive access to buffer pool metadata in order to find and pin a buffer in shared memory. This semaphore is requested most frequently in LIBTP. However, once it is acquired, only a few instructions must be executed before it is released. On one architecture for which we were able to gather detailed profiling information, the cost of the semaphore calls accounted for 25% of the total time spent updating the metadata. This was fairly consistent across most of the critical sections.

In an attempt to quantify the overhead of kernel synchronization, we ran tests on a version of 4.3BSD-Reno which had been modified to support binary semaphore facilities similar to those described in [POSIX91]. The hardware platform consisted of an HP300 (33MHz MC68030) workstation with 16MBytes of main memory, and a 600MByte HP7959 SCSI disk (17 ms average seek time). We ran three sets of comparisons which are summarized in figure five. In each comparison we ran two tests, one using hardware spinlocks and the other using kernel call synchronization. Since the test was run single-user, none of the the locks were contested. In the first two sets of tests, we ran the full transaction processing benchmark described in section 5.1. In one case we ran with both the database and log on the same disk (1 Disk) and in the second, we ran with the database and log on separate disks (2 Disk). In the last test, we wanted to create a CPU bound environment, so we used a database small enough to fit completely in the cache and issued read-only transactions. The results in figure five express the kernel call synchronization performance as a percentage of the spinlock performance. For example, in the 1 disk case, the kernel call implementation achieved 4.4 TPS (transactions per second) while the semaphore implementation achieved 4.6 TPS, and the relative performance of the kernel synchronization is 96% that of the spinlock (100 * 4.4 / 4.6). There are two striking observations from these results:

- even when the system is disk bound, the CPU cost of synchronization is noticeable, and
- when we are CPU bound, the difference is dramatic (67%).

4.4. Transaction Protected Access Methods

The B-tree and fixed length recno (record number) access methods have been modified to provide transaction protection. Whereas the previously published interface to the access routines had separate open calls for each of the

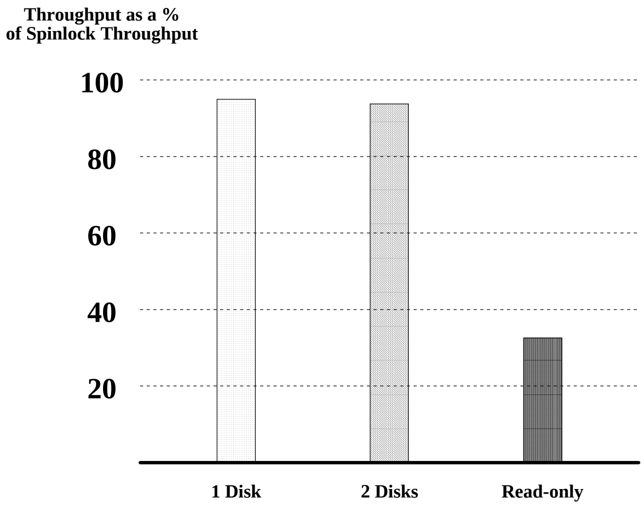

Figure 5: Kernel Overhead for System Call Synchronization. The performance of the kernel call synchronization is expressed as a percentage of the spinlock synchronization performance. In disk bound cases (1 Disk and 2 Disks), we see that 4-6% of the performance is lost due to kernel calls while in the CPU bound case, we have lost 67% of the performance due to kernel calls.

access methods, we now have an integrated open call with the following calling conventions:

```
DB *dbopen (const char *file, int flags, int mode, DBTYPE type,
            int dbflags, const void *openinfo)
```

where *file* is the name of the file being opened, *flags* and *mode* are the standard arguments to **open**(2), *type* is one of the access method types, *dbflags* indicates the mode of the buffer pool and transaction protection, and *openinfo* is the access method specific information. Currently, the possible values for *dbflags* are DB_SHARED and DB_TP indicating that buffers should be kept in a shared buffer pool and that the file should be transaction protected.

The modifications required to add transaction protection to an access method are quite simple and localized.

1. Replace file *open* with *buf_open*.
2. Replace file *read* and *write* calls with buffer manager calls (*buf_get*, *buf_unpin*).
3. Precede buffer manager calls with an appropriate (read or write) lock call.
4. Before updates, issue a logging operation.
5. After data have been accessed, release the buffer manager pin.
6. Provide undo/redo code for each type of log record defined.

The following code fragments show how to transaction protect several updates to a B-tree.[3] In the unprotected case, an open call is followed by a read call to obtain the meta-data for the B-tree. Instead, we issue an open to the buffer manager to obtain a file id and a buffer request to obtain the meta-data as shown below.

```
char *path;
int fid, flags, len, mode;

/* Obtain a file id with which to access the buffer pool */
fid = buf_open(path, flags, mode);

/* Read the meta data (page 0) for the B-tree */
if (tp_lock(fid, 0, READ_LOCK))
      return error;
meta_data_ptr = buf_get(fid, 0, BF_PIN, &len);
```

The BF_PIN argument to *buf_get* indicates that we wish to leave this page pinned in memory so that it is not swapped out while we are accessing it. The last argument to *buf_get* returns the number of bytes on the page that were valid so that the access method may initialize the page if necessary.

Next, consider inserting a record on a particular page of a B-tree. In the unprotected case, we read the page, call *_bt_insertat*, and write the page. Instead, we lock the page, request the buffer, log the change, modify the page, and release the buffer.

```
int fid, len, pageno;   /* Identifies the buffer */
int index;              /* Location at which to insert the new pair */
DBT *keyp, *datap;      /* Key/Data pair to be inserted */
DATUM *d;               /* Key/data structure to insert */

/* Lock and request the buffer */
if (tp_lock(fid, pageno, WRITE_LOCK))
      return error;
buffer_ptr = buf_get(fid, pageno, BF_PIN, &len);

/* Log and perform the update */
log_insdel(BTREE_INSERT, fid, pageno, keyp, datap);
_bt_insertat(buffer_ptr, d, index);
buf_unpin(buffer_ptr);
```

Succinctly, the algorithm for turning unprotected code into protected code is to replace read operations with *lock* and *buf_get* operations and write operations with *log* and *buf_unpin* operations.

---

[3] The following code fragments are examples, but do not define the final interface. The final interface will be determined after LIBTP has been fully integrated with the most recent **db**(3) release from the Computer Systems Research Group at University of California, Berkeley.

5. Performance

In this section, we present the results of two very different benchmarks. The first is an online transaction processing benchmark, similar to the standard TPCB, but has been adapted to run in a desktop environment. The second emulates a computer-aided design environment and provides more complex query processing.

5.1. Transaction Processing Benchmark

For this section, all performance numbers shown except for the commercial database system were obtained on a DECstation 5000/200 with 32MBytes of memory running Ultrix V4.0, accessing a DEC RZ57 1GByte disk drive. The commercial relational database system tests were run on a comparable machine, a Sparcstation 1+ with 32MBytes memory and a 1GByte external disk drive. The database, binaries and log resided on the same device. Reported times are the means of five tests and have standard deviations within two percent of the mean.

The test database was configured according to the TPCB scaling rules for a 10 transaction per second (TPS) system with 1,000,000 account records, 100 teller records, and 10 branch records. Where TPS numbers are reported, we are running a modified version of the industry standard transaction processing benchmark, TPCB. The TPCB benchmark simulates a withdrawal performed by a hypothetical teller at a hypothetical bank. The database consists of relations (files) for accounts, branches, tellers, and history. For each transaction, the account, teller, and branch balances must be updated to reflect the withdrawal and a history record is written which contains the account id, branch id, teller id, and the amount of the withdrawal [TPCB90].

Our implementation of the benchmark differs from the specification in several aspects. The specification requires that the database keep redundant logs on different devices, but we use a single log. Furthermore, all tests were run on a single, centralized system so there is no notion of remote accesses. Finally, we calculated throughput by dividing the total elapsed time by the number of transactions processed rather than by computing the response time for each transaction.

The performance comparisons focus on traditional Unix techniques (unprotected, using **flock**(2) and using **fsync**(2)) and a commercial relational database system. Well-behaved applications using **flock**(2) are guaranteed that concurrent processes' updates do not interact with one another, but no guarantees about atomicity are made. That is, if the system crashes in mid-transaction, only parts of that transaction will be reflected in the after-crash state of the database. The use of **fsync**(2) at transaction commit time provides guarantees of durability after system failure. However, there is no mechanism to perform transaction abort.

5.1.1. Single-User Tests

These tests compare LIBTP in a variety of configurations to traditional UNIX solutions and a commercial relational database system (RDBMS). To demonstrate the server architecture we built a front end test process that uses TCL [OUST90] to parse database access commands and call the database access routines. In one case (SERVER), frontend and backend processes were created which communicated via an IP socket. In the second case (TCL), a single process read queries from standard input, parsed them, and called the database access routines. The performance difference between the TCL and SERVER tests quantifies the communication overhead of the socket. The RDBMS implementation used embedded SQL in C with stored database procedures. Therefore, its configuration is a hybrid of the single process architecture and the server architecture. The graph in figure six shows a comparison of the following six configurations:

LIBTP	Uses the LIBTP library in a single application.
TCL	Uses the LIBTP library in a single application, requires query parsing.
SERVER	Uses the LIBTP library in a server configuration, requires query parsing.
NOTP	Uses no locking, logging, or concurrency control.
FLOCK	Uses **flock**(2) for concurrency control and nothing for durability.
FSYNC	Uses **fsync**(2) for durability and nothing for concurrency control.
RDBMS	Uses a commercial relational database system.

The results show that LIBTP, both in the procedural and parsed environments, is competitive with a commercial system (comparing LIBTP, TCL, and RDBMS). Compared to existing UNIX solutions, LIBTP is approximately 15% slower than using **flock**(2) or no protection but over 80% better than using **fsync**(2) (comparing LIBTP, FLOCK, NOTP, and FSYNC).

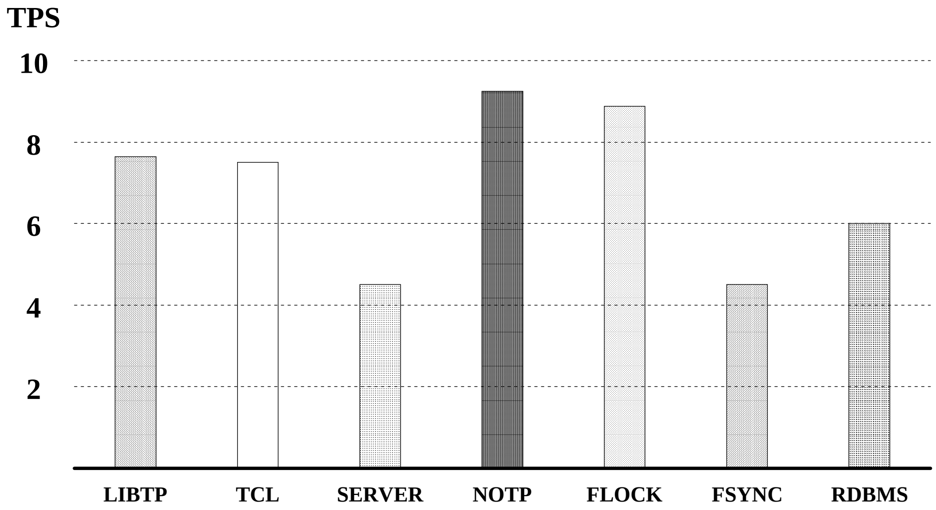

Figure 6: Single-User Performance Comparison.

5.1.2. Multi-User Tests

While the single-user tests form a basis for comparing LIBTP to other systems, our goal in multi-user testing was to analyze its scalability. To this end, we have run the benchmark in three modes, the normal disk bound configuration (figure seven), a CPU bound configuration (figure eight, READ-ONLY), and lock contention bound (figure eight, NO_FSYNC). Since the normal configuration is completely disk bound (each transaction requires a random read, a random write, and a sequential write[4]) we expect to see little performance improvement as the multiprogramming level increases. In fact, figure seven reveals that we are able to overlap CPU and disk utilization slightly producing approximately a 10% performance improvement with two processes. After that point, performance drops off, and at a multi-programming level of 4, we are performing worse than in the single process case.

Similar behavior was reported on the commercial relational database system using the same configuration. The important conclusion to draw from this is that you cannot attain good multi-user scaling on a badly balanced system. If multi-user performance on applications of this sort is important, one must have a separate logging device and horizontally partition the database to allow a sufficiently high degree of multiprogramming that group commit can amortize the cost of log flushing.

By using a very small database (one that can be entirely cached in main memory) and read-only transactions, we generated a CPU bound environment. By using the same small database, the complete TPCB transaction, and no **fsync**(2) on the log at commit, we created a lock contention bound environment. The small database used an account file containing only 1000 records rather than the full 1,000,000 records and ran enough transactions to read the entire database into the buffer pool (2000) before beginning measurements. The read-only transaction consisted of three database reads (from the 1000 record account file, the 100 record teller file, and the 10 record branch file). Since no data were modified and no history records were written, no log records were written. For the contention bound configuration, we used the normal TPCB transaction (against the small database) and disabled the log flush. Figure eight shows both of these results.

The read-only test indicates that we barely scale at all in the CPU bound case. The explanation for that is that even with a single process, we are able to drive the CPU utilization to 96%. As a result, that gives us very little room for improvement, and it takes a multiprogramming level of four to approach 100% CPU saturation. In the case where we do perform writes, we are interested in detecting when lock contention becomes a dominant performance factor. Contention will cause two phenomena; we will see transactions queueing behind frequently accessed data, and we will see transaction abort rates increasing due to deadlock. Given that the branch file contains only ten

[4] Although the log is written sequentially, we do not get the benefit of sequentiality since the log and database reside on the same disk.

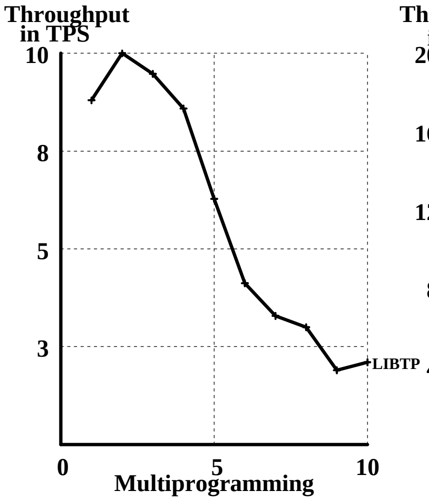

Figure 7: Multi-user Performance. Since the configuration is completely disk bound, we see only a small improvement by adding a second process. Adding any more concurrent processes causes performance degradation.

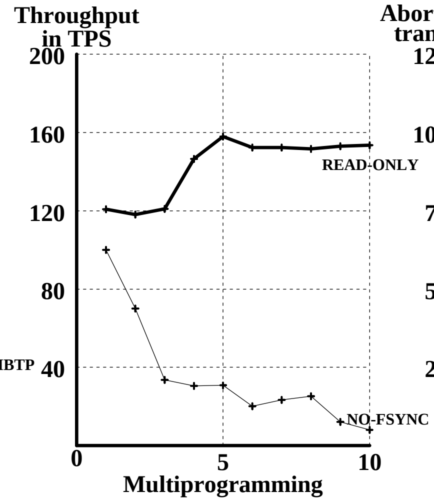

Figure 8: Multi-user Performance on a small database. With one process, we are driving the CPU at 96% utilization leaving little room for improvement as the multiprogramming level increases. In the NO-FSYNC case, lock contention degrades performance as soon as a second process is added.

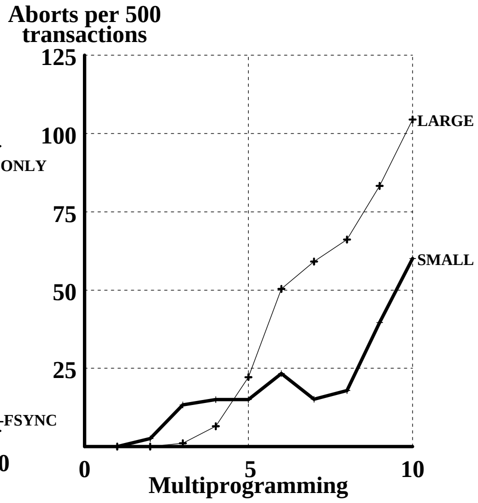

Figure 9: Abort rates on the TPCB Benchmark. The abort rate climbs more quickly for the large database test since processes are descheduled more frequently, allowing more processes to vie for the same locks.

---

records, we expect contention to become a factor quickly and the NO-FSYNC line in figure eight demonstrates this dramatically. Each additional process causes both more waiting and more deadlocking. Figure nine shows that in the small database case (SMALL), waiting is the dominant cause of declining performance (the number of aborts increases less steeply than the performance drops off in figure eight), while in the large database case (LARGE), deadlocking contributes more to the declining performance.

Deadlocks are more likely to occur in the LARGE test than in the SMALL test because there are more opportunities to wait. In the SMALL case, processes never do I/O and are less likely to be descheduled during a transaction. In the LARGE case, processes will frequently be descheduled since they have to perform I/O. This provides a window where a second process can request locks on already locked pages, thus increasing the likelihood of building up long chains of waiting processes. Eventually, this leads to deadlock.

5.2. The OO1 Benchmark

The TPCB benchmark described in the previous section measures performance under a conventional transaction processing workload. Other application domains, such as computer-aided design, have substantially different access patterns. In order to measure the performance of LIBTP under workloads of this type, we implemented the OO1 benchmark described in [CATT91].

The database models a set of electronics components with connections among them. One table stores parts and another stores connections. There are three connections originating at any given part. Ninety percent of these connections are to nearby parts (those with nearby *ids*) to model the spatial locality often exhibited in CAD applications. Ten percent of the connections are randomly distributed among all other parts in the database. Every part appears exactly three times in the *from* field of a connection record, and zero or more times in the *to* field. Parts have *x* and *y* locations set randomly in an appropriate range.

The intent of OO1 is to measure the overall cost of a query mix characteristic of engineering database applications. There are three tests:

- *Lookup* generates 1,000 random part *ids*, fetches the corresponding parts from the database, and calls a null procedure in the host programming language with the parts' *x* and *y* positions.
- *Traverse* retrieves a random part from the database and follows connections from it to other parts. Each of those parts is retrieved, and all connections from it followed. This procedure is repeated depth-first for seven hops from the original part, for a total of 3280 parts. Backward traversal also exists, and follows all connections into a given part to their origin.
- *Insert* adds 100 new parts and their connections.

The benchmark is single-user, but multi-user access controls (locking and transaction protection) must be enforced. It is designed to be run on a database with 20,000 parts, and on one with 200,000 parts. Because we have insufficient disk space for the larger database, we report results only for the 20,000 part database.

5.2.1. Implementation

The LIBTP implementation of OO1 uses the TCL [OUST90] interface described earlier. The backend accepts commands over an IP socket and performs the requested database actions. The frontend opens and executes a TCL script. This script contains database accesses interleaved with ordinary program control statements. Database commands are submitted to the backend and results are bound to program variables.

The parts table was stored as a B-tree indexed by *id*. The connection table was stored as a set of fixed-length records using the 4.4BSD recno access method. In addition, two B-tree indices were maintained on connection table entries. One index mapped the *from* field to a connection record number, and the other mapped the *to* field to a connection record number. These indices support fast lookups on connections in both directions. For the traversal tests, the frontend does an index lookup to discover the connected part's *id*, and then does another lookup to fetch the part itself.

5.2.2. Performance Measurements for OO1

We compare LIBTP's OO1 performance to that reported in [CATT91]. Those results were collected on a Sun 3/280 (25 MHz MC68020) with 16 MBytes of memory and two Hitachi 892MByte disks (15 ms average seek time) behind an SMD-4 controller. Frontends ran on an 8MByte Sun 3/260.

In order to measure performance on a machine of roughly equivalent processor power, we ran one set of tests on a standalone MC68030-based HP300 (33MHz MC68030). The database was stored on a 300MByte HP7959 SCSI disk (17 ms average seek time). Since this machine is not connected to a network, we ran local tests where the frontend and backend run on the same machine. We compare these measurements with Cattell's local Sun 3/280 numbers.

Because the benchmark requires remote access, we ran another set of tests on a DECstation 5000/200 with 32M of memory running Ultrix V4.0 and a DEC 1GByte RZ57 SCSI disk. We compare the local performance of OO1 on the DECstation to its remote performance. For the remote case, we ran the frontend on a DECstation 3100 with 16 MBytes of main memory.

The databases tested in [CATT91] are

- INDEX, a highly-optimized access method package developed at Sun Microsystems.
- OODBMS, a beta release of a commercial object-oriented database management system.
- RDBMS, a UNIX-based commercial relational data manager at production release. The OO1 implementation used embedded SQL in C. Stored procedures were defined to reduce client-server traffic.

Table two shows the measurements from [CATT91] and LIBTP for a local test on the MC680x0-based hardware. All caches are cleared before each test. All times are in seconds.

Table two shows that LIBTP outperforms the commercial relational system, but is slower than OODBMS and INDEX. Since the caches were cleared at the start of each test, disk throughput is critical in this test. The single SCSI HP drive used by LIBTP is approximately 13% slower than the disks used in [CATT91] which accounts for part of the difference.

OODBMS and INDEX outperform LIBTP most dramatically on traversal. This is because we use index lookups to find connections, whereas the other two systems use a link access method. The index requires us to examine

Measure	INDEX	OODBMS	RDBMS	LIBTP
Lookup	5.4	12.9	27	27.2
Traversal	13	9.8	90	47.3
Insert	7.4	1.5	22	9.7

Table 2: Local MC680x0 Performance of Several Systems on OO1.

Measure	*Cache*	*Local*	*Remote*
Lookup	cold	15.7	20.6
	warm	7.8	12.4
Forward traversal	cold	28.4	52.6
	warm	23.5	47.4
Backward traversal	cold	24.2	47.4
	warm	24.3	47.6
Insert	cold	7.5	10.3
	warm	6.7	10.9

Table 3: Local vs. Remote Performance of LIBTP on OO1.

two disk pages, but the links require only one, regardless of database size. Cattell reports that lookups using B-trees instead of links makes traversal take twice as long in INDEX. Adding a link access method to **db**(3) or using the existing hash method would apparently be a good idea.

Both OODBMS and INDEX issue coarser-granularity locks than LIBTP. This limits concurrency for multi-user applications, but helps single-user applications. In addition, the fact that LIBTP releases B-tree locks early is a drawback in OO1. Since there is no concurrency in the benchmark, high-concurrency strategies only show up as increased locking overhead. Finally, the architecture of the LIBTP implementation was substantially different from that of either OODBMS or INDEX. Both of those systems do the searches in the user's address space, and issue requests for pages to the server process. Pages are cached in the client, and many queries can be satisfied without contacting the server at all. LIBTP submits all the queries to the server process, and receives database records back; it does no client caching.

The RDBMS architecture is much closer to that of LIBTP. A server process receives queries and returns results to a client. The timing results in table two clearly show that the conventional database client/server model is expensive. LIBTP outperforms the RDBMS on traversal and insertion. We speculate that this is due in part to the overhead of query parsing, optimization, and repeated interpretation of the plan tree in the RDBMS' query executor.

Table three shows the differences between local and remote execution of LIBTP's OO1 implementation on a DECstation. We measured performance with a populated (warm) cache and an empty (cold) cache. Reported times are the means of twenty tests, and are in seconds. Standard deviations were within seven percent of the mean for remote, and two percent of the mean for local.

The 20ms overhead of TCP/IP on an Ethernet entirely accounts for the difference in speed. The remote traversal times are nearly double the local times because we do index lookups and part fetches in separate queries. It would make sense to do indexed searches on the server, but we were unwilling to hard-code knowledge of OO1 indices into our LIBTP TCL server. Cold and warm insertion times are identical since insertions do not benefit from caching.

One interesting difference shown by table three is the cost of forward versus backward traversal. When we built the database, we inserted parts in part *id* order. We built the indices at the same time. Therefore, the forward index had keys inserted in order, while the backward index had keys inserted more randomly. In-order insertion is pessimal for B-tree indices, so the forward index is much larger than the backward one[5]. This larger size shows up as extra disk reads in the cold benchmark.

6. Conclusions

LIBTP provides the basic building blocks to support transaction protection. In comparison with traditional Unix libraries and commercial systems, it offers a variety of tradeoffs. Using complete transaction protection is more complicated than simply adding **fsync**(2) and **flock**(2) calls to code, but it is faster in some cases and offers

[5] The next release of the 4.4BSD access method will automatically detect and compensate for in-order insertion, eliminating this problem.

stricter guarantees (atomicity, consistency, isolation, and durability). If the data to be protected are already formatted (*i.e.* use one of the database access methods), then adding transaction protection requires no additional complexity, but incurs a performance penalty of approximately 15%.

In comparison with commercial database systems, the tradeoffs are more complex. LIBTP does not currently support a standard query language. The TCL-based server process allows a certain ease of use which would be enhanced with a more user-friendly interface (*e.g.* a windows based query-by-form application), for which we have a working prototype. When accesses do not require sophisticated query processing, the TCL interface is an adequate solution. What LIBTP fails to provide in functionality, it makes up for in performance and flexibility. Any application may make use of its record interface or the more primitive log, lock, and buffer calls.

Future work will focus on overcoming some of the areas in which LIBTP is currently deficient and extending its transaction model. The addition of an SQL parser and forms front end will improve the system's ease of use and make it more competitive with commercial systems. In the long term, we would like to add generalized hierarchical locking, nested transactions, parallel transactions, passing of transactions between processes, and distributed commit handling. In the short term, the next step is to integrate LIBTP with the most recent release of the database access routines and make it freely available via anonymous ftp.

7. Acknowledgements

We would like to thank John Wilkes and Carl Staelin of Hewlett-Packard Laboratories and Jon Krueger. John and Carl provided us with an extra disk for the HP testbed less than 24 hours after we requested it. Jon spent countless hours helping us understand the intricacies of commercial database products and their behavior under a variety of system configurations.

8. References

[ANDR89] Andrade, J., Carges, M., Kovach, K., ‘‘Building an On-Line Transaction Processing System On UNIX System V’’, *CommUNIXations*, November/December 1989.

[BAY77] Bayer, R., Schkolnick, M., ‘‘Concurrency of Operations on B-Trees’’, *Acta Informatica*, 1977.

[BERN80] Bernstein, P., Goodman, N., ‘‘Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems’’, *Proceedings 6th International Conference on Very Large Data Bases*, October 1980.

[BSD91] DB(3), *4.4BSD Unix Programmer's Manual Reference Guide*, University of California, Berkeley, 1991.

[CATT91] Cattell, R.G.G., ‘‘An Engineering Database Benchmark’’, *The Benchmark Handbook for Database and Transaction Processing Systems*, J. Gray, editor, Morgan Kaufman 1991.

[CHEN91] Cheng, E., Chang, E., Klein, J., Lee, D., Lu, E., Lutgardo, A., Obermarck, R., ‘‘An Open and Extensible Event-Based Transaction Manager’’, *Proceedings 1991 Summer Usenix*, Nashville, TN, June 1991.

[CHOU85] Chou, H., DeWitt, D., ‘‘An Evaluation of Buffer Management Strategies for Relational Database Systems’’, *Proceedings of the 11th International Conference on Very Large Databases*, 1985.

[DEWI84] DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., Wood, D., ‘‘Implementation Techniques for Main Memory Database Systems’’, *Proceedings of SIGMOD*, pp. 1-8, June 1984.

[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.

[KUNG81] Kung, H. T., Richardson, J., ‘‘On Optimistic Methods for Concurrency Control’’, *ACM Transactions on Database Systems* 6(2); 213-226, 1981.

[LEHM81] Lehman, P., Yao, S., ‘‘Efficient Locking for Concurrent Operations on B-trees’’, *ACM Transactions on Database Systems*, 6(4), December 1981.

[MOHA91] Mohan, C., Pirahesh, H., ‘‘ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method’’, *Proceedings 7th International Conference on Data Engineering*, Kobe, Japan, April 1991.

[NODI90] Nodine, M., Zdonik, S., ‘‘Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications’’, *Proceedings 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[OUST90] Ousterhout, J., ‘‘Tcl: An Embeddable Command Language’’, *Proceedings 1990 Winter Usenix*, Washington, D.C., January 1990.

[POSIX91] ‘‘Unapproved Draft for Realtime Extension for Portable Operating Systems’’, Draft 11, October 7, 1991, IEEE Computer Society.

[ROSE91] Rosenblum, M., Ousterhout, J., ‘‘The Design and Implementation of a Log-Structured File System’’, *Proceedings of the 13th Symposium on Operating Systems Principles*, 1991.

[SELT91] Seltzer, M., Stonebraker, M., ‘‘Read Optimized File Systems: A Performance Evaluation’’, *Proceedings 7th Annual International Conference on Data Engineering*, Kobe, Japan, April 1991.

[SPEC88] Spector, Rausch, Bruell, ‘‘Camelot: A Flexible, Distributed Transaction Processing System’’, *Proceedings of Spring COMPCON 1988*, February 1988.

[SQL86] American National Standards Institute, ‘‘Database Language SQL’’, ANSI X3.135-1986 (ISO 9075), May 1986.

[STON81] Stonebraker, M., ‘‘Operating System Support for Database Management’’, *Communications of the ACM*, 1981.

[SULL92] Sullivan, M., Olson, M., ‘‘An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System’’, to appear in *Proceedings 8th Annual International Conference on Data Engineering*, Tempe, Arizona, February 1992.

[TPCB90] Transaction Processing Performance Council, ‘‘TPC Benchmark B’’, Standard Specification, Waterside Associates, Fremont, CA., 1990.

[YOUN91] Young, M. W., Thompson, D. S., Jaffe, E., ‘‘A Modular Architecture for Distributed Transaction Processing’’, *Proceedings 1991 Winter Usenix*, Dallas, TX, January 1991.

Margo I. Seltzer is a Ph.D. student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Her research interests include file systems, databases, and transaction processing systems. She spent several years working at startup companies designing and implementing file systems and transaction processing software and designing microprocessors. Ms. Seltzer received her AB in Applied Mathematics from Harvard/Radcliffe College in 1983.

In her spare time, Margo can usually be found preparing massive quantities of food for hungry hordes, studying Japanese, or playing soccer with an exciting Bay Area Women’s Soccer team, the Berkeley Bruisers.

Michael A. Olson is a Master’s student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His primary interests are database systems and mass storage systems. Mike spent two years working for a commercial database system vendor before joining the Postgres Research Group at Berkeley in 1988. He received his B.A. in Computer Science from Berkeley in May 1991.

Mike only recently transferred into Sin City, but is rapidly adopting local customs and coloration. In his spare time, he organizes informal Friday afternoon study groups to discuss recent technical and economic developments. Among his hobbies are Charles Dickens, Red Rock, and speaking Dutch to anyone who will permit it.