Making System Calls Transaction Safe

Neelam Goyal Dheeraj Agrawal

University of Wisconsin, Madison.
{neelam, dheeraj}@cs.wisc.edu.

ABSTRACT

The advent of multi-core processors has renewed interest in the idea of incorporating transactions into the programming model used to write multi-threaded programs. However, using lock based multi-threaded programming is hard to get right and easy to deadlock. Transactional Memory (TM) simplifies multi-threaded programming, by replacing use of locks with transactions. But, there are many problems that TM system development is facing today. Firstly, *there are limited TM workloads, since TM systems existing today are in a preliminary stage.* Secondly, *TM systems cannot handle system calls inside transactions as of today.*

In this project, we look at TM related issues on two important server workloads - AOLserver and BIND. Our focus is primarily on network send()/recv() calls and their variants. We propose using a cross-layer mechanism which leverages both kernel and user level support to provide an easy and minimum performance overhead solution to handle network calls. From our current results we see upto 3X overhead when send()/recv() are used for sending/receiving large packets (e.g. 1400 bytes). But, we believe that can be brought down to a negligible level using simple optimizations.

Finally, we also handle conditional sychronization in transactions which can potentially lead to a deadlock. We use a simple busy-wait approach explored later in the paper.

1. INTRODUCTION

The advent of multi-core processors has renewed interest in the idea of incorporating transactions into the programming model used to write multi-threaded programs. It's a well known fact that multi-threaded programming is hard to get right and easy to deadlock. Transactional Memory (TM) simplifies multi-threaded programming by removing the need to associate a lock to get exclusive access to shared data. Instead, programmers specify transactions, which forces a set of statements to execute atomically.

Many TM systems have been proposed so far. But as it stands today, there are many limitations that still require significant attention for making TM a potential programming model for multi-core processors.

One of the key limitations in TM systems today is that they cannot support system calls which change the state of a transaction in such a way that it is either difficult to roll it back or even worse, the system calls make the changes irreversible. For e.g. system calls inside a transaction that write to the video output, allocate memory, send/receive packets from network, cannot be undone if the corresponding transaction has to be aborted. We categorize such system calls as system calls with *side effects*. Such system calls are currently prohibited inside transactions which certainly questions the applicability of transactions in near future.

The second key limitation to TM research is the lack of suitable workloads. Many of the challenges of architecting TM systems derive from a lack of understanding of how programmers will use such systems. In this respect, research in TM systems would be greatly facilitated by the public availability of a set of workloads that were designed with transactional memory in mind. Developing a collection of workloads could be achieved through a community effort, but a number of challenges remain in realizing this goal.

Motivated by such needs we explore some large server workloads viz. BIND and AOLserver which can greatly benefit in terms of both performance and ease of programming if they are written in the transaction memory paradigm. BIND (Berkeley Internet Name Domain),an implementation of the Domain Name System (DNS) protocols is a large, concurrent software that uses a large number of locks while processing a DNS request. AOLserver is a multithreaded, Tcl-enabled web server that uses locks extensively while processing requests. The purpose of choosing server workloads for this work is twofold.

We have chosen the two workloads because both BIND and AOLserver are multithreaded servers having interesting usage of locks. Thus they are potential workloads to transactionalize since they give us ample opportunity to convert lock based critical sections to transactions. In particular, we explore two aspects of transactional memory in these workloads, *support for system calls in transactions* and change in the programming paradigm due to conversion from locks to transactions.

Contributions: The following are the contributions of the paper

Providing support for system calls inside transactions. While prior work has resulted in attempts at universal, one-size-fits-all, solution to the problem of system calls invoked in transactions, such as deferring such actions until the transaction is certain to commit [1, 2] or placing them in completion actions [3], forcing such transactions to go non-speculative [4, 1] is either too restrictive or too inefficient. System calls that effect global state do so in such drastically different ways that no global solution until now is the best solution. As an alternative, we propose a hybrid approach which involves commit and compensation actions [5] at user level and kernel level support in certain scenarios, in order to balance the trade-offs between semantic restrictions and reduced performance.

Implementing user level support though simple is not sufficient for the complete class of system calls. Since every system call has a different side effect, we need kernel level support for more complex system calls such as network calls. In this work, we provide user level support of file system calls by changing wrappers in the C library and kernel level support for network calls by introducing a pseudo network driver.

Transactionalizing BIND and AOLserver. To analyze the impact on the programming paradigm due to replacing use of locks with transactions, we transactionalize parts of the two server workloads, BIND and AOLserver. We use Wisconsin GEMS with support for LogTM HTM for simulating the transactionalized versions of the workloads.

In the next section 2, we review the problems raised by system calls when invoked inside transactions. In section 3, we focus on the important system calls to support in transactions and some issues with conditional synchronization in transaction. We then discuss our approach for handling non-network related system calls and conditional synchronization in section 4. In section 6, we present a full-fledged strategy to support TCP/UDP network calls inside transactions. Finally we present our evaluation platform and the results in section 8.

2. MOTIVATION

While transactional memory promises to ease the burden on programmers writing multi-threaded applications, it introduces a variety of unresolved problems in real concurrent systems. A major source of these problems are system calls that can be made within a transaction. TM supports speculative access of shared data which causes no *side-effects*, effects which essentially change the logical state of the system in such a way which often can't be easily rolled back or, in certain cases is completely irreversible. We identify two main issues with system calls when invoked inside transactions, as mentioned below.

Aborts. To fulfill the first and foremost requirement of TM, which is atomicity, we should allow transactions to abort in case of a conflict. Allowing aborts demands complete rollback of the state touched by the aborting transaction.

System call actions such as I/O (including disk and network), memory allocation etc. are not only hard to rollback, it's almost impossible to revert them back. A packet sent to a remote machine in case of a send() system call, for instance can't be brought back without effecting the state of remote machine. To avoid this, we could disallow the transactions invoking system calls to abort. Though semantically correct, this is an extremely restrictive solution.

Isolation violation. Second equally important requirement of TM, which is isolation, is almost always violated whenever I/O is done inside a transaction. For example, when a transaction which has sent a packet to a remote machine is aborted, the packet has still been sent to the remote machine.

Though mentioned separately, aborts and isolation guarantee are not completely mutually exclusive. We might be able to abort but without guaranteeing isolation. To provide isolation, we could retain the use of locks. But even this might not be possible in certain case such as network calls where the side effects are not local to the machine.

3. METHODOLOGY

To understand how transactions in AOLserver and BIND compare over the corresponding lock based versions both in terms of performance and ease of programming, we transactionalized only the most frequently accessed code paths in these workloads. For identifying such code paths, we profile AOLserver and BIND using Dtrace [6], a dynamic instrumentation tool from Sun.

3.1 Observations

System calls. We identify three categories of system calls in AOLserver and BIND which get invoked in critical sections. The first category includes system calls with no side-effects e.g. getpid(). The second category comprises of system calls with side-effects but can be easily handled by using escape actions which we discuss in 5, e.g. file system calls. The final category includes system calls with side-effects but which *cannot* be easily handled by simply using escape actions, e.g. network calls, fork() etc. In this project, we address file system calls and network calls which form part of the last two categories. Table 3.1 also shows other system calls such as doorfs() for inter-process communication and ioctl() which we don't currently address. In addition system calls with no side effects need no special consideration as they don't effect system state.

Conditional Synchronization We also observe scenarios of conditional synchronization using condition variables in both the workloads. Condition variables are synchronization objects that allow threads to wait for certain events (conditions) to occur. The protocol for using condition variables includes

Category	System Calls
System calls with no side-effects	getpid()
File system calls	open(), read(), write(), close(), fflush(), fdsync(), lseek()
Network calls	send(), recv()
Other	ioctl(), doorfs()

a mutex, a boolean predicate (true/false expression) and the condition variable itself. The threads that are cooperating using condition variables can wait for a condition to occur, or can wake up other threads that are waiting for a condition.

This kind of synchronization is often not supported by TM systems, as many of them don't allow thread suspension inside a transaction. For the systems which allow thread suspension, conditional synchronization still suffers issues such as deadlock. If a transactional thread is waiting for a condition to occur by checking the value of condition variable, the variable gets added to the read set of that transaction.

Now when a thread wants to write to that condition variable to wake up the other waiting threads (including the transactional thread), the transactional thread will conflict with the write operation. In case of LogTM with requester-stalls conflict resolution policy, the writer thread will keep on stalling and the transactional thread will be blocked waiting on the condition variable. This leads to a deadlock.

4. APPROACH

We propose supporting system calls at two levels (i) User level (ii) Kernel Level. User level support is easier to implement but is less flexible at the same time as compared to the kernel level support. Kernel level support though harder to implement is more powerful and might be necessary in certain scenarios, such as network calls as we discuss later.

4.1 User Level - in libc

We borrow the notion of ESCAPE ACTIONS from LogTM [7]. The idea is to execute system code such as compiler run time and OS kernel code outside the transactional scope. These actions are not transactional, but allow the current transaction to continue after they complete. In a LogTM system, it is undesirable for the operating system kernel to participate in transactions with user-level applications.

First, doing so would cede control over locking of kernel memory to user-level code, which would violate the isolation between the kernel and applications. Second, the undo log for LogTM is stored at user-level, and it would be similarly unsafe to rely on user-level data for the correctness of kernel data structures. Finally, the abort processing code executes at user-level and is not able to restore changes to kernel data structures.

Since the escape actions are executed outside transactional scope, their effects are visible immediately to the outside world thus breaking isolation. In order to preserve isolation, user-level sentinel can be used for calls where the side effect is only visible within a single process. Before making a system call that modifies per-process state, a user-level thread would modify this sentinel.

As a result, any other thread that attempted to make the same call within a transaction would detect a conflict before attempting the system call. Similarly, any call that reads kernel state should read a sentinel that will detect conflicts with attempts to modify that state.

In addition, in event of an abort, the side-effects of the code executed in escape actions should be explicitly rolled back using compensating actions. Compensating actions are registered when invoking system calls and are provided with information required for rolling the effects back. On an abort compensating actions are invoked for the roll back. In certain scenarios delaying the system call till commit time serves as a plausible solution. In such cases, commit action is registered and is invoked when committing the transaction.

In this work, we provide support for escape actions with compensating and commit actions in C library. We essentially make changes to the system call wrappers in libc.

4.2 Kernel Level - for network calls

Kernel level support is necessary for more complex system calls which can't be easily rolled back with compensating actions using the information available at user level or which are not suitable for delaying till commit. We identify network calls as potential candidates which could leverage kernel support. We discuss these calls in detail in section 6.

5. HANDLING FILE SYSTEM CALLS

Since, all system calls go via the libc interface, to take transaction specific actions for such system calls, we created wrappers around existing system call interfaces in libc.

In our approach, when executing a system call in libc we first check whether the call was made within a transaction scope. If the call has been invoked outside a transaction, the wrapper behavior remains unchanged, otherwise appropriate actions are taken depending on the system call which may involve registering a commit action, registering a compensating action, storing some meta data which might be used in the commit/compensating actions and escaping the transaction.

Below is the description for the system calls we have changed used in the workloads:

write(int fildes, const void *buf, size_t nbyte)

We specify no compensation for stdin, stdout, stderr. For regular file, we first escape the transaction, and store the file descriptor and the initial location of the file pointer. If the file

already has some data, data upto nbytes is read and stored in a buffer. A write system call is then invoked and if this call succeeds, we store the number of bytes written and the original file size. After storing all the essential meta data, a compensating action is registered which gets invoked if the transaction aborts. The compensating actions, restores the file to its original state which involves restoring the file pointer and any overwritten data (if the file had some content originally).

pwrite(int fildes, const void *buf, size_t nbyte, off_t offset)

The pwrite() function performs the same action as write(), except that it writes into a given position (offset) without changing the file pointer. So the approach followed for pwrite() is similar to that for write() expect for a few changes. pwrite() is not permitted for stdin, stdout, stderr, so we don't need to check for them explicitly. Instead of storing the original file pointer (which isn't changed in pwrite()) we store the offset which helps restoring (during compensation) if any original data is overwritten by the pwrite() function at correct location which is specified by offset.

pread(int fildes, void *buf, size_t nbyte, off_t offset)

The pread() function performs the same action as read(), except that it reads from a given position (offset) without changing the file pointer. Since the state of file is completely unchanged in this case, we don't need any compensation here. The system call just escapes the transaction.

lseek(int fildes, off_t offset, int whence)

Since the side effect of lseek system call is similar to read system call, (they both change the file pointer). we deal with lseek exactly as we deal with read system call. i.e. store the file descriptor and original file pointer and execute the lseek system call inside an escape action. A compensating action is registered if the call succeeds which restores the original file pointer when invoked.

close(int fildes)

The approach to this system call is to delay the closing of file till the transaction commits. For this a commit handler is registered inside the wrapper and the commit handler just closes the file when the transaction commits.

open(const char *path, int oflag, mode_t mode)

open system call either creates a new file or opens an existing file. We check in the wrapper if the file already exists and store that information along with the file descriptor. A compensating action is registered which either closes the file (if the file already existed) or deletes the file (if the file was created during the transaction).

fdsync(int fd, int flag)

This system call is handled by delaying it till commit time. A commit handler is registered inside the wrapper in libc and the commit handler is responsible for invoking the fdsync() on transaction commit.

brk(void *new_brk)

The brk() function sets the break value to new_brk and changes the allocated space accordingly. We store the old break value and invoke the brk system call. If the call succeeds, we register a compensating action which restores the old break value when invoked.

sbrk(intptr_t addend)

The sbrk() function adds addend function bytes to the break value and changes the allocated space accordingly. This is dealt in a manner similar to brk. The old break value is stored and the compensating action uses that value to restore the old value when invoked.

To summarize, at user level, we handle system calls by registering commit and compensating actions inside libc wrappers. In most of the cases, the commit action is used when the system call is delayed till commit (we call them lazy calls). Compensating actions are invoked in libc wrappers when the corresponding system call is invoked immediately (we call them eager calls) and the effects are undone inside compensating actions in case of an abort.

In case of lazy calls, on an abort, the calls are simply dropped while in case of eager calls, the actions are explicitly rolled back by the compensating actions in case of an abort. Isolation among transactional and non-transactional thread is provided by user level sentinels.

6. HANDLING NETWORK CALLS

One of the most difficult aspects of TM systems is handling I/O calls which are irreversible, e.g. printing, network I/O. The problem is that executing such I/O calls immediately breaks the isolation between the running transaction and the outside world (printer, other networks).

However, network calls are a unique class of I/O calls which can be allowed inside transactions by exposing the knowledge of underlying network protocol (e.g. TCP/UDP) to transaction manager. Though this does not permit the transaction manager to reverse the effect of network calls, by using protocol knowledge, the manager can take appropriate action (maybe by making more network calls) to restore the state of the system (and the outside world) to the state previous to making the network call.

There has been some initial work in this regard. Haskin et. al, in their system QuickSilver [8], use a distributed commit/abort protocol between the participating network entities to undo the effects of network I/O calls on transaction abort.

Nakano et al. propose ReviveI/O system [9] for fault tolerant systems, which delays sending packets to the network until commit and also delays acknowledgment of packets coming to the host to transactionalize network calls. However, their approach reduces the network performance drastically and also violates correctness in certain cases. In this paper, we borrow some of the ideas of ReviveI/O but build a much more robust solution without sacrificing performance.

6.1 Solution Overview

Before going to our solution we briefly discuss the solution and problems of ReviveI/O.

ReviveI/O is primarily meant for supporting TCP calls inside transactions though it can be applied to other connection based protocols. When there is a `send()` call, in the normal scenario this would cause the data to be buffered by the local TCP subsystem which would eventually push the data out onto the network as network packets.

ReviveI/O solution. In ReviveI/O approach, they insert a layer (called PDD) between the TCP subsystem and the network device driver which holds the packet from being sent onto the network until checkpointing thus keeping isolation between the transaction and the outside world. The advantage of keeping this layer below the TCP stack is that all of the processing associated with send() is handled above and including the TCP subsystem and thus this method allows an eager execution of send() call without violating transaction semantics. In our system, we call this layer *xnet*,

To support TCP `recv()` calls within a transaction, ReviveI/O holds the outgoing acknowledgments at the PDD layer and sends them out only during commit (in their case after checkpointing). By doing so, if their is an abort, since the intended receiver would not receive the acknowledgments, it would simply retransmit the packet which the running transaction will consume when it retries.

ReviveI/O problems. As mentioned above, ReviveI/O delays acks until a corresponding recv() is called. Delaying acks (at the host end) suffers from two major performance problems.

Firstly, the transmission of acks by the TCP subsystem is asynchronous to calling recv() within an application and thus allows the TCP subsystem to buffer incoming packets. By delaying acks, the TCP subsystem would block the remote sender to send packets to the host TCP subsystem until transaction commit which is a serious performance bottleneck.

Secondly, delaying acks would cause the remote sender to assume that the packet sent was lost because of network congestion and would therefore reduce its sending rate resulting in reduced TCP bandwidth.

Xnet solution. In xnet, we use a solution similar to that proposed in ReviveI/O for send() calls within transactions, since it has very little performance overhead when transactions are short lived (which is the norm). However, for supporting recv() call in transactions, we use *buffering* of packets as opposed to delaying acks used in ReviveI/O. Though buffering adds an extra space requirement at the xnet layer, it gives a significant boost in performance over the ReviveI/O approach which is much more crucial for commercial servers.

6.2 Solution details

Though the ideas presented in this paper are applicable for UDP as well, for the purposes of this paper we consider only TCP connections. For simplicity of understanding we assume that at a given moment of time, only one thread/process accesses a given TCP connection.

Cross-layer support (libc and device driver). One obvious approach would be to delay the actual invocation of the system calls. This could be easily done by registering commit actions inside the libc wrappers. It would be impossible to compensate for the effects of these calls once the packet has either been sent to or received from the other machine in the network.

Although, it is possible to buffer the data for send() call within libc, this would greatly escalate the probability of the send() call aborting, since, a number of errors might occur during processing of data at the TCP layer. For e.g. the TCP layer might be willing to accept only X amount of data while the send() call has $X + \delta$ data. Thus, by buffering the data at the xnet device driver layer (i.e. below the TCP layer), the data would have gone through all the TCP processing and would mostly likely not cause an abort. Similarly, since acks are known only at the TCP layer and below, the xnet layer helps in buffering the acknowledgements.

In a distributed setting where all the participating systems are aware of transactions and have transactional memory support, the solutions could be totally different. For e.g. the system quicksilver [8] uses a distributed commit/abort protocol to undo the effects of a transactions on all participating network entities on an abort. We focus on a more generalized setting where the host machine should be able to handle transaction aborts by itself and should not require other network entities any knowledge of ongoing transactions.

Handling send(). To re-iterate, we propose processing the packet locally and delay sending it on the wire. Any send() request will be blocked by the PDD layer until signaled on commit time. Since this pseudo driver is oblivious of the transactions as such, we need some mechanism to signal it to send the packet on wire.

We do so by registering commit action inside libc wrapper. In the wrapper, the send() call is not delayed as is done for the system calls following actual delay semantics. Instead, in addition to the call invocation, a commit handler is registered which notifies the pseudo driver of the commit. Since the packet is allowed to reach the pseudo driver, it goes through

the local processing and thus any local errors are reported immediately to the application.

Handling recv(). For handling recv() and its variants, we require only user level support inside libc wrappers. This approach is different from what we have proposed for send() and its variants. Since, receiving packets by the TCP subsystem is an asynchronous process to receiving data by the application, we try to maintain the same behavior in a transaction scope also to minimize performance overhead due to transactions.

To handle recv() during aborts, we buffer the data used by recv() in libc. After an abort, when a transaction retries recv(), the data is fetched directly from libc's local buffer. Not only does this technique increase performance significantly over ReviveI/O approach (by allowing packet reception to occur asynchronously) but it also makes transaction retries faster, since the data is fetched from local buffers in libc. The local buffer is flushed on a commit by registering a commit action.

More details on the implementation issues follow later in this section.

Why recv in libc. In our prototype, we handle recv() call at user level in libc. We believe providing support at libc level for recv() is ideal because all the local processing on the packet would have been complete by the time packet reaches the libc. Thus any errors occurred during the local processing would be reported to the application immediately.

Why send in driver. send() call is dealt with at kernel level via pseudo device driver support. Again the packet would have been locally processed by the time it reaches the pseudo driver and thus any local errors will be caught before the packet leaves the local machine.

Moving functionality to tcp layer. As explained above, the reason for handling send() and recv() at different levels is to be able to finish the local processing immediately as soon as the system call is invoked in the application. But it's still an open question whether such an approach is pragmatic or not. For manageability point of view, it might be easier to to provide support at just one level. That could be made possible by moving support for both send() and recv() to the TCP subsystem.

6.3 Implementation Aspects

recv(). As mentioned before we handle recv() and its variants at user level in libc wrappers. In transactional case, we buffer the packet contents obtained by invoking a recv() call in a ring buffer. A separate ring buffer is maintained per connection and the complete set of ring buffers for all the connections is maintained in a hash table. Ring buffer gives us the abstraction of writing to and reading from a disk.

Two pointers are maintained namely head and tail. The head

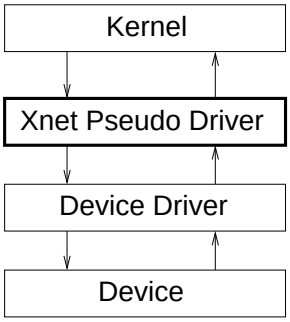

Figure 1: Block diagram showing that our xnet pseudo driver sits between the kernel and the device specific driver.

pointer moves when data is consumed from the ring buffer, if there are contents and the tail pointer moves when the data is stored to the ring buffer after invoking the recv() call. The idea is to consume the buffer whenever there is any content and invoke recv() only when the requirement can't be fulfilled from the buffer. But data is stored to the ring buffer only by the transactional thread. This is because, a non-transactional thread would never need to rollback (i.e. re-read the content).

To keep the head and tail pointers consistent, we need to maintain an additional commithead pointer which reflects the consistent head pointer. This is because, while reading from the buffer, the head pointers moves. So we need to maintain the previous location of head pointer which would be needed on an abort. In this case, we register both a commit and a compensating action which is often not the case with other system calls. The commit action essentially changes the commithead pointer in proportion to the data consumed from the buffer while the compensating action restores the head pointer to the previous consistent value (which would be stored in commithead pointer).

In case of a transactional thread, the steps are executed inside an escape action and isolation is provided by acquiring sentinels on the hashtable of ring buffers. We don't use sentinels for each ring buffer (i.e. per connection) since we assume that the connection will be accessed *sequentially*, a reasonable assumption for TCP connections.

send(). As mentioned earlier, we implemented a pseudo network driver for delaying the send till commit. This pseudo driver gives the abstraction of real network driver i.e. all the packets which are meant for the network driver are blocked the pseudo driver. The pseudo driver buffers them until signaled to send them at commit. The pseudo driver is a minimal driver which enables us to buffer the packets for required duration and send them to the real driver at appropriate time. The pseudo driver is signaled to flush the packets on commit by the libc wrapper for send().

Two main routines in the pseudo driver, which are of interest are send() and ioctl().

- send()

  The send() function in a typical network driver, receives the locally processed packets from the kernel and queues them to the device for transmission. Thus we customize the send() function in the pseudo driver to buffer the packets in a hashtable. Each packet is stored on a per connection basis which is identified by five parameters viz. source port, destination port, source ip address, destination ip address and the underlying protocol. Only packets which are identified as IP packets are buffered and rest of the packets (such as acks) are directly sent to the device specific network driver.

- ioctl()

  The ioctl() function in a typical network driver, implements any device-specific ioctl commands. We leverage this function to receive commands from libc to send packet out to the network on transaction commit and to discard the packet on transaction abort. Since, the TCP layer uses buffering, it might send a packet to xnet driver which might be smaller or larger than the number of bytes sent in the send() call by the transaction. So, when we receive a command in the ioctl() function, to send/discard data, we make corresponding checks to discard/send only the amount of data requested by the transaction.

Limitations. We don't handle retransmissions in our current implementation. Every packet is queued after the packets already present in the buffer. But retransmissions can be taken care of with some enhancements using TCP sequence numbers. This sequence could be used to order the transmission of packets and thus any retransmitted packets would be sent before the packets with sequence number higher than the retransmitted packet.

Another limitation of our approach is that, buffering support for both send() and recv() works, but may require buffering unbounded amount of data depending upon how large the transactions invoking such are calls, are. Due the absence of real workloads, the amount of data that needs to be buffered can't be predicted accurately. But the server workloads (the focus of our work) have small transactions that invoke such calls and thus our solution should perform well.

Testing. Since, our objective was to test our approach on the LogTM platform which does not support any networking with remote hosts at this point, we designed the following approach to test our solution.

We abstracted to the user the presence of a network interface. We then created two programs to communicate with each other via this interface. We had to do some tweaks in the networking subsystem, so that the packets are forwarded to our pseudo driver. In the normal case both the programs are running on the same machine, the packets would be given directly from the kernel to the requesting application and would never go to the driver.

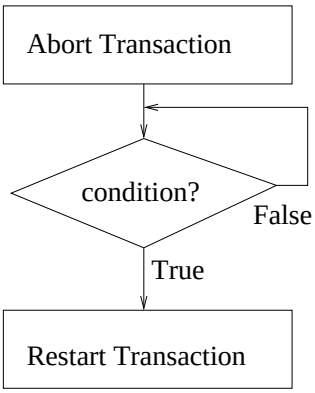

Figure 2: Figure shows our approach to solve deadlocks due to conditional synchronization in transactions. The reader transaction aborts to release isolation and sits in a tight loop until the condition variable is true and then restarts.

7. STRATEGY FOR CONDITIONAL SYNCHRONIZATION

To support conditional synchronization, we expose two new functions, tm_cond_wait() and tm_cond_signal(). While using conditional synchronization, the programmer invokes these two functions instead of the conventional condition variable based POSIX APIs.

The underlying idea in supporting conditional synchronization in transactions is to release isolation thereby allowing other transaction to modify the condition variable and busy wait on the conditional variable as opposed to blocking. A transactional thread that wants to wait for certain condition to become true, first has to release isolation and then busy wait until that condition becomes true.

Once the writer changes this variable (indicating the condition has become true), all the waiting threads will contend to move forward by accessing TM condition variable and reverting it back to its old value atomically by using atomic compare and swap. Since this is done atomically, only one thread will succeed and move forward and the rest will retry. Figure 2, shows the process as a flowchart.

Though we are aware of the disadvantages of busy waiting, this becomes the only available option for the TM systems which don't support blocking inside a transaction. Since that's the common case for the existing TM systems, we propose busy waiting instead of blocking which would be a potential option for systems having that support.

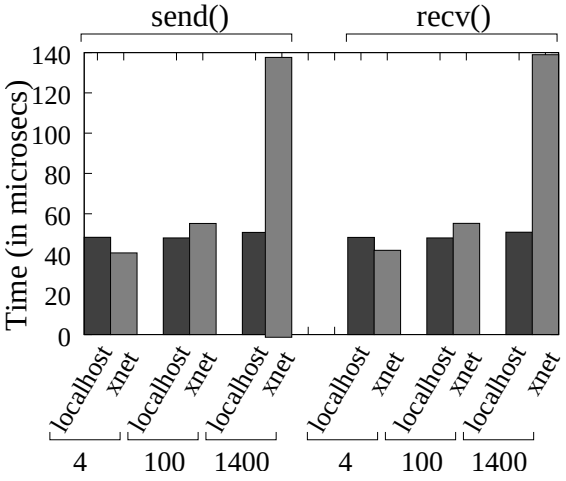

Figure 3: Figure shows time taken per recv()/send() call for different packet sizes 4, 100 and 1400 bytes when using our xnet driver and when using the loopback interface. The overhead for small packet sizes is low ¡10%, but it increases significantly for large packet sizes (1400 bytes)

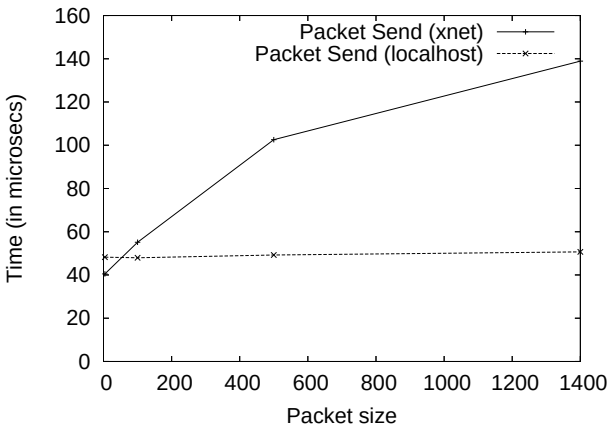

Figure 4: Figure shows the effect on time taken per send() call with varying packet size. Clearly, the time taken increases linearly with the packet size which is likely due to the malloc() calls in xnet

Processor	Sun T1 (Niagara)
Kernel	Solaris
Compiler	SunStudio
Xnet driver	780 lines of code
LibC	900 lines of code

8. EVALUATION

We evaluate only the performance overheads of network calls in this paper. For being able to get the performance results without requiring support from other networked machines, we used a simple technique. We created both our client and server programs on the same machine. When, the client sends a packet to the the server or vice versa, we forced the packet to go via our xnet driver. Normally, the packet would directly be forwarded to the server (and vice versa) by the kernel itself and thus we wouldn't be able to test our driver.

To force the packet in the xnet driver, we made both the client and server assume each other to be an imaginary ip, say 192.168.1.2 with an imaginary mac address, though none exists. Now, when the client sends the packet to server, it believes that the ip of server is 192.168.1.2. Since the kernel knows that this ip does not belong to any of the native interfaces, it forwards the packet to the driver. The driver, then swaps the ip addresses and mac addresses and forwards it to the server. The server thinks, that the ip address 192.168.1.2 is that of the client and creates a TCP connection accordingly. This allows us to rigorously test our driver. This is great for the LogTM simulator, since the simulator cannot connect with any remote hosts on the network.

We are mainly concerned about the performance overhead of network calls which is caused by the additional processing that needs to be done on every such call. For measuring the overhead, we compare our modified versions of system calls with unmodified ones. While comparing we pessimistically always assume transactions, while we expect them to be very few in real scenarios. We perform the evaluation on the system described in Table 2.

For testing the network calls, we used a simple client-server microbenchmark. The client sends multiple packets to the server and the server receives them.

8.1 Results

We compared the performance of the modified network calls against unmodified ones, w.r.t. to time taken to perform the call. Considering the bimodal distribution of TCP packets, we evaluated our calls for four packet sizes viz. 4 bytes and 100 bytes (which are small packets), 500 and 1400 bytes (which are large packets). Time taken per send() and recv() system all is reported in figure 3.

As shown in the figure 3, the average overhead per send() and recv() system calls is less than 10% for small packets (4 bytes and 100 bytes). However, for larger packets, the overhead increases significantly to 200% for 1400 byte packets. Figures 4 and 5 shows a linear trend in the send()/recv() overhead increase with the increase in the packet size. Since the maximum size of a TCP packet sent on the wire is 1500 bytes, maximum overhead with our system is 200% at the moment.

We suspect the reason for larger overhead in case of larger packets to be memory allocation to buffer packets done by the xnet driver. We could optimize this overhead by pre-allocating a large chunk of memory and storing packets in it.

We made an interesting observation regarding the size of

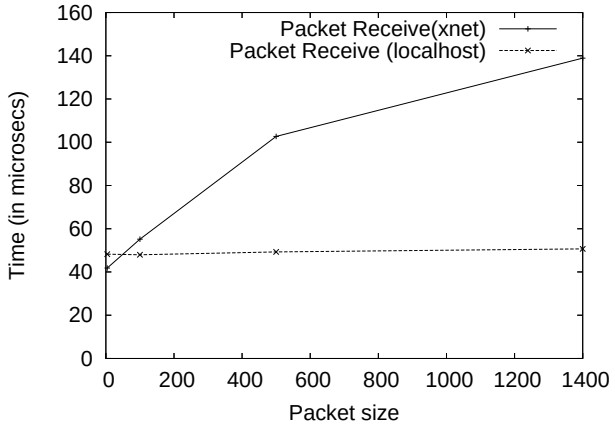

Figure 5: Figure shows the effect on time taken per recv() call with varying packet size. Clearly, the time taken increases linearly with the packet size which is likely due to the malloc() calls in xnet

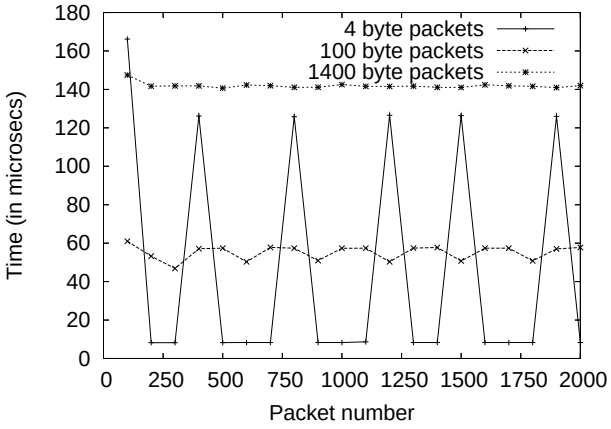

Figure 6: Figure shows the time taken for recv() call for different packet sizes over time. We can see some interesting buffering effects of TCP for small packet sizes (4 bytes here). For 4 byte packets, the time taken is higher for every 3rd packet, since TCP buffers this packet until it receives more data.

packet being sent. For very small packets, such as 4 byte packets, we can observe some spikes in the time taken for recv() call. While for larger packets, the time taken for send()/recv() is fairly uniform. We suspect the reason for spikes in case of small packets, to be packet concatenation done by the TCP protocol. Since recv() is a blocking call, the time taken is high when data is not readily available which happens when TCP starts combining packets instead of sending each packet individually. This is shown in figure 6.

Due to limitation in time, we could not perform an extensive evaluation. We could have have liked to test some of the following issues:

- Performance improvements over ReviveI/O.
- Performance improvements of recv() call on a transaction retry. We should perform significantly well, since the data for recv() should be fetched from local libc buffers.
- Testing rigorously on LogTM simulator
- more rigorous testing of all our modified system calls by running them with BIND and AOLserver.

9. LIMITATIONS

In this work, our primary focus is to address the issues raised by system calls when invoked inside transactions. We try to limit the restrictions posed by current TM systems, yet don't guarantee a complete solution to the problem. The problem is complex, and the complete class of system calls demands more time and resources. The solutions proposed certainly make it possible to use limited number of system calls inside transactions. We identify the following limitations with our approach -

Multiple system calls in a transaction. Current solution limits the number of system calls handled using delay till commit approach inside a transaction to 1. This is because of the way delay till commit approach works. Since it's assumed optimistically that the system call will succeed at commit time, a correct value is returned to the caller. If there are multiple such system calls in a transaction, and one of them fails at commit time, we can't abort the transaction because some of the system calls might have already made their effects visible in the commit handler.

send() followed by recv(). Currently, we cannot invoke send() followed by corresponding recv() in the same transaction. This is because send() is handled lazily and thus recv() will be blocked until the transaction commits (and thus actually send the packet). But the transaction can't commit before recv() gets through and this leads to a deadlock.

10. RELATED WORK

Our approach for system calls inside transactions, is inspired from LogTM's escape actios [5]. Morovan et al. proposed the generic idea of escaping transactions and executing system cade outside transactional scope. In addition they present compensating and commit actions which make aborts possible. We borrow the same ideas and apply them at library level by modifying system call wrappers in libc. In addition, we identify scenarios where mere user level support is not suitable such as network I/O.

Nakano et al. [9] borrow the idea of adding a PDD to the kernel to support disk I/O recovery which was proposed by Masubuchi et al [10]. They present a PDD based prototype implementation for I/O undo/redo in highly-available rollback-recovery servers. We borrow their idea of PDD for

supporting network I/O inside transactions. We apply this idea only for send() and its variants. While for recv() and its variants we propose a hybrid approach.

Haskin et al., in their system QuickSilver [8], use a distributed commit/abort protocol between the participating network entities to undo the effects of network I/O calls on transaction abort. Their solution works in a distributed setting where all the participating network entities have transactional memory support. We propose a generic solution without assuming transactional support in the remote entities.

McDonald et al. [11], propose commit and compensation actions (and their variation, violation actions) as the generic solution to all the system calls. But as we discussed earlier, such a model might not work for be appropriate in some scenarios. We on the other hand, analzye the issues with certain system calls more closely and propose kernel level support for network I/O.

McDonald et al. [11], propose doing conditional synchronization inside transactions using open-nesting and violation actions. Since our work is in context of LogTM with respect to compensation and commit actions, and LogTM has no direct support of violation actions, we propose a simpler solution for conditional synchronization based on condition variables. Though our solution changes the conditional synchronization semantics, they are still close enough to convetion. Also we don't rely on support for open nesting and violation actions.

Hudson et al. [12] and Damron et al. [13] looked at implementations of transaction safe malloc() and free() but we look at a broader class of system calls. Lowell et al. propose vistagrams [14] for fault tolerant systems, which handles state consistency using a co-ordinated effort between various network entities in a distributed system. However, our approach assumes no transactional support from any other outside entity in the network. Similarly, TIC proposed by Smaragdakis et al. [15], proposes a solution to maintain consistency in transactions using co-operative effort between various threads. However, we assume no such support in our system.

11. FUTURE WORK

In this work we provide support for a limited number of system calls and conditional synchronization in transactions. For making network I/O support more generic and robust, we would like to extend it with few more features such as handling retransmissions. In addition, We currently evaluate our implementation of system calls using microbenchmarks and by faking transactions.

For seeing the real impact, we would like to evaluate on real workloads such as BIND and AOLserver on a transactional memory system. Current implemenations of the HTMs are simulator based and thus too slow to analyse the system with rigourous workloads while STMs don't provide flexibilies such as escape actions.

12. CONCLUSION

Transaction Memory systems promise to be the right way to write concurrent programming code and propose to solve many of the programming issues lock based systems pose today. However, TM systems have a long way to go before they can be a practically viable solution. One of the roadblocks is that current TM systems cannot support system calls inside transactions.

In this regard, we looked at important server workloads - AOLserver and BIND for interesting use of system calls within locks which can be transactionalized . We observed two important classes of system calls, file system calls and network system calls which can be handled in transactions. In this project we focus on the network calls send()/recv().

We propose using a combination of user level and kernel level support for handling network calls leveraging support of device drivers. Though our performance overhead of this approach is high, by doing certain optimizations in the memory allocation aspect, we can minimize these overheads. Finally, we proposed a solution for handling conditional synchronization which might potentially cause deadlocks.

There are still multiple problems to be solved in making system calls transaction safe and we believe with a good effort from the TM community, the results should soon be visible.

13. REFERENCES

[1] L. Hammond et al., "Transactional memory coherence and consistency," in *ISCA*, 2004.

[2] T. Harris, "Exceptions and side-effects in atomic blocks," in *Sci. Comput. Program*, 2005.

[3] C. Zilles and L. Baugh, "Extending hardware transactional memory to support non-busy waiting and non-transactional actions," in *TRANSACT*, 2006.

[4] E. C. Lewis C. Blundell and M. M. K. Martin, "Unrestricted transactional memory: Supporting i/o and system calls within transactions," in *Tech Report, Department of Computer and Information Science, University of Pennsylvania*, 2006.

[5] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood, "Supporting nested transactional memory in logtm," in *ASPLOS*, 2006.

[6] Michael W. Shapiro Bryan M. Cantrill and Adam H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX*, 2004.

[7] Moore et. al, "Logtm: Log-based transactional memory," in *HPCA*, 2006.

[8] Yoni Malachi Rober Haskin and Gregory Chan, “Recovery management in quicksilver,” in *ACM Transactions on Computer Systems*, 1988.

[9] Kourosh Gharachorloo Jun Nakano, Pablo Montesinos and Josep Torrellas, “Revivei/o: Efficient handling of i/o in highly-available rollback-recovery servers,” in *HPCA*, 2006.

[10] Y. Masubuchi et al., “Fault recovery mechanism for multiprocessor servers,” in *International Symposium on Fault-Tolerant Computing*, 1997.

[11] Austen McDonald et al., “Architectural semantics for practical transactional memory,” in *ISCA*, 2006.

[12] Ali-Reza Adl-Tabatabai Richard L. Hudson, Bratin Saha and Benjamin C. Hertzberg, “Mcrt-malloc: a scalable transactional memory allocator,” in *ISMM*, 2006.

[13] Damron et al., “Hybrid transactional memory,” 2005.

[14] Persistent Messages in Local Transactions, “David e. lowell and peter m. chen,” in *PODC*, 1998.

[15] Reimer Behrends Yannis Smaragdakis, Anthony Kay and Michal Young, “Transactions with isolation and cooperation,” in *OOPSLA*, 2007.