Supporting File Operations in Transactional Memory

Brian Demsky and Navid Farri Tehrany

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

bdemsky@uci.edu

CECS Technical Report 10-09
August 24, 2010

Supporting File Operations in Transactional Memory

Brian Demsky and Navid Farri Tehrany

Abstract

Researchers have proposed transactional memory as a concurrency primitive to simplify the development of multithreaded programs. In this paper we present a new approach for supporting I/O operations in the context of transactional memory. Our approach provides isolation between the file operations of different transactions while allowing multiple transactions to concurrently perform I/O. To ease adoption, our approach attempts to implement the traditional I/O programming interface as closely as possible. We formalize aspects of our approach and use the formalization to reason about the correctness of the approach.

We have implemented our approach as a Java library and have integrated it with the DSTM2 transactional memory system. We have evaluated the approach with several benchmarks including JCarder, TupleSoup, a financial transaction benchmark, parallel sort benchmark, and a parallel grep benchmark. Our experience shows that the approach provides a straightforward mechanism for developers to integrate I/O in a transactional memory environment and that it performs well.

Keywords: Transactional Memory, I/O

1. Introduction

As multi-core processors become more prevalent, parallel software development needs to become mainstream. Traditional concurrency primitives like locks require software developers to reason about often complex interactions between threads. In an effort to simplify parallel programming, researchers have developed transactional memories [1, 2, 3, 4, 5, 6, 7]. With transactional memory, the developer simply declares which parts of the application code should be executed with transactional semantics and the transactional memory implementation handles the details of providing transactional semantics. While software transactional memories do not eliminate all of the difficulties of developing parallel applications, they do simplify many aspects including eliminating the possibility of deadlocks. Moreover, through optimistic strategies, transactional memories can often provide more parallelization than coarse grain locking. Recently, researchers have explored a number of approaches to allow transactions to perform I/O operations.

Researchers have developed an approach to transactional I/O called xCalls that uses a combination of deferral and compensation to allow transactions to make system calls [8]. Transactions in xCalls lock the file system at the file granularity. Writes incur the extra overhead of reading and buffering old data to support rollback if a transaction aborts. One major weakness of xCalls is that it serializes transactions that access the same file. The relatively common practice of appending to a log file would serialize transactions. Because xCalls detects conflicts at the file granularity, transactions that randomly access disjoint portions of a file would be serialized. Our approach can be viewed as complementary to xCalls, we focus on maximizing potential concurrency for file system operations and assume that an approach like xCalls would be used for socket communication and other system calls. Our approach allows multiple transactions to simultaneously update the same file — this is critical for applications that log data to a file or applications like databases that update disjoint parts of large files.

In this article, we present a transactional-I/O implementation that enables multiple transactions to simultaneously perform file I/O operations. Our system has been designed to be easily integrated with existing software transactional memory implementations. Our approach enhances the capabilities of transactions in general since the software developer can now mix both I/O operations and transactions.

1.1. The Basic Approach

Our basic approach defers making changes to the actual file system until a transaction commits. This enables our system to avoid rollbacks to the underlying file system. Moreover, this delay enables several optimizations for the common file append operation. Our implementation defers computing the exact location of writes that simply append to a file until the transaction commits, thus eliminating a potential source of transaction conflicts.

Our system maintains *visible reader* structures that the transaction commit process uses to discover all live transactions that have read from a block or file descriptor offset. Our implementation uses these structures to detect and abort any conflicting transactions when a transaction commits.

Each transaction maintains a write store for each inode (i.e. file) that records all writes that the transaction has made to that inode. Transactional reads first check the transaction's write buffer to see if the current transaction has written to the location and then access the underlying file system to read the committed changes.

We implement optimizations for two common I/O patterns: append-only accesses and random accesses. Our approach tracks a transaction's dependence on the file offset for each file descriptor. The append-only optimization enables multiple transactions to simultaneously log to the same file descriptor by deferring the selection of the file offset until the transaction commits. The random access optimization enables multiple transactions to simultaneously read and write to disjoint parts of a file using the same file descriptor. This optimization determines that the transaction first seeks and therefore does not depend on the committed value of the file descriptor's file offset.

Our implementation coordinates transaction commits with the transactional memory system to guarantee transactional semantics for transactions that combine file and memory operations. To commit a transaction, our implementation locks the file resources that the transaction accessed to ensure the transaction cannot be aborted, then calls the transactional memory system to commit the memory changes, and finally commits the file changes.

1.2. Contributions

This article makes the following contributions:

- **Transactional I/O System:** The article presents a transactional I/O approach that enables I/O to be efficiently integrated into transactions. Our approach bridges the gap between language-level transactional constructs and the underlying operating system primitives that do not support transactions. Compared to previous work, our system allows more transactions to simultaneously perform I/O creating the potential for greater parallelization benefits. Our system implements strong atomicity [9] — the developer can perform I/O operations outside of transactions and each individual I/O operation is treated as a transaction that performs a single I/O operation. Note that this guarantee is made within the context of one application — no guarantees are made regarding concurrent reads and writes to the same files from external applications. Such guarantees would require operating system support.

- **Formalizing Transactional I/O Operations:** This article formalizes aspects of our system. It then uses the formalization to derive a set of conditions that ensure the correctness of the implementation. This formalization makes it easy to reason about the correctness of the optimizations that our system implements.
- **Adaptability to Various Memory System:** Our implementation has minimal dependencies on the specific software transactional memory implementation and should be easy to port to other implementations. This will enable researchers who have taken different approaches to implementing transactional memories to easily support I/O.
- **Backwards Compatible Interface:** Our library was designed to provide an API that is similar to the standard Java API. This provides developers with the flexibility to easily port existing code to the transactional model rather than rewriting the code from scratch.

The remainder of the article is structured as follows. Section 2 presents the programming model for our approach to transactional I/O. Section 3 presents usage scenarios and discusses the implications on how transactions depend on file offsets. Section 4 presents formal properties of our approach and reasons about a set of checks that ensure correctness of executions. Section 5 presents our implementation. Section 6 presents our evaluation of our implementation. Section 7 discusses related work; we conclude in Section 8.

2. Programming Model

We assume the presence of an underlying transactional memory system. In particular, we assume that the software transactional memory system conceptually provides the developer with two operations *begin transaction* and *end transaction* that denote the beginning and end, respectively, of a sequence of operations in a thread of execution that should be executed with transactional semantics. In this context, transactional semantics mean that the behavior of the set of operations that the transactional region of code performs is consistent with some serialization of the transactions. We assume that the underlying transactional memory system implements transactions for a set of operations that include memory reads and writes.

Our transactional I/O approach extends the base transactional memory semantics with transactional support for file operations. We preserve the standard Java file API when possible. We present the core transactional file API below. We omit a number of helper functions that are straightforward to implement using the core API and a number of other file classes whose functionality is a subset of the `RandomAccessFile` class.

- `new TransactionalRandomAccessFile(path)`: This constructor creates a new transactional file object for the file whose location is given by the string `path`.
- `read(byte[] data)`: This method reads from the current offset of the file object, stores the bytes in the byte array `data`, increments the file offset by the number of bytes read, and returns the number of bytes read.
- `write(byte[] data)`: This method writes the bytes stored in the array `data` to the current offset of the file object and increments the file offset by the number of bytes written. Writes in our system always write all of the bytes. For backwards compatibility with the standard file API, our implementation of `write` returns the number of bytes that the application requested to write.
- `getFilePointer()`: This method retrieves the current offset of the file object.
- `seek(int offset)`: This method assigns the current offset of the file object to the value from the parameter `offset`.

The intention is that developers replace uses of the standard `RandomAccessFile` class with the `TransactionalRandomAccessFile` class. The class can be used both inside and outside of transactions. Our transactional I/O approach implements strong atomicity — if a file operation is invoked outside of a transaction, the invocation has the behavior of a transaction with a single I/O operation.

3. File Offset Dependencies

In this section we discuss several common I/O usage patterns and how the patterns introduce dependencies on a file descriptor's file offset. We describe these dependencies and present a finite state machine that models them.

3.1. Linear Access

We first discuss the linear file access pattern. In this pattern, multiple transactions access a single shared file descriptor. These transactions read and/or write from the file descriptor without performing any intermediate seeks. Since all transactions access the file through same descriptor, they share the same file offset. This shared file offset can introduce dependencies. If a transaction reads from a file descriptor and a second transaction that accessed the same file descriptor commits, the first transaction must abort as it has read data from the wrong location in the file.

We note that if a transaction only writes to a file descriptor and a second transaction that accessed the same file descriptor commits, it is still possible to commit the first transaction. This is possible because the first transaction has no dependence on the current file offset — this strategy delays computing the location of the write until the transaction commits and therefore eliminates the transaction's dependence on the current file location. This is a common usage pattern that appears in applications that log data to files.

3.2. Random Access

We next discuss the random access pattern. In this pattern, an application executes transactions on a single shared file descriptor. These transactions first seek to set the file offset and then execute reads and/or writes to or from the file descriptor. Since the transactions first set the current file offset, they do not depend on the current committed file offset.

3.3. Dependencies

We next describe how our approach tracks a transaction's dependence on the file offset associated with a file descriptor. We describe a transaction's dependence on the offset of a file descriptor using the following 4 states:

- `No Access`: This is the initial state for all file descriptors in a transaction and is changed as soon as the transaction performs a file operation on the file descriptor.

- `No Dependence`: This state indicates that the transaction does not depend on the initial value of the file descriptor's file offset.

- `Write Dependence`: This state indicates that although the transaction depends on the initial value of the file descriptor's file offset, this dependence can be resolved during the commit process.

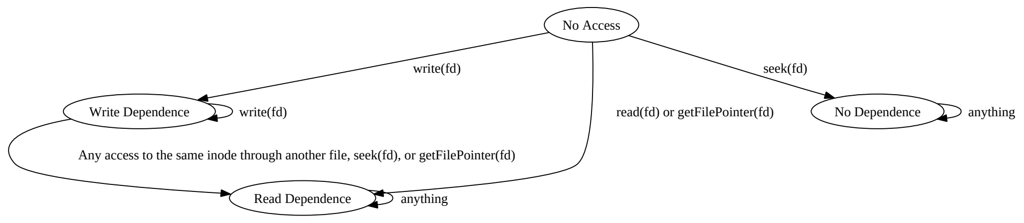
No Access
write(fd)
seek(fd)
Write Dependence
write(fd)
read(fd) or getFilePointer(fd)
No Dependence
anything
Any access to the same inode through another file, seek(fd), or getFilePointer(fd)
Read Dependence
anything

Figure 1: State Machine for Offset Dependence States

- `Read Dependence`: This state indicates that the transaction depends on the file descriptor's offset and that the transaction must be aborted if the offset is changed by any subsequent transaction commits.

Figure 1 presents a finite state machine that describes how file descriptors accessed by a transaction transition through the dependence states. At the beginning of a transaction, all file descriptors start in the `No Access` state.

If the first access to a file descriptor is a seek, then the descriptor transitions into the `No Dependence` state. This indicates that after the seek operation sets the file offset, all subsequent operations in the transaction on the same descriptor use the offset value from the seek. Therefore, the subsequent operations do not depend on the initial file offset and the transaction does not depend on the initial file offset for that descriptor.

If the first access to a file descriptor is a read, then the file descriptor transitions into the `Read Dependence` state. This indicates that the read operation depends on the initial value of file descriptor's file offset. Therefore, the transaction depends on the file descriptor's file offset.

If the first access to a file descriptor is a write, then the file descriptor transitions into the `Write Dependence` state. This indicates that although the transaction conceptually depends on the initial offset, this dependence can be resolved during the commit process by delaying computing the exact offset of the write.

If a transaction reads from the same underlying file as a file descriptor in the `Write Dependence` state, then both file descriptors transition into the `Read Dependence` state. This indicates that the location of the previous writes must be computed to determine if the previous writes overlap with the read operation.

If a transaction performs an operation that accesses the file pointer of a file descriptor or seeks in the `No Access` or `Write Dependence` states, then the file descriptor transitions into the `Read Dependence` state. This indicates that the transaction now depends on the initial value of the file descriptor's file offset.

3.4. Translating I/O Operations into Primitives

We next describe how the API level file operations are translated into low-level primitives. This process makes the dependencies on file offsets explicit. This dependence is represented with a special `readoffset` primitive. The variable `fd` denotes the value for the file descriptor.

- `startTransaction()`: This operation starts a new transaction and sets the state of all file descriptors to `No Access`.
- `seek(fd)`: This operation is translated into a seek primitive. It also transitions the state of the file descriptor as shown in Figure 1.
- `write(fd, data)`: This operation is translated into a write primitive. It also transitions the state of file descriptor as show in Figure 1.
- `getFilePointer(fd)`: This operation is translated into a `readoffset` primitive if the fd is in the `No Access` or `Write Dependence` states.
- `read(fd)`: Read operations require the location of all writes to the same file in the same transaction to be resolved. Formally, $\forall \texttt{fd}' \neq \texttt{fd}$ such that $\texttt{inode}(\texttt{fd}') = \texttt{inode}(\texttt{fd})$ and $\texttt{fd}'$ is in the `Write Dependence` state, the operation inserts a `readoffset` primitive for $\texttt{fd}'$.

  A read operation is translated into a combination of a `readoffset` primitive and a `read` primitive if the `fd` is in the `No Access` or `Write Dependence` states. A read operation is translated into a `read` primitive if the `fd` is in the `No Dependence` or `Read Dependence` states. This operation transitions the state of the file descriptor as shown in Figure 1.
- `endTransaction()`: This operation initiates the transaction commit.

4. Formal Model

In this section we formalize aspects of our approach. The section begins by presenting the semantics for each I/O operation. We then derive a set of runtime checks that guarantee that I/O operations in a transaction conform to these semantics.

4.1. Formal Programming Model

Each program execution contains a set of transactions $\{T_1, ..., T_n\}$. Each transaction T consists of a sequence of operations $\Phi_T = \o_{T1}; ...; \o_{Tn_T}$. A program execution $\Phi_{execution}$ consists of a totally ordered sequence of operations $\o_{T_{i_1}j_1}; ...; \o_{T_{i_n}j_n}$. We supposed that individual operations are atomic, we discuss this issue in more detail in Section 4.4. We use the standard correctness definition for transactions which we formalize below in Definition 4.1.

Definition 4.1 (Execution Correctness)**.** A program execution $\Phi_{execution} = \o_{T_{i_1}j_1}; ...; \o_{T_{i_n}j_n}$ is correct if there exists a totally order sequence $T_{k_1}; ...; T_{k_m}$ that includes all of the transactions in the program such that the corresponding sequence of operations $\Phi_{execution}$ has the same observable behavior as the sequence $\o_{T_{k_1}1}; ...; \o_{T_{k_1}n_{T_{k_1}}}; ...; \o_{T_{k_m}1}; ...; \o_{T_{k_m}n_{T_{k_m}}}$.

4.2. Primitive Operations

In this section, we discuss the set of primitive operations. Our system contains the following primitive operations:

4.2.1. Write Primitive

The `write(fd, buffer, length)` primitive writes `length` bytes from the array `buffer` to the file descriptor `fd`. Our formalization assumes that the write statement writes to the transaction's local buffer and has no externally visible effects until the transaction commits. The effects of a write are visible immediately to reads in the same transaction and only after this transaction commits to reads from other threads.

4.2.2. Read Primitive

The `read(fd, buffer, length)` primitive returns the data read from the file descriptor `fd`. The read begins starting at the transaction local file offset for the given file descriptor, reads the specified number of bytes, and then updates the local file offset. The read operation observes data written in the same transaction to the same file (through this or other file descriptors) and data written by committed transactions. Our formalization assumes that the read operation first checks to see if the current transaction's local buffer contains the data and if not reads the committed data from the file system.

4.2.3. Seek Primitive

The `seek` primitive updates the transaction's local offset for the file descriptor to the specified offset. Our formalization assumes that the seek primitive has no externally visible effects until the transaction commits. This offset can be observed by subsequent transaction local file operations to the same descriptor, and is used to update the committed offset for the file descriptor when the transaction commits.

4.2.4. Readoffset Primitive

The primitive operation `readoffset(fd)` reads the committed file offset for the file descriptor and returns this value. This introduces a dependence of the transaction on the currently committed offset of the file descriptor. If the transaction has already performed write operations that append to the file descriptor, this operation has the effect of forcing the transaction to choose an absolute offset for those write operations.

4.2.5. Commit Primitive

A transaction commits when it invokes the `commit` primitive. After the transaction commits, the values of any writes it made become visible to other threads and it updates the committed file offsets with the transaction local offsets. Our formalization does not make transaction aborts explicit, aborted transactions simply do not include a `commit` primitive.

4.3. Commutativity of Operations

We next derive checks that guarantee our implementation's correctness. We begin with several lemmas on the commutativity of primitive operations.

Lemma 4.1 (Commutivity of File Primitives)**.** Let $\Phi_{executed} = \emptyset_1; ...; \emptyset_i; \emptyset_{i+1}; ...; \emptyset_n$, $\Phi_{derived} = \emptyset_1; ...; \emptyset_{i+1}; \emptyset_i; ...; \emptyset_n$, $\emptyset_i \in \Phi_{T_k}$, and $\emptyset_{i+1} \in \Phi_{T_l}$. If $k \neq l$ and both $\emptyset_i$ and $\emptyset_{i+1}$ are taken from the set of operations $\{$`read`, `write`, `readoffset`, `seek`$\}$ then $\Phi_{executed}$ and $\Phi_{derived}$ have the same observable behavior.

Proof Sketch: This lemma follows from the observation that operations other than commit only update state that is local to the given transaction and access state that is either local to the transaction or global. Therefore, primitive operations other than commit commute with primitive operations other than commute from other transactions.

Lemma 4.2 (Commutivity of `write` and `commit` primitives)**.** Let $\Phi_{executed} = \emptyset_1; ...; \emptyset_i; \emptyset_{i+1}; ...; \emptyset_n$, $\Phi_{derived} = \emptyset_1; ...; \emptyset_{i+1}; \emptyset_i; ...; \emptyset_n$, $\emptyset_i \in \Phi_{T_k}$, and $\emptyset_{i+1} \in \Phi_{T_l}$. If $k \neq l$ and both $\emptyset_i$ and $\emptyset_{i+1}$ are `write` and `commit`, respectively, then $\Phi_{executed}$ and $\Phi_{derived}$ have the same observable behavior.

Proof Sketch: This follows from the observation that because `write` does not return a value (that depends on its state), its behavior is not changed by a `commit` primitive from a different transaction. Since the action of the `write` operation affects only transaction local state until the corresponding `commit`, it cannot affect the `commit` operation in the other thread.

Lemma 4.3 (Commutivity of `seek` and `commit` primitives)**.** Let $\Phi_{executed} = \o_1; ...; \o_i; \o_{i+1}; ...; \o_n$, $\Phi_{derived} = \o_1; ...; \o_{i+1}; \o_i; ...; \o_n$, $\o_i \in \Phi_{T_k}$, and $\o_{i+1} \in \Phi_{T_l}$. If $k \neq l$ and both $\o_i$ and $\o_{i+1}$ are `seek` and `commit`, respectively, then $\Phi_{executed}$ and $\Phi_{derived}$ have the same observable behavior.

Proof Sketch: This follows from the observation that because `seek` does not return a value and only updates the transaction local offset (until the corresponding `commit`), its behavior is not changed by a `commit` operation in a different transaction.

Lemma 4.4 (Commutivity of `read` and `commit` primitives for non–conflicting transactions)**.** Let $\Phi_{executed} = \o_1; ...; \o_i; \o_{i+1}; ...; \o_n$, $\Phi_{derived} = \o_1; ...; \o_{i+1}; \o_i; ...; \o_n$, $\o_i \in \Phi_{T_k}$, and $\o_{i+1} \in \Phi_{T_l}$. If

1. $k \neq l$,
2. $\o_i$ and $\o_{i+1}$ are `read` and `commit` operations, respectively, and
3. the blocks read from the committed copy of the file by the `read` operation are disjoint from blocks written by the committed transaction,

then $\Phi_{executed}$ and $\Phi_{derived}$ have the same observable behavior.

Proof Sketch: The only shared state accessed by a `read` operation are the blocks it reads from the committed copy of the file. If the committed transaction does not update any of the same blocks, the read operation must have the same behavior in both $\Phi_{executed}$ and $\Phi_{derived}$.

Lemma 4.5 (Commutivity of `readoffset` and `commit` primitives)**.** Let $\Phi_{executed} = \o_1; ...; \o_i; \o_{i+1}; ...; \o_n$, $\Phi_{derived} = \o_1; ...; \o_{i+1}; \o_i; ...; \o_n$, $\o_i \in \Phi_{T_k}$, and $\o_{i+1} \in \Phi_{T_l}$. If

1. $k \neq l$,
2. $\o_i$ and $\o_{i+1}$ are `readoffset` and `commit` operations, respectively, and
3. the committed transaction does not contain any `read(fd)`, `write(fd)`, or `seek(fd)` primitive operations for the same file descriptor `fd` accessed by the `readoffset(fd)` primitive,

then $\Phi_{executed}$ and $\Phi_{derived}$ have the same observable behavior.

Proof Sketch: By inspection of the behavior of the primitive operations, a transaction that does not contain any `read(fd)`, `write(fd)`, or `seek(fd)` primitive operations for the file descriptor `fd` does not modify the committed file offset for `fd`. The `readoffset(fd)` primitive only accesses the committed offset for the requested file descriptor `fd` and does not update any committed state. Therefore, a `commit` operation for a transaction that does not contain any `read(fd)`, `write(fd)`, or `seek(fd)` primitive operations for the same file descriptor `fd` commutes with the `readoffset(fd)` primitive.

4.3.1. Commit Checks

We next derive from the previous lemmas a set of conditions under which if a transaction always aborts guarantees the correctness of the execution. A transaction abort prevents a transaction from updating the committed program state or from being visible to other transactions. Note that aborted transactions do not successfully complete a `commit` primitive operation.

Lemma 4.6 (Transaction Aborts)**.** Let $\Phi_{executed}$ be an execution in which transaction T_i aborts. Let Φ' be the sequence $\Phi_{executed}$ with the operations in Φ_{T_i} removed. Then $\Phi_{executed}$ and Φ' have the same observable behavior.

Proof Sketch: As T_i has been aborted, it does not include a `commit` operation. Therefore, the operations in Φ_{T_i} do not change state that can be accessed by any of the operations in transactions other than T_i and therefore can be removed without changing the observable behavior.

Theorem 4.7 (Correctness of Committed Transactions)**.** If the `commit` primitive aborts all uncommitted transactions T' in which

1. the execution sequence $\Phi_{T'}$ contains a `read(fd)` operation that reads the a block from the committed copy of a file written to by a transaction that commits after the `read(fd)` operation but before T' finishes
2. the execution sequence $\Phi_{T'}$ contains a `readoffset(fd)` operation on a file descriptor `fd` that a transaction which commits after the `readoffset(fd)` but before T' finishes performed a `read(fd)`, `write(fd)`, or `seek(fd)` on

then any execution sequence $\Phi_{executed}$ is correct.

Proof Sketch: By Lemma 4.6, we can construct Φ' that has same behavior as $\Phi_{executed}$ but contains only operations from committed transactions. Let $T_{h_1}; ...; T_{h_m}$ be the order that the transactions in $\Phi_{executed}$ committed.

Let $\Phi_{sequential} = \emptyset_{T_{h_1}1}; ...; \emptyset_{T_{h_1}n_{T+h_1}}; ...; \emptyset_{T_{h_m}1}; ...; \emptyset_{T_{h_m}n_{T_{h_m}}}$. By Lemma 4.1, file primitives from different transactions commute. By Lemmas 4.2 and 4.3, `write` and `seek` primitives commute with `commit` primitives from different transactions. By Lemma 4.4 and condition 1, `read` primitives commute with `commit` primitives from different transactions. By Lemma 4.5 and condition 2, `readoffset` primitives commute with `commit` primitives from different transactions. In summary, all pairs of primitive operations (except a pair of `commit` operations) from different transactions commute. As the `commit` operations appear in the same order in both Φ' and $\Phi_{sequential}$ and the operations in a given transaction appear in the same relative order, the commutativity properties above allow the operations in Φ' to commuted to the order in $\Phi_{sequential}$ without changing the execution's behavior.

4.4. Atomicity of Operations

The formalization assumes that individual operations are atomic. An implementation must take care to ensure that `read`, `readoffset`, and `commit` primitive operations are atomic with respect to each other. Our implementation uses locks to ensure this property. Individual primitives other than `commit` are trivially atomic with respect to each other because they operate on disjoint state (they only modify transaction local state).

5. Implementation

In this section we describe the approach taken by our transactional I/O implementation. Our implementation uses a *visible-reader* based approach. For each file block, our implementation maintains a list of the transactions that have read that block. When a transaction commits changes to a set of file blocks, it must first abort all transactions that have read from those blocks. Similarly, our implementation maintains a list of transactions that have read the file offset from a file descriptor. When a transaction commits changes to the offsets of a set of file descriptors, it must first abort all transactions that have read those file offsets.

5.1. State

We next describe the state maintained by our implementation. For each block of each inode, our implementation maintains a visible reader structure. Our implementation also maintains a visible reader structure for the file descriptor. Visible reader structures contain a lock and a list of readers.

Our implementation maintains a *transaction store* for each transaction. The transaction store tracks the state for each file descriptor the transaction accesses as described in Section 3.3. The transaction store maintains a buffer for each inode that contains a list of the writes that the transaction has performed on that inode. Each write in the list contains the array of bytes written and an offset. The offset can either be an absolute location in a file or a relative offset to the current committed offset.

5.2. Operations

We next describe the implementation of each operation:

- `write` **operation:** The `write` operation writes to the transaction write buffer of the underlying inode. Our implementation has two cases — one for writes to an absolute location and one for writes to a relative location. The implementation performs an absolute write if the file descriptor is in the `Read Dependence` or `No Dependence` states. If the file descriptor is in any other state, the implementation performs a relative write. The write operation copies the data to the write buffer. The write buffer is organized as a tree. If a write overlaps with data previously written by the same transaction, the implementation combines the writes.

- `readoffset` **operation:** The `readoffset` operation first locks the file descriptor. It next adds the transaction to the reader list for the file descriptor, reads the current offset value, and unlocks the file descriptor. It then scans the transaction's buffer of uncommitted writes for this file descriptor and sets their offsets using the current committed offsets. Finally, it stores the transaction's current offset for the file.

- `read` **operation:** The `read` operation obtains the current file descriptor offset from the transaction's current offset for the file. For each byte, it then checks to see if the data is available in the transaction's write buffer. Note that the translation step that generates `readoffset` operations to ensure that all writes in the write buffer are absolute before a `read` primitive. If a byte is not available in the transaction's write buffer, the `read` operation reads it from the file. When the `read` operation accesses a new block in the file, it locks that block's visible reader structure, adds the current transaction as a reader, and finally unlocks the structure and issues a read call to the operating system.

- `commit` **operation:** The `commit` operation begins by locking the visible reader structures for all the file descriptors that the transaction read, wrote, or seeked. It next computes the absolute offsets for all writes it performed and locks the visible reader structures for all the file blocks that the transaction accessed. At this point the transaction cannot be aborted by the I/O of another transaction. The `commit` operation for the transactional memory system is then called. If the transactional memory system aborts, our library releases all of the locks and aborts the transaction. Otherwise, our library commits the I/O operations by writing the contents of all the write buffers to the corresponding files. The transaction then removes itself from all visible reader lists and aborts all other transactions that appear in a visible reader list for any block the transaction writes to or any file descriptor it modifies. Finally, the transaction releases all of the locks.

5.3. Correctness

The correctness of our implementation follows from Theorem 4.7. Note from Section 5.2 that when a transaction commit writes to a block, it first acquires a lock on the visible reader structure for that block. Any uncommitted transaction that has already read from that block would appear in the list of readers, and therefore would be aborted by the transaction commit. Therefore, the implementation satisfies condition 1 of the theorem.

Similarly, when a transaction that performed a `read`, `write`, or `seek` to a file descriptor commits, it locks the visible reader structure for that descriptor. As all uncommitted transactions that performed a `readoffset` to that descriptor will appear in its visible reader list, they will be aborted when the first transaction commits. The implementation satisfies condition 2 of the theorem and, therefore, the implementation guarantees serializability.

5.4. Consistency of Aborted Transactions

A second concern is whether transactions that will eventually abort can observe values that are inconsistent. Our approach ensures that all transactions observe consistent values at all times. The combination of visible readers with commit operations that abort all transactions that have read or updated data ensure that transactions are aborted before they can read values that are inconsistent.

5.5. Discussion

The designs of transactional memory are still evolving. One of the primary goals of this project is to develop a transactional I/O library that can easily be adapted for use by a wide range of transactional memory implementations. An important constraint on our design is that both the memory component and the file accesses of a transaction must serialize at the same time. Some transactional memory implementation approach commit a transaction with a single instruction — we have to ensure that the file operations serialize at the same instance.

A simple approach to supporting the wide range of possible transactional memory implementation strategies is to split the I/O commit operation into two components: the first acquires locks to ensure that the transaction's I/O component can safely commit at any point in the future and the second then commits the I/O component. The memory transaction is committed between the execution of these two components.

This strategy requires acquiring a read lock on all file data that a transaction reads. This largely negates the benefits of invisible readers, and therefore we chose to make readers visible so that contention management code can make better decisions. One additional benefit of this decision is that it makes it trivial to guarantee that transactions never read inconsistent values from a file (transactions that read inconsistent values are often called zombie transactions in the literature).

We do note that using our implementation with transactional memories like TL2 [10] that rely on consistent snapshots to avoid reading inconsistent values would require an additional check to verify its read set after each file read is performed as a file read could access data that is inconsistent with the (stale) memory snapshot. Our approach can be extended to assign a version number to the file system, the check can then be elided if the version number indicates that no transaction has committed a change to the file system since the transaction's last validation of its read set. We expect that file operations are performed less frequently than memory operations and therefore visible readers for files are likely to scale to a much larger number of cores than they do for memory operations.

Our implementation has focused on the correct behavior for normal file accesses. It is possible that a delayed write operation could fail because of insufficient permissions or lack of disk space. Our current implementation does not support error handling. However, it is straightforward to check write permissions and to reserve sufficient space on the disk when the write operation is buffered.

6. Evaluation

We next discuss our evaluation of our implementation on several microbenchmarks and five benchmarks: TupleSoup, a database; JCarder, a deadlock detector; a financial benchmark; a parallel sort; and a parallel grep.

6.1. Methodology

We implemented a fully transactional file library for Java. It is available for download at `http://demsky.eecs.uci.edu/software.php`. We have extended DSTM2 to support our transactional file library. As a reference for our system we have developed an inevitable implementation, referred to as inevit in graphs, that uses inevitable transactions [11]. In this approach the system allows a single transaction to be designated as inevitable. Inevitable transactions are guaranteed to commit — if a conflict is detected between an inevitable transaction and a second transaction, the contention manager resolves it by aborting the second transaction. This approach allows a single transaction to invoke I/O operations and any other transaction that attempts I/O must wait for the first transaction to commit.

We executed the benchmarks on a machine with two quad-core 2.2 GHz Xeon E5520 (Nehalem) processors. The machine runs the 64-bit version of the Debian Linux distribution running kernel version 2.6.32rc8 and the Sun Java HotSpot 64 bit Server VM version 1.6.0_17.

6.2. Microbenchmarks

We present results for several microbenchmarks that only perform I/O to explore the overheads of our transactional I/O implementation. We present numbers for four versions of the microbenchmarks: `trans`, a version that uses our transactional I/O implementation; `inevit`, a version that uses Inevitable I/O; and `lock`, a version that protects I/O with a global lock. We include for comparison an `unsafe` version that simply eliminates all concurrency control primitives and as it name suggests would be unsafe.

In the first microbenchmark, each thread uses its own file descriptor to seek to a disjoint location in a file, execute 300 transactions that each perform a single write operation, and then repeats this process until it has executed its share of the 900,000 transactions that the benchmark executes. Figure 2 presents results for this microbench. We see that the Transactional I/O version scales well to 4 cores and is faster than the inevitable version for 2 or more cores. The Transactional version is faster than the `lock` version for 4 cores. After 4 cores, the transactional version slows down. The `unsafe`

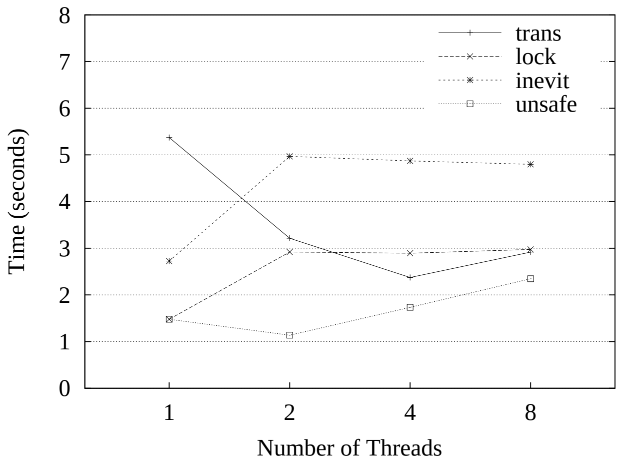

Figure 2: Times for Write-Only Microbenchmark

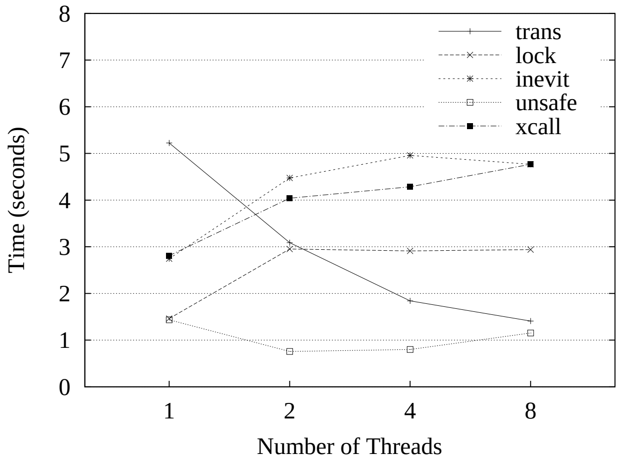

Figure 3: Times for Separate Files Write-Only Microbenchmark

version shows a similar behavior — increasing the number of cores from 4 to 8 also slows down this version revealing that the underlying cause is inside the Linux kernel. Further investigation reveals that Linux protects the mapping between files and pages at the file granularity — therefore writes to the file by multiple threads can cause conflicts inside the kernel.

We next modify the write-only, single-file microbenchmark to have each thread write to its own file. Figure 3 presents the results for this microbenchmark. With separate files the Transactional I/O scales to 8 threads. For this microbenchmark, we present results for an `xcall` version of the benchmark. This version is designed to simulate the behavior of the xCall approach to transactional I/O. Before writing, the xCall version first reads the data that will be overwritten in order to support rollback in case of a transaction abort. This simulation is lighter weight than a real xCall implementation would be — it does not actually acquire locks on files. We note that the transactional I/O version is significantly faster than the `xcall` version because the transactional I/O version does not need to perform read system calls to support rollback. Note that we omit the xCall comparison for other benchmarks as the threads in those benchmarks access the same files and therefore an xCall would serialize those transactions.

The reader may note that the multi-threaded transactional I/O version for the single-file write microbenchmark does not exceed the performance of the unsafe single-threaded version and the separate file version of the multi-threaded transactional I/O version is only approximately the same performance. It is important to note that the transactional I/O implementation enables more than one transactions to execute simultaneously. If like nearly all real workloads the transactions perform other operations in addition to I/O operations, then the transactional I/O can easily exceed the performance of the one-threaded version.

Real applications will execute transactions that mix computation with I/O. We next modify the write-only, single-file microbenchmark so that each

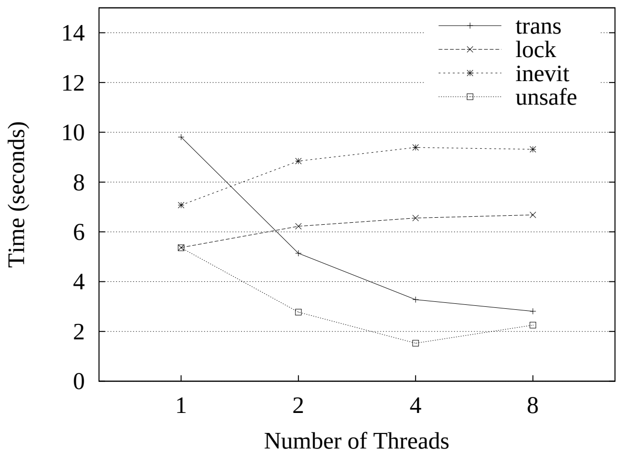

Figure 4: Times for Write-Only with Computation Microbenchmark

transaction performs a write and then computes the first 5,000 numbers in the Fibonacci sequence. This microbenchmark is designed to model a more realistic workload in which transactions mix computation with I/O operations. Figure 4 presents results for this microbenchmark. Both the inevitable and global lock versions are only able to execute one transaction at a time and therefore scale poorly. The `unsafe` version and the transactional I/O version are both able to execute multiple transactions simultaneously and achieve significantly better performance.

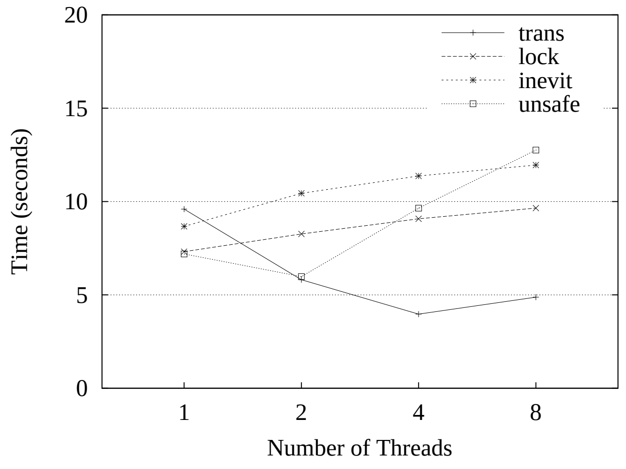

Figure 5: Times for Write-Only with 5 Writes

Transactions in many real applications are likely to perform more than one write operation to the output file. We next modify the write-only, single-file microbenchmark so that each transaction performs 5 write operations to consecutive locations in the file. Figure 5 presents results for this microbenchmark. The transactional I/O version is significantly faster than the other versions. The reason is that transactional I/O version makes fewer write system calls than the other versions.

Each thread in the next microbenchmark uses its own file descriptor to repeatedly seek to a disjoint location in the same file and then performs 300 transactions that each execute a single read operation. The benchmark performs a total of 900,000 transactions. Figure 6 presents results for this microbenchmark. The Transactional I/O scales well for this benchmark and is faster than all versions except the `unsafe` version.

Figure 7 presents results for a version of the read-only microbenchmark in which each thread accesses a separate file. Unlike the write microbenchmarks,

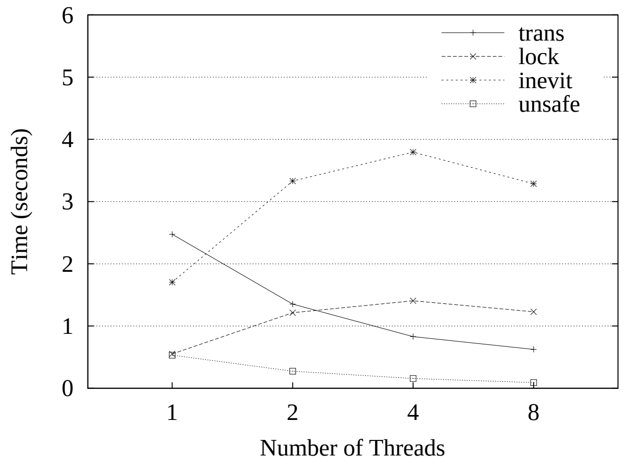

Figure 6: Times for Read-Only Microbenchmark

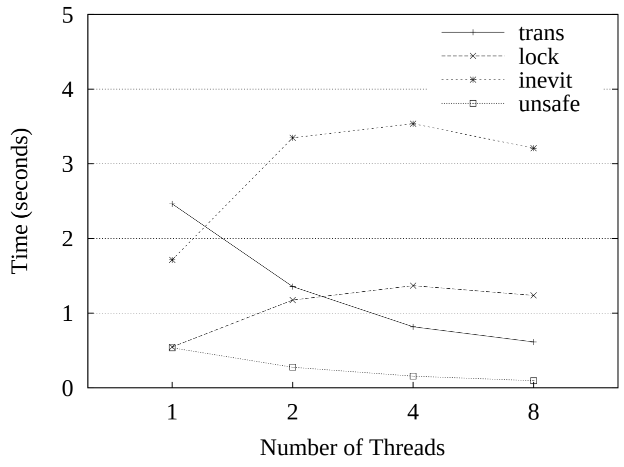

Figure 7: Times for Read-Only Microbenchmark from Separate Files

the results are nearly identical to the shared-file read benchmark. Linux kernels after version 2.6.27 implement a lockless page cache that eliminates locks for file read operations.

Figure 8 presents results for a version of the read-only microbenchmark that performs a read followed by a Fibonnacci computation. Both the inevitable and global lock versions are only able to execute one transaction at a time and therefore scale poorly. The `unsafe` version and the transactional I/O version are both able to execute multiple transactions simultaneously and achieve significantly better performance.

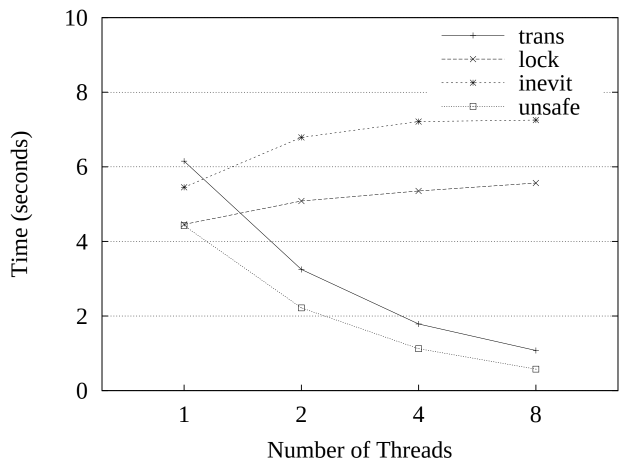

Figure 8: Times for Read-Only with Computation Microbenchmark

Transactions in many real applications are likely to perform more than one read operation from an input file. We next modify the microbenchmark so that each transaction performs 5 read operations. Figure 9 presents results for the multiple read microbenchmark. The general behavior is similar to the single-read microbenchmark.

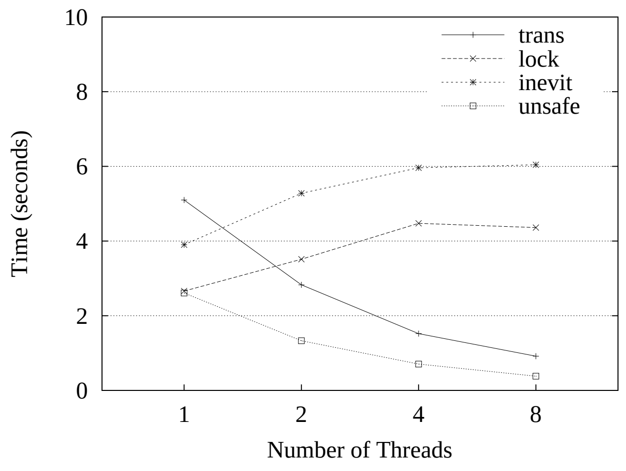

Figure 9: Times for Read-Only with 5 Reads

6.3. Benchmarks

We next evaluate our implementation on several benchmarks. We report numbers for `trans`, our transactional I/O implementation, and `inevit`, the inevitable implementation. When available, we report numbers for `lock`, lock-based implementations. These benchmarks require locking to execute and therefore unsafe versions are unavailable.

6.3.1. TupleSoup

TupleSoup is a Java framework for storing and retrieving simple hashes. The original source code is available at `http://tuplesoup.sourceforge.net/`. We ported the benchmark to the DSTM2 transactional memory system. We made modifications to eliminate transactional conflicts including changing the hash table implementation to avoid conflicts on the size field and table array.

The system uses an index file and multiple data files. Insertions into the table are appended to one of the data files. The tuples are randomly distributed between the data files. The index files keeps track of the offset in the record file that each tuple is stored.

Our workload for TupleSoup first constructs a table with 500 tuples with identifiers in the range of 0 to 500. The workload then performs 400,000 operations. Each operation randomly selects an identifier between 0 and 500. The operation looks up the identifier and then updates it.

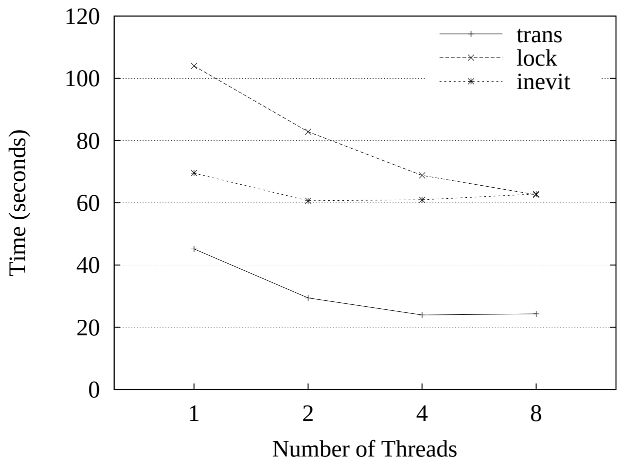

Figure 10: Times for TupleSoup

Figure 10 presents results for our implementation, the original TupleSoup implementation (`lock`, and the inevitable implementation for 1, 2, 4, and

8 threads. Inevitability serializes all transactions and therefore limits the amount of parallelism. The transactional I/O version performs significantly better than the original version of TupleSoup. Part of the reason is that the transactional I/O implementation performs fewer write system calls.

We observe that the transition from 4 to 8 threads reveals a scalability limitation for the transaction version. We measured the abort rates and found that less than 1% of the transactions aborted. To understand the source of this slowdown, we then instrumented the executable to record how much time was spent in transactions. We determined that much of the execution time was used by DSTM2 to allocate transactional objects. Figure 11 presents the results of this experiment.

	1 thread	2 threads	4 threads	8 threads
Object Allocation	23.7s	17.1s	17.0s	20.0s

Figure 11: Time spent Allocating Objects in TupleSoup

6.3.2. JCarder

JCarder is an open source tool designed to help developers find potential deadlocks in multi-threaded Java applications. JCarder operates by instrumenting bytecode dynamically and then looks for cycles in the graph of acquired locks. The original source code is available at `http://www.jcarder.org/`.

For the workload, we chose the dining philosophers benchmark that is distributed with JCarder. When threads in the workload application enter a synchronized block, the rewritten code invokes methods in JCarder. The code instrumented by JCarder records information about the locks to shared data structures and files. The data written in the files is later postprocessed by a separate phase of JCarder. After recording information about the run, the user runs the second phase of JCarder to discover information about possible deadlocks in the workload application.

Figure 12 presents execution times for JCarder. Note that the workload per thread is held constant and therefore perfect scalability occurs when the time remains constant as the number of threads is increased. The transactional I/O version matches the performance of the inevitable version for fewer than 4 cores and exceeds the performance of the 8 core inevitable version.

6.3.3. Financial Transaction

The financial transaction benchmark simulates a trading system. The benchmark maintains trading account records in a file. Each account record

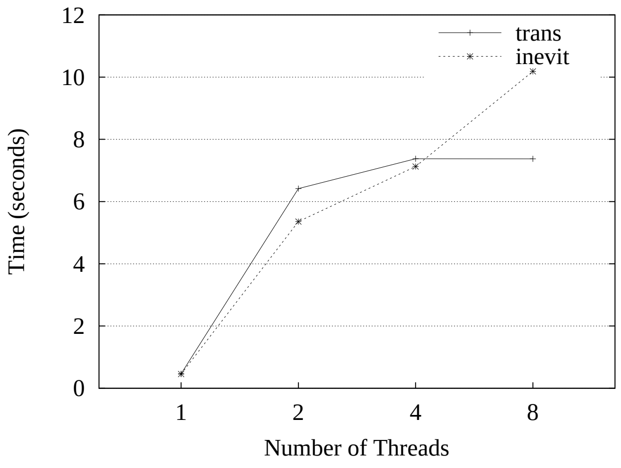

Figure 12: Times for JCarder on Dining Philosophers

maintains an inventory of stock holdings and cash. The input for the benchmark is a list of financial trades. Each trade consists of the names of the parties involved, the stock, the number of shares exchanged, and the price.

Each thread processes its own list of stock trades. For each trade, the benchmark modifies the account records for the two parties to the trade. The system also maintains a record of the last five financial transactions for each stock in a shared memory data structure.

Each thread uses a different file descriptor to read in the list of trades and therefore there is no contention on the list of trades. However, two trades do conflict if they both attempt to access the same block in the trading record file. We note that trades can also conflict if they both attempt to update the in-memory record of the last five trades for a stock. Figure 13 presents the results for both versions. For more than 1 core, the transaction I/O version is the fastest. Its relative performance improves with the transition from 2 to 4 cores.

The transition from 4 to 8 threads reveals a scalability limitation for the transaction version. We discovered that transaction aborts play a role in limiting the scalability of this benchmark. To quantify this limitation, we measured the abort rates of the transactional I/O version. Figure 14 presents these results.

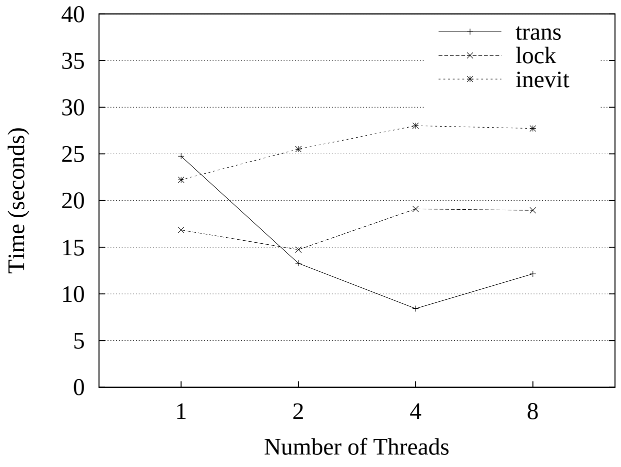

Figure 13: Times for Financial

	1 thread	2 threads	4 threads	8 threads
Abort %	N/A	3 %	9%	52%

Figure 14: Abort Percentages for Financial

6.3.4. Parallel Sort

The sort benchmark is designed to measure I/O performance. This benchmark accesses a list of words as input and sorts the words into files based on their first letter. This benchmark performs file reads outside of the transactions and the transactions write the output. Our transactional I/O implementation allows transactions to append to the same file without conflicting.

Our workload is an unsorted dictionary file. Threads seek to non-overlapping regions of the input file and start reading the words until they have read the predefined number of bytes. Each thread executes 10 transactions and each transaction reads 1,000 words from the input file and writes them to the corresponding output files. The thread finishes when it has read the predetermined number of bytes. Figure 15 presents execution times for the benchmarks. The transactional I/O version scales nearly perfectly for this benchmark. The other two versions are unable to execute transactions in parallel and therefore do not improve performance beyond the single-threaded version.

6.3.5. Parallel Grep

The grep benchmark searches an input file for a set of words and then logs the offsets of any appearances of the words in the output file. For each

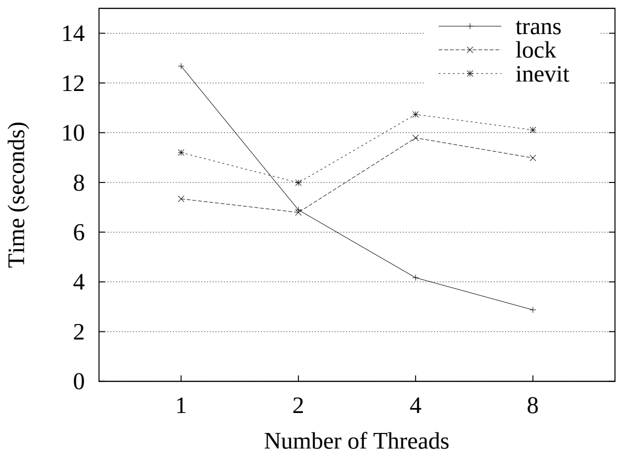

Figure 15: Times for Sort

of these words, it keeps a counter that tracks the number of occurrences of that word in the text. The threads seek to disjoint regions of the input file and start searching for the search terms. Upon finding a search term, the thread logs the success and increments the corresponding counter.

There is a conflict on the memory data structure whenever two transactions find the same word. The output file is an example of a logging pattern and the I/O operations never cause a transaction to abort. This is because the transactions only read from the input file and only append to the output file. For the input file each thread uses a different descriptor and hence there is no contention over the offset. Figure 16 presents results for the three versions of the grep benchmark. The transactional I/O version scales nearly perfectly for this benchmark. The other two versions are unable to execute transactions in parallel and therefore do not improve performance beyond the single core version.

7. Related Work

We survey related work in databases, transactional file systems, transactional I/O, and transactional memory.

7.1. Filesystems and Databases

Historically, database researchers have pioneered transactions [12]. Due to different requirements [13], techniques used for databases differ in many aspects from those of transactional memory. Software transactional memory

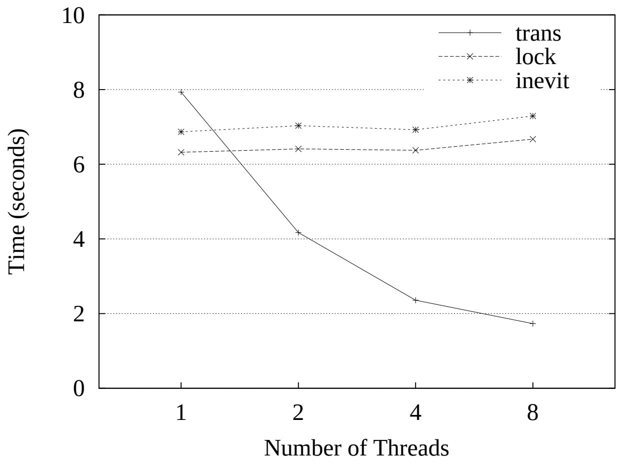

Figure 16: Times for Grep

systems largely ignore durability, while interactions with non-transactional code is not a challenge in databases or transactional file systems [14, 15, 16, 17]. Many distributed transactional file systems maintain multiple copies for files [18, 15]. While transactional file systems can provide rich and powerful primitives, operating system support for such file systems is rare.

7.2. Integration of I/O into TMs

The simplest approach is to prohibit I/O inside a transactional block [19, 20, 2, 21]. Obviously, this approach avoids the issue and leaves software developers with few good options if multiple threads share an I/O resource.

Another strategy is to make a transaction inevitable or irrevocable as soon as it calls an I/O system call, meaning the transaction will not be aborted by other transactions. The system can be implemented in hardware [22, 5, 23] or software [11, 3, 24].

The major drawback of these systems is the severe limitations they impose on the degree of concurrency. Some versions of the technique do not allow committing any transactions while an inevitable transaction is in progress, even if these transaction have no conflicts with the inevitable transaction. [22, 25, 2].

Note that file accesses turn the transaction into an inevitable transaction and forces all transactions that attempt I/O operations to either block or abort even if they access completely disjoint files. A second approach allows non-I/O transactions that have no conflicts with the inevitable transaction that is in progress [3, 5, 11, 26] to proceed. Transactions with side-effects (i.e. those with I/O operations within them) have to be serialized with respect to each other regardless of whether they actually conflict. This limits scalability. Moreover, the notion of inevitable transaction precludes using constructs such as "retry" in transactional programming languages and systems [27, 22].

Another approach is to provide programmers with the ability to register commit and abort handlers that define what sort of actions should be taken in case of an abort or a successful validation (i.e. the transaction is ready to commit) [28, 29, 30]. This approach demands that the developer reasons about the flow of the program and hence introduces a new source for bugs in the transactional application [26] and violates the benefits of implicit transactional programming.

Another strategy is to defer I/O [31]. The idea is to buffer the output and reflect the changes at commit and invoke the read system call during the execution and re-buffer the input in case of abort. This approach largely ignores the possibility of conflicts on I/O operations. This approach may be difficult to program as a transaction cannot see the changes it makes to the underlying file system.

Our approach differs from existing methods in that it guarantees serializability of both memory and file operations. To our knowledge no system provides strong atomicity for I/O operations. We provide strong atomicity that can avoid potential errors [32] and eases the job of the programmer.

Our approach is related to both transaction boosting [33]. Like boosting, our transactional I/O implementation leverages higher-level semantics to allow transactions that are conceptually linearizable but that might conflict at a low-level to commit. Similarly, multi-level concurrency control leverages semantic properties to improve concurrency [34].

8. Conclusion

Input and output has been largely overlooked by transactional memory research. We present a new fully transactional approach to I/O for transactional memory. We have formalized our approach, proved correctness properties, implemented the approach, and evaluated the approach on several Java applications. In many cases, the performance of our benchmarks was limited by current operating system bottlenecks. As operating systems evolve to better support many-core processors, many of these bottlenecks will be resolved and the performance of our approach will narturally improve.

Previous approaches either serialize all transactions that perform I/O or serialize all transactions that output to the same file and therefore ultimately limit the scalability of the application. Fully transactional I/O allows transactions to freely perform I/O operations, and our evaluation suggest that it will perform better than either of the previous approaches, xCalls and inevitable transactions, for real world applications.

References

[1] N. Shavit, D. Touitou, Software transactional memory, in: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing.

[2] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, III, Software transactional memory for dynamic-sized data structures, in: Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing.

[3] M. Olszewski, J. Cutler, J. G. Steffan, JudoSTM: A dynamic binary-rewriting approach to software transactional memory, in: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques.

[4] M. F. Spear, V. J. Marathe, W. N. Scherer III, M. L. Scott, Conflict detection and validation strategies for software transactional memory, in: Proceedings of the Twentieth International Symposium on Distributed Computing.

[5] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, S. Lie, Unbounded transactional memory, in: Proceedings of the 11th International Symposium on High-Performance Computer Architecture.

[6] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, K. Olukotun, Characterization of TCC on chip-multiprocessors, in: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques.

[7] R. Rajwar, J. R. Goodman, Transactional lock-free execution of lock-based programs, in: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems.

[8] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, A. Welc, xCalls: Safe I/O in memory transactions, in: Proceedings of the 4th ACM European Conference on Computer Systems, ACM, New York, NY, USA, 2009, pp. 247–260.

[9] M. Martin, C. Blundell, E. Lewis, Subtleties of transactional memory atomicity semantics, IEEE Computer Architecture Letters 5 (2006) 17.

[10] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: Proceedings of the 20th International Symposium on Distributed Computing.

[11] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, M. L. Scott, Implementing and exploiting inevitability in software transactional memory, in: 2008 International Conference on Parallel Processing.

[12] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

[13] P. Felber, C. Fetzer, R. Guerraoui, T. Harris, Transactions are back—but are they the same?, SIGACT News 39 (2008) 48–58.

[14] F. B. Schmuck, J. C. Wyllie, Experience with transactions in QuickSilver, in: Proceedings of the ACM Symposium on Operating Systems Principles.

[15] B. Liskov, R. Rodrigues, Transactional file systems can be fast, in: 11th ACM SIGOPS European Workshop, Leuven, Belgium.

[16] M. A. Olson, The design and implementation of the inversion file system, in: USENIX Winter, pp. 205–218.

[17] S. Quinlan, A cached worm file system, Software Practice and Experience 21 (1991) 1289–1299.

[18] J. Garcia, P. Ferreira, P. Guedes, The PerDiS FS: A transactional file system for a distributed persistent store, in: Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications.

[19] M. Herlihy, V. Luchangco, M. Moir, A flexible framework for implementing software transactional memory, in: Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications, pp. 253–262.

[20] K. Fraser, T. Harris, Concurrent programming without locks, ACM Transactions on Computer Systems 25 (2007).

[21] T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy, Composable memory transactions, in: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[22] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun, Transactional memory coherence and consistency (TCC), in: Proceedings of the 11th International Symposium on Computer Architecture.

[23] C. Blundell, J. Devietti, E. C. Lewis, M. M. K. Martin, Making the fast case common and the uncommon case simple in unbounded transactional memory, in: Proceedings of the 34th Annual International Symposium on Computer Architecture.

[24] A. Welc, B. Saha, A.-R. Adl-Tabatabai, Irrevocable transactions and their applications, in: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, ACM, New York, NY, USA, 2008, pp. 285–296.

[25] Y. Smaragdakis, A. Kay, R. Behrends, M. Young, Transactions with isolation and cooperation, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications.

[26] L. Baugh, C. B. Zilles, An analysis of I/O and syscalls in critical sections and their implications for transactional memory, in: IEEE International Symposium on Performance Analysis of Systems and Software.

[27] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. E. Kozyrakis, K. Olukotun, The Atomos transactional programming language, in: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation.

[28] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, K. Olukotun, Architectural semantics for practical transactional memory, in: Proceedings of the 33rd Annual International Symposium on Computer Architecture.

[29] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, D. A. Wood, Supporting nested transactional memory in logTM, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 359–370.

[30] J. E. B. Moss, A. L. Hosking, Nested transactional memory: Model and architecture sketches, Science of Computer Programming 63 (2006) 186–201.

[31] T. Harris, Exceptions and side-effects in atomic blocks, Science of Computer Programming 58 (2005) 325–343.

[32] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, B. Saha, Enforcing isolation and ordering in STM, in: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation.

[33] M. Herlihy, E. Koskinen, Transactional boosting: A methodology for highly-concurrent transactional objects, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[34] D. B. Lomet, MLR: A recovery method for multi-level systems, in: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data.