Speculative Messages

David Campbell
University of Texas
Austin, TX
campbeda@utexas.edu

Jeffrey Nelson
University of Texas
Austin, TX
jdn2283@utexas.edu

***Abstract*—The following paper describes an algorithm that can be used to implement speculative messages. Speculative messages allow applications to predict that message ordering and consistency requirements are maintained and manage when they are violated. Applications define sets of speculative operations called transactions and define the rules for rolling back speculative state. This paper argues that transactions can reduce protocol overhead for applications that have strict ordering requirements. They can also speed up applications that use a distributed mutex and have low lock contention. This paper implements two example applications that can benefit from transactions: a mock file system and a matrix swap algorithm.**

I. INTRODUCTION

In a distributed system, algorithms are often required to guarantee data consistency and message ordering among processes in the system, which results in a computational overhead that can grow increasingly large. For example, some mutual exclusion algorithms must acquire a globally distributed lock before a process is permitted to modify a shared resource. Algorithms also absorb overhead when they must enforce message delivery requirements such as guaranteeing causal and/or total order. These strict ordering requirements often require a trade-off in parallelism that leaves process nodes with idle CPUs.

In response to the above problems, this paper proposes a means for amortizing this overhead through the use of speculative messages. Speculative messages are messages that are sent and computed optimistically, i.e. before their operations are guaranteed to be safe. Rather than enforcing strict message ordering, operations are executed as they are received on each process node and are later aborted or committed. It is only after commit that the results are visible to clients. If an ordering or consistency violation is detected during speculative execution, a roll back to a consistent state is performed. To implement a rollback mechanism, speculative state must be tracked, which may result in an increase in memory consumption depending on the application. The computational overhead of performing a rollback must be minimized to avoid performance loss. When the number of rollback operations is small compared to the number of commits, the net performance of the overall system will compared to typical systems with larger protocol overhead. This performance gain will be described empirically, but conceptually, applications that can expect a performance improvement are those that exhibit the following properties:

- **Low lock contention** - when access to a shared resource is lowly contended, speculative messages avoid lock overhead by assuming there will be no write conflicts on the modified data. If a write conflict occurs, the speculative operations will be rolled back, and the application can try again or fall back to a lock.
- **Ordering overhead (causal/total ordering)** - as opposed to enforcing ordering before message delivery, speculative messages predict causal and/or total ordering will be maintained, otherwise the computation will be rolled back, and the application can try again or fall back to enforcing ordering.
- **High I/O latency & low I/O throughput** – writing to I/O devices such as hard-disks is often orders of magnitude slower than CPU operations. Speculative messages allow I/O writes to be performed before the write has been resolved, which takes advantage of device parallelism. *Speculator*[1] heavily exploits this parallelism.

The following are applications that we identified as exhibiting the above properties. To exercise our transactional memory platform, we chose to implement the first two applications.

- **Distributed File System** - speculative messaging can be used to pipeline disk writes on data modification before it is known if the message ordering requirements on different file system transactions are maintained.
- **Matrix Swap Benchmark** - speculative messaging can be used to randomly swap two different indices within a matrix and to benchmark the relationship between the number of processes and the number of successful/failed transactions.
- **Distributed Databases** - similar to distributed file systems, distributed databases benefit from speculative writes and parallel transactional commits.

II. SPECULATIVE MESSAGES USING TRANSACTIONS

To implement speculative messages, we designed a protocol largely based on transactional updates to shared resources. A transaction is a sequence of operations defined by the application that must guarantee mutual exclusion or message ordering properties. Applications can tune the size of transactions dynamically based on contention as well as true atomicity requirements. The application is responsible for calling the following methods to send a transaction:

- **tStart** – indicates the beginning of a transaction. Any operation following this message (FIFO channels) up to **tEnd** will be tracked speculatively. At this point, the transaction is allocated a unique (*process ID, transaction ID*) tuple.
- **tUpdate** - indicates an operation to send to peer process nodes. Peer process nodes look at tuple in message header in order to track this update in the context of associated transaction. Peer process nodes either silently execute update or send early abort to originating process node if collision has been observed.
- **tEnd** - indicates the end of the speculative computation sequence. If all process nodes acknowledge that the transaction occurred atomically, the transaction will be committed and visible to clients. If any process node observed a collision, the transaction will be aborted, and process nodes must roll back speculative state.

To provide the above semantics for an application, a *TransactionManager* was constructed and run on each process node. The transaction manager is responsible for managing transactions and calling transaction interfaces implemented by applications. The following pseudo-code shows our implementation of *TransactionManager*.

---

bool failed : init false;
int transactionId : init 0;
int myPid;
int commitAck[numProcesses];

On application tStart:
- *transactionId += 1*
- *failed := false*
- *∀i: commitAck[i] := false*

On application updateReq:
- *broadcast UPDATE(transactionId, myPid)*

On application tEnd:
- *if(failed): send ABORT(transactionId, myPid) && return false*
- *broadcast COMMIT_REQ(transactionId, myPid)*
- *wait for COMMIT_ACK from all process nodes:*
- *if(failed): send ABORT(transactionId, myPid) && return false*
- *broadcast COMMIT_FIN(transactionId, myPid)*
- *return true*

On UPDATE(transactionId, theirPid):
- *appResult := call application tUpdateReq(transactionId, theirPid)*
- *if(appResult == false):*
    - *send ABORT(transactionId, theirPid) to process with pid=theirPid*

On ABORT(transactionId, theirPid):
- *if(theirPid == myPid):*
    - *failed := true;*
    - *broadcast ABORT(transactionId, myPid)*

On COMMIT_REQ(transactionId, theirPid):
- *appResult := call application tCommitReq(transactionId, theirPid)*
- *if(appResult == true):*
    - *send COMMIT_ACK(transactionId, myPid) to process with pid=theirPid*

On COMMIT_ACK(transactionId, theirPid):
- *commitAck[theirPid] := true*

On COMMIT_FIN(transactionId, theirPid):
- *Call application tCommit(transactionId, theirPid)*

From the above algorithm, we provide the following guarantees:

- **Safety** - no transactions can modify the same shared resource in a way that violates total order, causal order, or consistency. Such conflicts will be detected.
- **Liveness** - every transaction is eventually serviced, but transactions are not guaranteed to ever succeed, even if they are retried forever.
- **Fairness** - our algorithm does not guarantee that transactions are processed in the order in which they are received. It only detects when ordering violations have occurred.

The message complexity added by our protocol is 3(N-1).

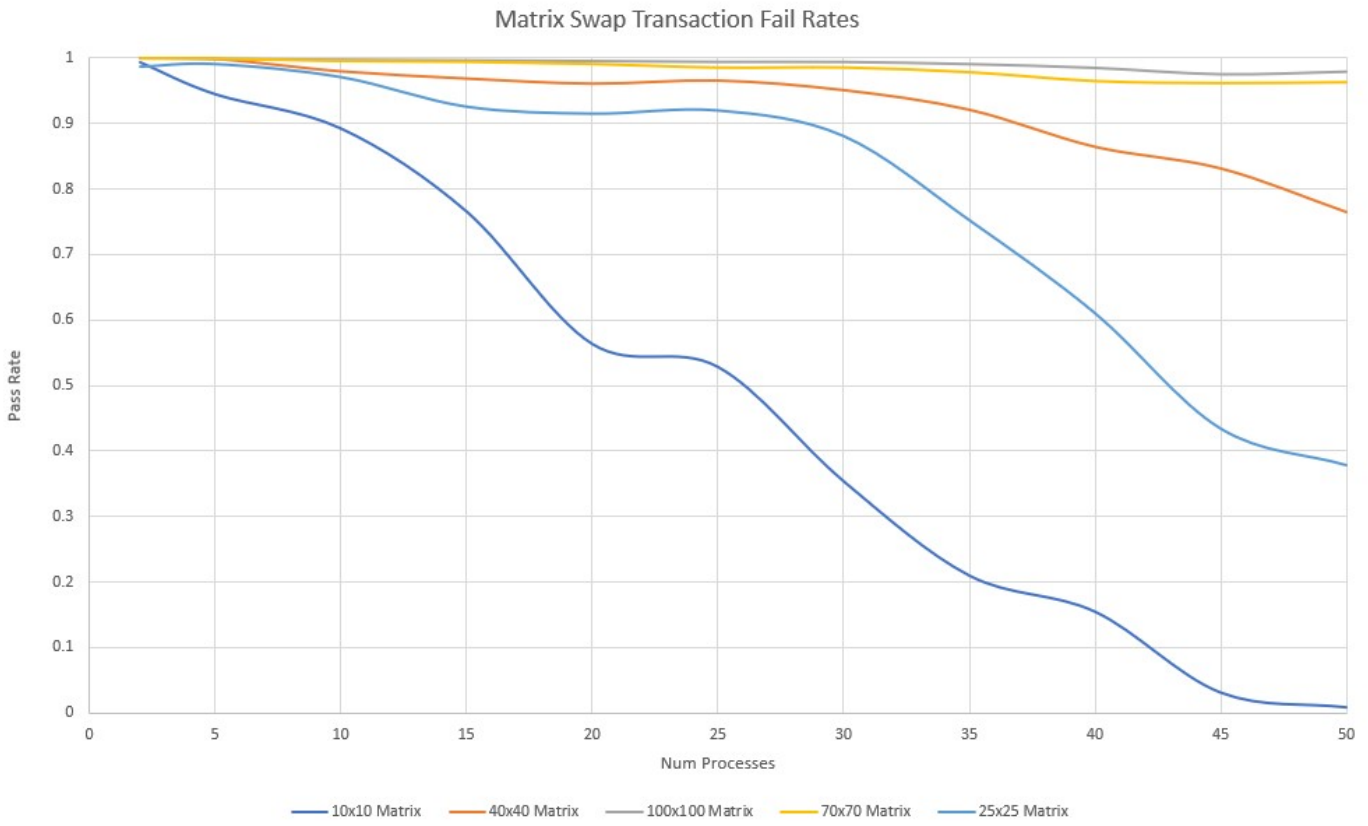

Figure 1 - Graph of Transaction Pass Rates

III. MATRIX SWAP BENCHMARK

A. Application

As an initial proof of concept, an application was constructed that built a *nxn* matrix and ran *m* processes. Each process ran several iterations of swapping two random elements in the matrix, where each of the two elements in the swap translates to one write command for the tUpdate method. Each of the two elements updated in the transaction were staged in a log, and if the transaction was committed, the information was swapped in the actual matrix. On failure, the information was removed from a log. Another log was also constructed to indicate if a given matrix element was in use by a transaction, which was cleaned up on transaction commit and transaction failure. This was used to detect transaction failures on update collisions.

Overall, atomicity of the swap was maintained by requiring all nodes in the system to agree that there was no collision on the order in which the elements of the matrix were being swapped before committing the actual results. The benefit of this approach is that many swapped transactions can occur in parallel between random data elements without having to acquire distributed locks for each matrix element. Overall, this would cut the message complexity in half.

B. Results

The above results were gathered by running 1000 transactions on each node and measuring the pass rate for all transactions. Overall, the experiment is showing that as the dataset size increases relative to the number of processes, the transaction failure rate goes down. Thus, more parallelism is achieved.

IV. MOCK FILE SYSTEM

A. Application

Our second application implemented a mock file system and applied transactions to the following file system operations: {*mkdir*, *rm*, *write*}. The file system provided a lot of unique challenges as the structure is inherently hierarchical. The following pseudo-data structures were used to implement a mock file system:

```
class MockFileSystem {
    inode root;
}

class node {
    String name;
    boolean isVisible;
    HashMap<> specOperations;
}
```

```
class inode extends node {
    HashMap<> children;
    HashMap<> files;
}

class dnode extends node {
    String data;
}
```

As a result, each file system operation had certain atomicity restraints to maintain:

- **mkdir <path>** - transactions that collide on mkdir for the same node do not need to report conflict as the end result is valid for both transactions
- **write <path> data** - transactions that collide on writing/creating a file (*dnode*) must report a conflict as writes must occur in the same order on every process node
- **rm <path>** – transactions that collide on rm for the same node do not need to report conflict as the end result is valid for both transactions
    - **rm targetInode** – if any *inode* or *dnode* in the tree rooted at *targetInode* has been speculatively written/created, transactions collide
    - **rm targetDnode** – if *dnode* has been speculatively written/created, transactions collide
    - **all operations** – if any inode along the path to target node is speculatively removed, transactions collide.

As speculative operations are executed, a per (*transactionId*, *pid*) map is maintained by the application in order to make COMMIT and ABORT as fast as possible. The abstraction here is that *inode*s and *dnode*s are always created speculatively (written to disk/backing store) but are not marked as visible to application until COMMIT. While they are in speculative state, file system grows until an abort comes in that requires application to quickly remove that speculative state. Since the application tracks which nodes are visible and which are not, an ABORT does not require the application to overwrite the data written to the backing store. It simply needs to drop the pointer to that location and allow it to be overwritten later.

B. Example

To better understand the implementation, let us walk through the following fully contained transaction from a single process.

```
tstart
mkdir /foo
write /foo/fileA hello
mkdir /foo/bar
tend
```

1. *tstart* is executed by the process initiating the transaction, assigns a unique transaction identifier for process *X*. Let's call this identifier *Y*.
2. *mkdir /foo* is tagged with (*X, Y*) and broadcast to all processes. Each process creates an *inode* for */foo* and marks it as not visible.
3. *write /foo/fileA hello* is tagged with (*X, Y*) and broadcast to all processed. Each process walks the path to */foo*, find that it is not visible, but continues because */foo* is tagged with *(X,Y)*. Each process then creates a *dnode* for *fileA* under */foo* and marks it as not visible. The data to be written to *fileA* is stored speculatively.
4. *mkdir /foo/bar* is tagged with (*X, Y*) and broadcast to each process. Each process walks the path to */foo* and creates a speculative *inode* for */foo/bar*.
5. *tend* is executed by originating process
    a. A COMMIT_REQ message tagged with *(X, Y*) is broadcast to all processes. Since no collisions have occurred, each process responds with COMMIT_ACK and updates speculative state to indicate that it has promised to commit (*X, Y*).
    b. Originating process collects ACKs and sends final commit message for (*X, Y*). On receipt, each process marks speculative nodes for (*X, Y*) as visible and clears speculative state for transaction.

The sequential case is straightforward, so for brevity, let us now consider the failure case from the perspective of a single process node. Consider the following received messages (in order) on process X:

```
(1, 1) mkdir /foo
(1,1) write /foo/bar hello
(2,1) mkdir /foo
(2,1) write /foo/bar goodbye
```

There is a potential for total order to be violated if every process does not see the same sequence for */foo/bar*. Handling is as follows:

1. Speculatively create */foo* and add (*1, 1*) to its speculative state.
2. Speculatively create */foo/bar*, add (*1, 1*) to its speculative state and cache *hello*.
3. Add *(2, 1)* to speculative state for */foo*. Note that no collision needs to be reported here.
4. Try to add *(2, 1*) to speculative state for */foo/bar* and detect collision with (*1, 1*). Send ABORT to P_1 with (*1, 1*) and to P_2 with (*2, 1*). Allow originating process to choose when/if to broadcast ABORT.

When *rm* is involved, the handling becomes slightly more challenging. Consider that */foo/bar/fileA* is a valid path and the following sequence is received:

(1, 5) rm /foo
(2, 3) mkdir /foo/bar/baz
(3, 7) write /foo/bar/fileA hello

In this case, *rm /foo* would speculatively mark */foo* as deleted and all subsequent transactions will quickly fail on the walk to their target *node*. Now that we have covered the basic operations, let us show a case where transactions outperform a distributed mutex approach. Consider the following sequence:

P_1: *mkdir /foo/bar/*
write /foo/one/fileA one
P_2: *write /foo/etc/conf/fileB xyz*
rm /foo/etc/conf2
P_3: *rm /foo/dev*
mkdir /foo/bar
write /baz/usr/fileC def

It should be clear that the above sequence has no true conflicts. With a distributed mutex used to guarantee total order, however, each process has to grab a lock for each *node* (or *subtree*) that it wishes to modify. Whether that lock is grabbed per-message or per-sequence, the process with the lock is unnecessarily starving out the other processes.

C. Metrics

The primary focus for the mock file system was consistency. The performance improvement comes from increasing I/O throughput by pipelining writes to backing store. Typical HDDs can deliver read/write speed of 80-160MB/s, while typical SSDs can deliver read/write speeds of 200-550MB/s[3]. These numbers are orders of magnitude slower than CPU operations and are on the same order of magnitude as network operations. Waiting for writes to be resolved by other nodes leaves the CPU idle and the disk unutilized, which slows down overall application latency and throughput. Throw in a single slow peer process node and the networking latency/protocol overhead slows down writes even further. For these reasons, file system performance can benefit significantly by increasing I/O throughput and decreasing network protocol.

V. RELATED WORK

In researching the usage of speculative messages, two applications that consistently appeared were file systems and databases. *Speculator*[1], which came out of the University of Michigan, sought to add speculative execution support to the Linux kernel and targeted NFS, a distributed file system, as an application to benchmark their improvements. Their goal was similar to our mock file system implementation, in that they sought to improve file system performance by masking I/O latency and increasing I/O throughput. The major difference was that they used a centralized master node to detect and resolve conflicts. They determined that locally checkpointing an application's state was significantly faster than any operation that required inter-process communication (IPC), and that many file system operations can be easily predicted. Their results show an average performance improvement of 2x compared to NFS.

Another distributed system, this time a database called CockroachDB[2], sought to improve database transaction management by decreasing networking protocol overhead while still maintaining the speculative nature of transactions and the ability to compute transactions in parallel. They found that network latency became a major scaling factor for a globally distributed database. Their transaction model differs fundamentally from our implementation, but their use and explanation of write intents helped us define what speculative state each node needs to track and how to handle data that has been "promised to be committed" but not yet committed.

VI. FUTURE WORK

To better evaluate the performance improvement for the transactional mock file system, its results should be compared against the same mock file system using a distributed mutex algorithm. Further extension could allow the user to specify that the application dynamically fall back to a distributed protocol that guarantees fairness, like Lamport's distributed mutex, during times of high resource contention or in the presence of a specific node getting starved.

For the transactional protocol itself, future work could decrease the message complexity to 2(N-1), possibly by piggybacking the TEND message. Another approach to decrease the message complexity could be to run each transaction in its own round, meaning that applications have a defined amount of time to send an ABORT message to the transaction originator, otherwise the COMMIT carries through. This approach has its own complexities and caveats, but it is worth investigating. On the same note, future work can extend the transaction management to handle node failures and node insertion.

VII. CONCLUSION

Our goal in implementing a speculative messaging framework was to demonstrate that applications that exhibit certain properties, such as low lock contention and low I/O throughput, can see a performance improve-

ment while still enforcing strong consistency and ordering requirements. Detecting such ordering and consistency violations can result in less overhead than enforcing such constraints when the number of violations is relatively small. From an application standpoint, the majority of transaction management can be abstracted away, requiring only the implementation of a set of transactional interface APIs. By allowing an application to define its own transactions, it can tune transaction size to maximize performance. Since message complexity applies to a transaction rather than a single message, the protocol overhead can be amortized as the transaction size increases. There are some tradeoffs, however, as large transactions can increase the frequency of conflicts. Furthermore, transactions do not provide fairness guarantees and are not guaranteed to ever succeed, even if consistently retired. For real world applications with low lock contention, this is likely a worthy tradeoff.

The *Matrix Swap* application demonstrated the potential scalability and performance gains for a system built with transactional messages, while the *Mock File System* demonstrated a real-world system that has already been shown to exhibit properties that can benefit from such a system. With more future work, this platform can provide a straightforward and modular solution for applications to test whether speculative messages can be used to improve performance. https://people.eecs.berkeley.edu/~brewer/cs262/speculator-nightingale.pdf

VIII. REFERENCES

1. Speculator: https://people.eecs.berkeley.edu/~brewer/cs262/speculator-nightingale.pdf
2. CockroachDB Parallel Commits: https://www.cockroachlabs.com/blog/parallel-commits/
3. HDD vs SDD Blog: https://tekhattan.com/blog/hardware/ssd-vs-hdd-speed-lifespan-and-reliability/