ABSTRACT

With the drastic reduction in price of fast memory, database systems have begun to favor its use over the use of traditional hard disk storage. However, although this increased use has led to the development of systems that have been able to achieve previously inconceivable throughput, the volatile nature of the storage layer has negatively impacted the durability of a system. Several checkpointing mechanisms have been developed to periodically write snapshots of the contents of main-memory to stable storage and prevent data loss in case of power outages, server attack, or other disruption. However, many of these schemes rely on the infrequent occurrence of these snapshots, triplication of the data layer, and physical points of consistency in the system. We discuss herein a checkpointing scheme which relies on the guarantee of a predetermined serial ordering to capture snapshots of the in-memory storage layer with a mere ten to fifteen percent reduction in total transactional throughput and at most a duplication of memory’s contents. This work is extended from research performed on Calvin, an architecture for a distributed storage system that relies on a deterministic ordering guarantee to support distributed transactions, while maintaining linear scalability and having no single point of failure.

1. BACKGROUND AND INTRODUCTION

Traditional database tautology has sought to ensure that any database system maintains so-called ACID-compliance. This model seeks to ensure that all transactions processed in a storage system are atomic, consistent, isolated, and durable [1]. The final characteristic, durability, refers to the fact that any transaction that has been committed to the database must be recoverable in the event of a node failure [8].

The increased availability and dramatically reduced cost of high-speed random-access memory, which is generally several orders of magnitude faster than hard disk storage, has resulted in the widespread use of database systems that are executed mostly or entirely in main memory [6]. In order to avoid data loss that necessarily occurs when volatile memory is reset during a node failure, several checkpointing protocols have been developed to periodically write the contents of memory to disk. ARIES [11], often considered the golden standard for checkpointing, uses write ahead logging along with redo logging and logical undo operations to recover a node that has experienced some form of failure. Recent improvements on this highly generalized method for database recovery have focused on leveraging specific aspects of the system they operate in to reduce the amount of time spent capturing a global snapshot. For example, Cao et. al discuss Ping-Pong and Zig-Zag [3], systems that achieve extremely short checkpoint periods in frequently consistent applications. However, this protocol relies heavily on the assumption that the database is guaranteed several instances in time where all transactions are committed and no effects of uncommitted transactions are reflected in the data layer. These are referred to as “physical points of consistency” and, although often found in common applications such as massively multiplayer online games, limit the frequency with which checkpoints can be captured.

Simultaneously, several popular distributed storage systems have begun to depart from consistency guarantees across replicated data centers. These products, including Google’s BigTable [4], Amazon’s Dynamo [5], and Facebook’s Cassandra [9], use the CAP theorem [7] to explain their non-compliance with desired ACID properties. This theorem states that reduced guarantees in cross-replication consistency are the only manner in which the system can remain globally available around the clock. Reduced guarantees of consistency in a distributed, multiply replicated system further complicate the ability to capture a global snapshot.

However, recent work has signaled a return to traditional views on the need for databases, even those replicated and distributed, to be ACID-compliant. Calvin [12][13], the distributed and synchronously replicated storage system this checkpointing scheme is implemented as part of, achieves global consistency through a replication of inputs rather than effects, avoiding the prohibitively expensive contention costs that had previously impeded the prevalence of systems supporting
Our protocol is based loosely on work developed on multi-versioned “historical queries” in the HARBOR [10] recovery and failover system, as well as the notion of points of consistency exploited by Ping-Pong and Zig-Zag[3]. We present herein a method where, when a serial ordering of transaction inputs is guaranteed, global system checkpoints can be captured without stopping the database’s execution, while only requiring at most a duplication of the storage layer. Furthermore, because the protocol relies on guarantees of a serial transaction ordering, only a “virtual” point of consistency is required, rather than a precise moment in time at which the entire data layer is consistent.

In section 2, we discuss the detailed design for the novel checkpointing scheme. Section 3 discusses how the prototype for the scheme was developed and section 4 discusses the prototype’s impact on total transactional throughput in Calvin. We discuss the work planned for this project during the Spring in section 5. In section 6, the experience of conducting this research in the context of the Special Projects course is discussed. Finally, in section 7, we conclude.

2. CHECKPOINTING IN CALVIN

The capability for a database system to handle failure, for example due to power outages or accidental server interrupts, is absolutely vital to its operation. Deterministic database systems have two basic advantages that greatly facilitate fault handling. First, highly replicated deterministic database systems enable “active replication,” so that the same transaction processing can happen entirely in parallel across replicas without direct communication between the replicas. As long as replicas receive the same inputs, they can process transactions independently and not diverge. This enables on-the-fly failover to a replica upon a failure.

Second, since transactional input is sufficient to entirely determine the final state of a database, only the transactional input must be logged (instead of the high overhead physical REDO logging presents in most modern database implementations). Replaying history of transactional input is therefore sufficient to recover the database system to the current state (transactional input is logged during the process in the sequencer that synchronously replicates the global transaction sequence). However, it is inefficient to replay the entire history of the database from the beginning of time upon every failure. Therefore, it is necessary to occasionally checkpoint the state of a database in order to provide a starting point for the history replay.

2.1 Protocol Architecture

With an agreed upon global serial order to which replicas are guaranteeing execution equivalence, the work
Figure 2: Execution of Checkpointing Protocol in Calvin
by Cao et. al. [3] can be applied even though there may never be a strictly physical point of consistency. Rather we can establish a virtual point of consistency at an arbitrary point in the global serial order, and capture a snapshot with regard to that demarcation. Intuitively, we can imagine a virtual point of consistency in a transactional database as a guarantee, stating that for some physical point in time \( P_1 \), all transactions that have an ID less than some specified value \( T_1 \) will have been committed to the database. Several transactions may still be in the middle of execution at this point in time, but the value of any record in the database at or before \( T_1 \) is stable.

We can therefore implement a checkpointing scheme similar to Zig-Zag. Although the specifics of the data layer are flexible, it is easiest to visualize the database as an associative key-value store where values may be chained. As shown in figure 2, the data layer receives in-place updates and overwrites the singular value maintained at \( Key \), similarly to any current storage system. Section (b) shows the value at \( Key \) after being updated from a value specified by transaction one \( (T_1) \) to a value specified by transaction two \( (T_2) \).

Now, assume that at some arbitrary time before \( T_4 \) begins executing that some control layer announces to the database that it should prepare for a checkpoint to occur at the virtual time represented by \( T_4 \). Any value updated by a transaction preceding \( T_4 \) or \( T_4 \) itself is updated in-place. However, as is shown in section (d), when a transaction following \( T_4 \), such as \( T_5 \), attempts to update the value at \( Key \) it is merely prepended to the chained list. Subsequent updates to \( Key \) overwrite this value, resulting in the maintenance of at most two values for any key (namely a stable version to be written to the checkpoint, and an unstable one).

At some unspecified later point in time, as depicted in section (f), the control layer announces to the database that it should begin a capture of the checkpoint. Upon receiving this message, the datastore accepts it as fact that the virtual point in time represented by the checkpoint it has been preparing for has passed. Because this is the paradigm followed, Calvin’s sequencer served as the control layer. The sequencer, as part of its duties, keeps track of when specific transactions have finished executing. Therefore, it can announce with certainty when it is guaranteed that a specific transaction has been fully committed or aborted.

Once the “capture” message has been received from the control layer, the database spawns an asynchronous writer, as shown in section (g), that begins placing the values at each key onto a stable media. It is important to note that although the writer is interacting with the database layer, the presence of a stable and unstable version precludes the need to acquire locks when capturing the snapshot. As shown in section (h) of figure 2, if a transaction, say \( T_7 \), updates the value at \( Key \) that update occurs normally, affecting only the unstable value in the chained pair.

Finally, the asynchronous writer is responsible for pruning the value of each key captured after it is finished writing to disk. Note that this pruning only occurs if there is a stable and unstable version of the key. If no unstable version exists then the stable version must not be pruned so that reads can proceed as normal.

2.2 Failover and Recovery

Using the system checkpoint, failover and recovery of a node is trivial. Given the failure of a single node in a single replica, the system routes all read requests within that replica to a nearby replica. When the failed node comes back online, it restores its state using the most recently captured checkpoint, and then uses the input transaction sequence to replay the history of all transactions that modify the state of the node until the node is caught up.

3. IMPLEMENTATION

The entire system was implemented as an extension of the Calvin storage layer. Calvin uses a simple std::unordered_map to keep track of an associative key-value store that represents the database. The checkpointing scheme was implemented as an abstracted component that contained a reference to the current Storage and operated on it. However, it is easiest to conceptualize the checkpointer as an interface that intercepted CRUD calls and modified the data layer accordingly.

Figure 3 shows relevant pseudocode for the implementation of our checkpointing scheme. It is important to observe that the checkpointer and control layer interact via a publish-subscribe model in order to promptly handle changes in the status of the current snapshot capture. Lines 3 and 5 of the Sequencer::HandleCheckpointing method show the calling thread having to wait on the checkpointer to provide a notification in order to proceed. The notification for the latter event is provided in line 5 of Storage::CaptureCheckpoint.

Figure 4 provides an event-based visualization of how the checkpointer intercepts CRUD commands and dispatches them to the proper storage layer. Although the Delete method is typically only called by the application to delete unstable keys, we maintain a time-versioned implementation of Delete so that the asynchronous writer can make a simple call to this in order to prune the extra storage during the checkpoint process.

4. EXPERIMENTAL RESULTS

As discussed in Section 1, ensuring that checkpoints of the database can be captured with only a modest
reduction in total transactional throughput is crucial. Figure 5 shows that our implementation of checkpointing, requiring an asynchronous writer and two versions of any data record, does not reduce our throughput to an unacceptable level.

When the checkpoint period begins, we notice an appreciable decline in total throughput, corresponding to an approximately twenty percent reduction in total throughput. Much of this can be attributed to the overhead associated with initializing the asynchronous writer’s thread, which is not efficiently garbage collected in our implementation. It should be noted that this level of throughput is the lowest that is ever reached by our system. In fact, halfway through the checkpointing period, the database system is operating at only a ten percent reduction in total throughput.

Finally, writing stable values to storage asynchronously does not increase contention or latency appreciably. Because the asynchronous writer is only processing data items that cannot be accessed by the mutator thread, both the mutator and checkpointing threads can retrieve a pointer to the multi-versioned linked-list record simultaneously and modify one of its versions.

What has made the use of such asynchronous workers so appealing is the fact that several machines are currently designed with more cores than can possibly be utilized by large database systems, and so performing the asynchronous actions on a newly spawned thread results in a negligible footprint. In fact, the problem of wasted cores is exacerbated in Calvin, where contention due to locks can be so high that the total throughput actually decreases when too many threads are allocated to perform mutation. Moreover, due to the ability of Calvin to scale near-linearly, it is highly logical to shard the storage onto several commodity nodes that can allocate extra threads to continual checkpointing. Capturing these continual checkpoints is essential, since these commodity nodes are far more likely to fail than high-end hardware. Furthermore, our scheme limits checkpoints only to the rate at which they can start and stop, so performing continual checkpoints does not present an issue.

5. FUTURE WORK

Several things remain to be completed during the Spring. First and foremost, the protocol established herein will have its implementation completed. Due to time constraints related to the submission of our architecture to SIGMOD, only a simple, proof-of-concept implementation was able to be programmed. Furthermore, the scheme must be compared to thorough implementations of standard checkpointing methods such as Ping-Pong and Zig-Zag to establish its relative performance.
Storage::ReadObject(Key, VirtualTimestamp) {
1: if VirtualTimestamp < VConsistentPoint_ then
2: return stable-value Key
3: else
4: return current-value Key
5: end if
}

Storage::PutObject(Key, Value, VirtualTimestamp) {
1: if VirtualTimestamp < VConsistentPoint_ then
2: update-stable-value Key, Value
3: else
4: update-current-value Key, Value
5: end if
}

Storage::PrepareForCP(VirtuallyConsistentPoint) {
1: VConsistentPoint_ = VirtuallyConsistentPoint
}

Storage::CaptureCheckpoint() {
1: for each Key in Datastore do
2: StableValue = Storage->ReadObject(Key, VConsistentPoint_)
3: write-to-disk Key, StableValue
4: end for
5: NotifySequencer()
}

Sequencer::HandleCheckpointing() {
1: loop
2: Storage->PrepareForCP(TransactionNearInFuture)
3: WaitForTransactionNearInFutureToPass()
4: Storage->CaptureCheckpoint()
5: WaitForStorageNotification()
6: end loop
}

Figure 3: Checkpointing Pseudocode

Although it is mentioned that recovery and failover are trivial, it is still important to create an actual implementation of failover and recovery to confirm the hypothetical ease with which they can be programmed. In addition to proving that implementing recovery and failover is simple and not prohibitively expensive, it would be excellent if we had time to explore the possibility of implementing intelligent online failover algorithms. For example, in a multiply replicated storage system, it would be important to determine not only where remote reads sent to a failed node should be rerouted to, but exactly what the minimum work that can be done in rerouting is, and if certain remote reads can be avoided.

Furthermore, several optimizations to the partially completed checkpointing scheme presented herein could be implemented. When the asynchronous writer begins searching through the versioned data layer for values to write to disk, at a bare minimum it need only capture those values that have been modified since the last checkpoint capture. The addition of so-called “dirty-bits,” which would indicated whether a record had been modified since the last snapshot was captured, could be added to each record to decrease the amount of time spent performing slow disk writes. Another version of this check would use a bloom filter [2] to quickly determine whether or not it was likely that the value at a given key had been updated. This implementation would test whether the time gained by the fast hash was greater than the amount of time spent writing false positives to stable storage.

6. RESEARCH EXPERIENCES

The experience of conducting this research as part of the Computer Science Special Projects course was excellent. I had begun work on Calvin during the Spring of my junior year, so being able to jump back into a project and immediately begin making contributions was incredibly exciting. However, in recommending this course to fellow seniors I would caution them to do as I did and pick an area of research that they already possess a considerable amount of knowledge about. Each semester at Yale goes by incredibly quickly and if, in addition to creating a prototype of sufficient importance, a person had to get formally acquainted with an unknown subject in Computer Science, the Special Projects course could very quickly become overwhelming.

6.1 Code Maintenance

The entire codebase for this project is maintained via a Gerrit repository available online to contributing members. Automatic documentation, generated by Doxygen, is also available online to project contributors. Being able to undergo actual code review with my peers has been an extremely edifying and rewarding process. It has helped achieve a personal goal of utilizing this Special Project as a means by which I can become a more efficient, skilled programmer.

6.2 Conference Paper Submission

Although not required for the Special Projects coursework, as part of this research our group submitted a paper to the proceedings of SIGMOD. This, although a
stressed experience, allowed me to keep my research in perspective and find a greater, more impactful meaning for what I had spent the entire term working tirelessly on. I would recommend to my peers that, when taking this course, they attempt to work on a project that will likely be part of a conference paper submission so that their research can be framed in a meaningful context and be compared against the work of respected leaders in the field.

### 6.3 Issues Encountered

Several issues were encountered throughout the course of this research project. First and foremost, the system on top of which the checkpointing scheme was developed, the Calvin prototype, was extremely unstable. Making the system stable enough so that snapshots could actually be captured and accurate data about total throughput recorded consumed most of the time working on this project.

At the implementation level, one of the major issues was, and will continue to be, how to provide a proper comparison to the Ping-Pong and Zig-Zag checkpointing methods. At its core, Calvin uses a simple `tr1::unordered_map` to represent an associative key-value store. However, both Ping-Pong and Zig-Zag rely on the use of in-memory byte-arrays, which have significantly higher random access times. These will need to be reconciled during the research period in the Spring.

### 6.4 Lazy Evaluation of Writes

This report constitutes an interim report for progress on a year-long body of research into novel methods for desirable database extensions such as checkpointing. In the Spring, however, not only will the research on our novel checkpointing scheme be finalized and submitted to the proceedings of VLDB, separate research on lazy evaluation in a data system will, if time permits, be conducted.

For this secondary project, we aim to measure the effects that lazy evaluation in a database has. The motivation for this is slightly hyperbolic, but could be generalized to several common use cases. Assume that an individual runs an online holiday shop that receives several tens of millions of transactions during the month of December, but only one million transactions during the other eleven months of the year. In this case, the total throughput of the database with respect to writes and in-place updates must be extremely elastic. Consider a transaction that specified, for given records \( x \) and \( y \) at time \( t \), that the value at \( x \) should be set to \( x + y \). Traditional systems would perform two queries to obtain the values of \( x \) and \( y \), sum their values, and update the value at \( x \) to reflect the sum. However, using a multi-versioned store we can reduce the total queries required in performing the in-place update by simply placing a lambda representing \( x_t + y_t \) into the multi-versioned row \( x_{t+1} \). Then, only when a read of \( x_{t+1} \) is performed is the actual value computed and placed into record \( x_{t+1} \). The ability to reduce the total number of queries required for in-place updates or writes would drastically improve the total throughput of the system for, say this holiday shop during December.

The implementation of the aforementioned system is, however, not trivial. Several considerations need to be taken into account including:

- How are a transaction’s effects and all dependent lambdas rolled back when an abort occurs?
- Is a reference counter necessary to prevent the storage size from exploding?
- How would the reference counter be updated quickly and atomically?

If time allows this system would be implemented and these questions investigated and, hopefully, solved.

### 7. CONCLUSION

We have presented a novel method for checkpointing that requires at most a duplication of the data layer, captures a checkpoint with a modest ten to fifteen percent reduction in total transactional throughput, and does not require a physical point of consistency. The prototype was implemented in the context of the Calvin codebase assembled for experimentation, as documented in [12].

It is important to note that although only demonstrated on Calvin, this protocol can be run on any storage system that has guarantees about the monotonically-increasing nature of transaction completion time. It remains extremely important to perform more research in the future into whether this scheme can be generalized to transaction processing systems that interleave database reads and writes, and therefore have no guarantees of monotonically-increasing times of completion.

As the cost to produce memory becomes cheaper and cheaper the pervasiveness of data storage systems that rely partially or entirely on storing records in volatile media will increase. In these systems, checkpointing is perhaps the most vital component to be integrated into the system so that data loss due to node failure is kept at an absolute minimum and overall time spent performing recovery and failover is decreased, improving the system’s availability. I look forward to continuing this line of research next semester.

### 8. ATTRIBUTION AND THANKS

Thanks are due to my thesis advisor, Professor Abadi, for his continued support on this project. Moreover, thanks to Alex Thomson, Philip Shao, Kun Ren, and...
Shu-Chun Weng for their tireless work on completing the Calvin prototype.

9. REFERENCES


