CALC: A Fast Method for Checkpointing Asynchronously in Arbitrarily Configured Main-Memory OLTP Systems

Thaddeus Diamond
Yale University
diamond@cs.yale.edu

I. 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 [2]. 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 [10].

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 [8]. 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 [16], 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 [5], Amazon’s Dynamo [6], and Facebook’s Cassandra [13], use the CAP theorem [9] 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 [18][19], 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 distributed transactions.

Our protocol is based loosely on work developed on multi-versioned “historical queries” in the HARBOR [14] 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.

The rest of this paper proceeds as follows. In Section II we discuss some background that we will leverage in accomplishing the goal of capturing cheap, fast snapshots in a database system. In Section III, we describe CALC, our contribution to snapshot capture in OLTP systems. Section IV presents our experimental setup, and Section V describes the results observed in our experiments. Finally, in Sections VI and VII we examine related work and conclude.

II. BACKGROUND

When we refer to checkpoints, we denote a set of \((key, value)\) pairs that are written out to stable storage. These pairs represent the state of the database, also referred to as application state, at a given point in time, \(t\).

Any algorithm used to capture a checkpoint and later recover the state of the database has a set of desirable properties as follows:

1) The process must not significantly impede the rate at which the database can perform read and write operations...
2) The process must not introduce unacceptable latencies in query execution into the system
3) The process must not assume that there is an abundance of available volatile storage (i.e. double or triple the size of the application state)
4) Recovery from a checkpoint must not be prohibitively expensive in terms of time taken to “catch up”

A. Physical and virtual points of consistency

A database is defined as being at a point of consistency if it reflects all changes made by all committed transactions, and no change made by any uncommitted, in-progress transaction.

Some applications (like many massive multiplayer online games) have natural points of consistency in their workload. Other applications do not, but it is always possible to force the database system to reach a physical point of consistency by quiescing the system entirely—not allowing new transactions to enter the system until the active transactions finish and commit or abort. Cao et. al. provide several algorithms for checkpointing applications that reach frequent finish and commit or abort. It is always possible to choose a point in that order, and expose the actual transaction commit order. Once this is known, it is possible to achieve fast asynchronous checkpoints even in the absence of physical points of consistency.

We introduce the notion of a virtual point of consistency. As described in section I, each transaction receives a monotonically increasing transaction ID based on the time it was entered into the system. A virtual point of consistency exists at a particular transactional number when all changes to data by transactions lower than this number are guaranteed to have completed. No guarantees are made about transactions greater than this. To find a virtual point of consistency without quiescing the system, it is easiest simply to know the actual transaction commit order. Once this is known, it is always possible to choose a point in that order, and expose only the writes performed by transactions that precede that point.

There are three obvious ways to achieve a virtual point of consistency:

- **Full multiversioning.** In systems implementing snapshot isolation via MVCC, a view of the database at a recent point of consistency is always available simply by choosing a timestamp preceding the start of the oldest currently active transactions and viewing all of the latest versions of each record whose timestamp precedes the one chosen.

- **Write buffering.** If transactions buffer their writes and only apply them (all at once, and atomically) to the underlying datastore at commit time, the underlying datastore is always kept in a physically consistent state. Write buffering is used in optimistic concurrency control schemes [12], and while OCC is seldom used in traditional database systems, software transactional memory (STM) implementations often use optimistic schemes[1].

- **Deterministic execution.** In deterministic systems, transaction execution order is explicitly known in advance, so determining which updates occur before a specified point in time simply requires looking at the location of the transaction that performed the update.

Our goal in this paper is to enable checkpointing virtual points of consistency without requiring a full multi-versioned database system, since main memory is still expensive, and requiring that many copies of the data be held in memory can be problematic. The checkpointing implementations in this paper are designed to work inside a deterministic database system, since recent research results indicate that determinism is a promising direction for main memory databases [19], [18]. Therefore we rely on deterministic transaction ordering to expose virtual points of consistency. However, write-buffering would allow our techniques to be effectively applied to traditional (nondeterministic) systems employing single-version stores and lock-based concurrency control.

Although buffering writes introduces significant overhead to transactions, only a small fraction of total transactions would actually have to buffer their writes in order for a virtual point of consistency to be exposed. Consider, for example, a scheme in which when each transaction enters the system, it decides whether it will buffer its writes until commit time or apply them directly, and then holds to that decision. By default, transactions do not decide to buffer their writes, but when a virtual point of consistency is needed, a flag is set such that all new transactions execute in write-buffering mode. Execution continues until only write-buffering transactions are still active (i.e. until all transactions that were active at the time the flag was set have completed). At this point, we can safely expose a virtual point of consistency, and then unset the flag so that subsequent transactions will not buffer writes—at least until the next checkpoint is requested.

B. Determinism

Traditional database system designs generally allow a number of highly nondeterministic behaviors. In particular, the serializable isolation level promised by ACID systems guarantees that concurrent transaction execution is actually equivalent to some serial execution of the same set of transactions, but it is ambivalent as to which serial execution. While this allows systems to reorder conflicting transactions at execution time, this nondeterministic behavior comes with certain costs in the areas of system replication and supporting distributed transactions.

Our recent work on determinism in database systems has explored some of the historical reasons why database system architects in the 1970s and 1980s made the design decision to allow transaction reordering and other nondeterministic behaviors [19]. The primary motivation appears to have been that on-the-fly reordering is extremely important for achieving high levels of concurrency and good CPU resource utilization when transactions are likely to stall due to disk IO latency. Only in recent years has the falling cost of memory raised the possibility of implementing systems in which...
transactions seldom or never block on disk IO, allowing the question of deterministic execution to be revisited.

Calvin, our deterministic transaction-processing framework prototype, achieves excellent replication properties and supports arbitrary distributed transactions at low cost [18]. By choosing in advance of execution time a serial order to which to maintain equivalence, the system is able to achieve strongly consistent replication while only replicating batches of transactional inputs. And since execution order agreement is effectively moved to much earlier in the transaction execution pipeline, Calvin can eschew two-phase commit protocols for transactions that access data spanning multiple partitions, vastly reducing contention footprints of such transactions.

The checkpointing schemes discussed in this paper are implemented in the Calvin codebase and make use of Calvin’s deterministic execution guarantee to capture virtual points of consistency.

III. CHECKPOINTING AND RECOVERY

In this section we present Checkpointing Asynchronously using Logical Consistency (CALC), a novel method for capturing a snapshot in an arbitrary OLTP that achieves an extremely modest reduction in throughput and overall latency.

A. CALC

The following assumptions are based off of the guarantees of virtual points of consistency as described in II-A. In such a system it can be guaranteed that at some point in time $t$, for a given set of transactions $T$, there is an an arbitrary point in time $p$ and an identifiable set $X$ that obeys $(\text{Timestamp}(x) < p \rightarrow x \in X) \land (x \in X \rightarrow \text{Status}(x) == COMPLETED)$. The resultant set $X$ is the set of transactions which are marked by the asynchronous checkpointer as ready to be written to disk.

Although the storage layer is flexible in our scheme, throughout the course of this discussion we will assume the use of a linearly probed hash map composed of pointers to linked lists. The linked list represents a particular set of values for a key in the database. It is important to note that although our data layer contains multiple versions, it is not a true multi-versioned database in that we do not keep track of the value for a specific key as seen by any possible transaction.

Our algorithm proceeds as follows. As seen in figure 1a, we initially maintain only one value for any specified key. We will label this value “stable”. As transactions proceed in the database, we will continue to overwrite the stable value, leaving at most one version per key in the datastore. A checkpoint is prepared for by signaling to the storage layer that there exists a transaction $T_1$ with respect to which the scheduler wishes to capture the snapshot. Figure 1c shows that the database is apathetic with regards to which component of the database is responsible for sending this signal. In practice, this will be sent by the transaction scheduler. At this point in time, it is required that $T_1$ and any transaction following it in the global serial order not have applied any write operations to the data layer.

The storage layer then makes note of $T_1$, without requiring any physical point of consistency. As transactions continue to be processed at the data layer, all read and write operations
are now handled differently. Any reads and writes requested by transactions executing before $T_1$ will continue operating against the stable value. However, $T_1$ and all transactions following it will apply reads and writes against a new “unstable” value. This “unstable” value will be chained at the head of the linked list (Figure 1d), as from then on it is the most likely to be accessed. It is important to note that throughout the entire duration of the checkpoint period, transactions perform read and write operations against the unstable value. However, in the case where a read occurs there is no unstable value, the stable value is returned.

At some physical point in time $P_1$, the scheduler must be able to guarantee that $T_1$ and all transactions that occurred before it in the global serial order have committed or stablized values. In order to do this, the process requests a new background process to begin asynchronously capturing the unstable values. Figure 1f shows this, after which the background checkpointer writes the key and the value returned from the stable query out to a non-volatile medium. Once all keys have been queried and their stable values flushed to disk, the background layer returns the value of the query in the stable version. If no stable version exists, then a null value is returned. After a value has been returned from the data layer, the checkpointing thread prunes the stable value from the in-memory hash map if there exists an unstable version to remain in the storage layer. If no unstable version exists or the stable version does not exist, no pruning occurs. Figure 1f shows this, after which the background checkpointer writes the key and the value returned from the stable query out to a non-volatile medium. Once all keys have been queried and their stable values flushed to disk, the background process can successfully close the checkpoint, and notify the scheduler of its completion. As shown in figure 1g, after this all values in the database are considered stable, and when reads or writes occur against the database from now until the next preparation point, stable values that were not pruned because there was no unstable value, but that later had an unstable write performed can be pruned ad hoc.

B. Partial checkpoints

In a heavily loaded system, all work done outside of performing actual transaction execution reduces the system’s maximum possible transactional throughput and may increase average transaction latency as well. It is therefore desirable to minimize total CPU utilization required to record each checkpoint. To accomplish this, we draw on inspiration from Cao et al.’s scheme of storing a dirty bit directly at the storage level that indicates whether or not the value at the specified key had been updated since the last checkpoint had been captured [4]. In their algorithm, values that have not been updated since the last checkpoint are copied from the previous checkpoint rather than read from storage. This has the advantage of reducing the number of direct reads from storage, as well as spreading work across more background threads to further parallelize the checkpointing process.

To further reduce overhead, we extend this idea to implement partial checkpointing (pCALC). Like Ping-Pong, pCALC tracks the set of keys that have been updated since the last checkpoint (using a hash map, a bloom filter, or a dirty bit placed directly at the storage layer). Unlike in Ping-Pong, however, if a record has not been updated, the key is skipped entirely and does not appear in the current checkpoint.

This additionally reduces the total resources spent by the CPU and disk logger in performing costly IO. Partial checkpointing increases the work that must be done during recovery (multiple partial checkpoints must be merged with the most recent full checkpoint); however merging is fairly lightweight — it can be done by overwriting values from oldest checkpoint to most recent.

Since recovery is a considerably rarer event than checkpoint capture, a modest increase in time to recovery after a crash is often a worthwhile expense in order to reduce runtime checkpointing overhead. Note also that the increase in recovery cost for the partial checkpoints can be limited by a background collapsing process as described in the next section.

C. Background collapsing of partial checkpoints

To reduce the amount of time it takes to bring a failed node back online, we periodically collapse pCALC’s partial snapshots in a background thread. This is generally done in a low-priority thread to take advantage of moments of sub-peak load and refrain from limiting peak throughput.

The collapsing process itself is a simple merge of two or more recent partial checkpoints, where the latest version is always used if a record appears in multiple partial checkpoints. Old checkpoints are discarded only once they have been collapsed. Thus a system failure during the collapsing process or before some recent set of partial snapshots has been collapsed has no effect on durability, since the original partial checkpoints can be merged on the fly at recovery time (though at some additional cost in time-to-recovery).

D. Recovery

Using the system checkpoint, recovery of a node is straightforward. When the failed node comes back online, it restores its state using the most recently captured full checkpoint, merges this with all partial checkpoints taken thereafter, and then uses the transactional input logs to replay all transactions and modify the state of the node until it has “caught up”. In systems where deterministic replay based on transactional inputs alone is not supported, transactions cannot be reported as durably committed to users until reflected in at least a partial checkpoint at at least one system replica.

This technique for recovering a node, as originally proposed in [4] is difficult to compare to a traditional checkpointing scheme. ARIES, the “gold standard” for checkpointing in traditional OLTP systems, has an extremely long
recovery process that involves reconstructing in-memory data structures, replaying physical data manipulation and performing logical UNDO for in-progress transactions. Our scheme obviates the need to perform any disk IO during the recovery process (except the initial load of the checkpoint).

IV. EXPERIMENTAL SETUP

This section describes our implementation of CALC, as well as approaches undertaken to provide an accurate comparison between CALC and other checkpointing algorithms currently in use.

The CALC algorithm was implemented as a component of Calvin, a deterministic database prototype in development at Yale. As discussed in Section II-B, the deterministic transaction scheduling policy was useful during the development of CALC and pCALC, although it could have been implemented in any system able to expose virtual points of consistency.

Our implementation of CALC uses a hash table-based key-value store where stable and unstable versions are chained together in a linked list structure. Figure 2 shows some simple pseudocode representing the general framework used by CALC. As in a multi-versioned store, when a read or write is performed on the data layer, CALC must perform the operation on the appropriate version. However, since CALC keeps at most two values of the each key at any one time, this operation has significantly less additional overhead than full multi-versioning as it is typically implemented in MVCC-based systems.

A. Benchmarking CALC

In order to compare the performance of CALC under different transactional throughputs and contention rates, we implemented three different versions of our checkpointing scheme. The first, labeled “CALC”, is the rudimentary version of our algorithm. The second, labeled “pCALC (bloom)”, is a version of our algorithm that captures partial checkpoints using a doubly-hashed Bloom filter. The third, labeled “pCALC (hash)”, stores keys’ dirty bits in a hash table.

B. Naïve Snapshot

As a point of reference, we implemented a simple version of “naïve snapshot” in our database. A naïvely taken snapshot involves acquiring an exclusive lock on the entire database and iterating through every existing key, writing its corresponding value to disk. Recent work by Lau et al. favors “round-robin” naïve snapshot as a low-cost way of achieving durability in highly-replicated systems [15].

C. Ping-Pong

We also implemented the “Interleaved Ping-Pong” (IPP) checkpointing algorithm [4]. IPP is an asynchronous method that accomplishes the capture of entire snapshots of a database without key locking by triplicating application data and relying on physical points of consistency. In this scheme, the storage layer maintains an application state composed of a simple byte array, and two additional byte arrays of size equal to the application state labeled “odd” and “even”. In addition to a byte representing application data, odd and even maintain a single dirty bit for each element in the array.

Data is initially stored in the application state and even arrays. In the latter, every key’s bit is marked dirty. In addition, odd is marked as the “current” array. The application then proceeds to execute, during which time writes are performed not only on the application state array, but also on the array pointed to by current. Any key that is updated has the corresponding dirty bit turned to “on” in the current array. At a physically consistent point, even becomes the current array, marking the switch into the first checkpoint period. During this period, a background process asynchronously writes to disk all the values that have been labeled dirty during the first period. After each element is written to disk or read from the first checkpoint, the corresponding dirty bit is set to “off”. Once completed, the process alerts the scheduler that at the next physical point of consistency the process can begin again.
IPP makes several assumptions about the application and data layers that need to be addressed. First, it assumes that the application layer has some physical point of consistency at which the database is in an entirely quiesced state. This means blocking any incoming transactions from starting until after all existing operations finish.

Furthermore, IPP assumes that the storage layer uses simple array storage using fixed-length values. We implemented IPP both using this array-based storage and also using the same hash-table-based storage engine used for CALC, which we also feel is more representative of typical database systems’ storage backends.

We further modified the hash-table-based IPP implementation to take only partial snapshots, just as pCALC does. We chose to do this because IPP relies on the order of its integer keys to quickly construct full snapshots, and sorting the required entries in our (unordered, hash table-based) storage layer proved prohibitively expensive.

V. RESULTS

In this section we present our findings when comparing CALC and pCALC against naïve snapshot and both IPP implementations. Our method begins a checkpoint period every thirty seconds, but never attempts to spawn two checkpointing processes at once. For those asynchronous methods that collapse partial snapshots in the background, such a background process is spawned every 100 seconds.

To test our results, we ran several benchmarking applications, including TPC-C and the Microbenchmark used in [18], which is based on the TPC-C New Order transaction, but with several tunable parameters, such as total data set size, contention between transactions, key size, and record size.

We configured the Microbenchmark to initialize the database with a set of bytestring-type keys (a la BigTable’s SSKeys) mapping to 1KB values. Each transaction retrieved five records from the database and updated a subset of the bytes in the record.

We also implemented an augmented Microbenchmark that includes a small number (fewer than 0.01% of total transactions) of “long-running” batch-inserts which take approximately two seconds to complete (several orders of magnitude larger than other transactions) but that do not contribute to overall lock contention.

All experimental results shown here were obtained on a commodity 8-core Intel machine with 12GB of physical memory.

A. TPC-C throughput

We first examined our checkpointing schemes in the context of a system running the TPC-C benchmark at maximum load. We achieved that by submitting a new transaction request to the system for each one completed, so that at any given time at least as many transactions are in the system as could possibly be executing concurrently.

Figure 3 shows TPC-C throughput under each checkpointing scheme—both instantaneous throughput over time and total throughput when averaged over several checkpointing periods. As one might expect, when no checkpointing is done, throughput remains constant—in the case of TPC-C, around 12,500 transactions per second.

For the other checkpointing schemes, we can see that both in the first second of starting the checkpoint ($t = 4$) and...
when the checkpoint is completing \((t = 24)\) transactional throughput drops briefly to below 9000, but hovers around 11000 for most of the duration of the checkpoint period. The major dip in throughput at the start is due to the overhead of analyzing dirty bits, and the dip at the end occurs because all remaining open write buffers are forced to disk when filestreams are closed. However, both of these drops are for such short periods of time that the amortized cost is small—all four asynchronous schemes achieve throughput within 3.8% of the system when no checkpointing is enabled.

Further, because of the low write locality, high-insert workload of TPC-C, most records are touched between each two consecutive checkpoints, so partial checkpoints are nearly as large as full checkpoints. We therefore see little difference between CALC and pCALC.

That IPP achieves significantly (13%) higher throughput when implemented using arrays appears to be due to two factors—reduced overhead of analyzing dirty bits and reduced key size (each key is an integer rather than a string).

B. TPC-C latency

It is also crucial to observe the effect that checkpointing algorithms have on transaction latencies at various points in the checkpointing process.

Because maximally loaded systems often have variable latency due to queuing, contention, thread-scheduling, network backoff, and other factors, we measured TPC-C latency while running our system under a moderate load: approximately 75% of the maximum throughput, or about 9,000 transaction requests per second. We don’t include a throughput graph for this experiment simply because all asynchronous methods are able to keep up with transaction requests, while naïve snapshot again drops its throughput to zero during checkpointing.

Figure 4 shows transaction latency over time. As one would expect, the naïve checkpointer achieves terrible latency (note that latency is measured on a logarithmic scale) for those transactions that were queued before and during the checkpoint but didn’t get to execute until afterward.

The data is similar for all six asynchronous methods, which achieve near-zero latency in all but one scenario. Just before the checkpoint begins to be captured, we notice a latency spike in both versions of Ping-Pong, regardless of implementation. This can be attributed to the fact that Ping-Pong requires a physical point of consistency, forcing the database to be quiesced while it is preparing for the next checkpoint. TPC/C New-Order transactions, which can take up to 50ms due to their insert-heavy workload, must all finish before a physical point of consistency can be guaranteed, spiking latency for successive transactions. We elaborate on this point further in section V-D.

C. Physical memory usage

Figure V-A demonstrates the database size in megabytes as a function of time under our Microbenchmark workload. The reason we use Microbenchmark to analyze memory usage is that there are no insert operations performed in Microbenchmark. Any differences therefore represent the results of multiple versions kept for the sake of checkpointing. No checkpointing at all and naïve snapshot result in only one copy of the database being maintained, a total of 512MB, constant over time. IPP requires keeping all records in triplicate and storing “dirty bits” alongside each value, for a total constant database size of 1536.2MB.

These two constant sizes contrast drastically with CALC and pCALC, whose memory usage grows only when needed for checkpointing. At the beginning of the checkpoint period \((t = 4)\), CALC begins forcing new writes to the “unstable” value of the record, increasing the size of the database. Because the checkpoint does not start for a few seconds after that due to the background process needing to gather the data layer’s keyset, the size of the database quickly grows. However, as soon as the asynchronous checkpointer begins pruning stable values and writing them to disk, the rate of growth of the data layer slows. When unstable values in storage begin to be overwritten—as opposed to being prepended in front of stable values—the size of the database actually begins to shrink from its local maximum of about
850MB. After several seconds the checkpoint period ends, at which point all writes to the previously unstable value, now “stable” in between checkpoint periods, force extraneous unpruned values to be pruned.

The size of the data layer does not return to 512MB immediately after the checkpoint period ends, because CALC uses a “safe” method for pruning, avoiding pruning stable values that have no unstable value prepended ahead of it. In this case, if the asynchronous writer writes the stable value to disk before there exists an unstable value, but an unstable value is prepended before the checkpoint period ends, there will be unnecessary redundancy. By allowing pruning in between checkpoint periods, unstable values can be cleared out in favor of a new stable value. With this technique in place, the size of the database returns to 512MB seconds soon after the checkpoint completes.

pCALC requires only slightly more memory usage to maintain a hash table or bloom filter, on the order of less than 250KB. Therefore, the footprint of that extra storage cannot be seen in the graph.

D. Long-running transactions

One of the major pitfalls of existing techniques for asynchronous checkpointing is the requirement of a physical point of consistency, in which the database is entirely quiesced for a short period of time. However, in many Web and OLTP workloads, occasionally transactions can last a long period of time, creating a prolonged duration where no transactions can execute while the checkpointer attempts to freeze the database.

Figure 6 shows exactly this phenomenon using Microbenchmark with long-running transactions. Although there are very few long-running transactions—at most one running at any given time—these transactions are nonetheless enough to cause unacceptable latency spikes with schemes that require quiescing.

These long-running transactions also cause variations in latency in CALC an pCALC. However, these latencies do not prevent the concurrent execution of unblocked transactions and are amortized over the latencies of all transactions executing at that time.
As shown in figure 7, throughput is also adversely affected by the requirement that the database be quiesced. For the first second of the checkpointing process, starting at \( t = 4 \), the throughput for IPP using array-based storage drops to about 18,000 transactions per second—69.5% lower than capable in the system’s control case. IPP using the default storage engine performs even worse, dropping to 3,800 transactions per second—less than 7% of total transactional throughput—for an entire second.

Note that these throughput reductions do not appear under the CALC and pCALC schemes, which use virtual points of consistency as opposed to physical ones.

E. High write-locality

We also examined the total performance of the schemes under varying write-locality conditions. High write locality means that most transactions update the same small “hot” subset of total data, so that the total number of records modified between two consecutive checkpoints is reasonably small. Applications are extremely common that have large data sets of which small subsets are hot.

In order to clearly show the performance difference between checkpointing schemes under varying write localities, Figure 8 plots the total time it takes to complete a checkpoint. Although this is not a precise measure of overhead cost, it represents a fair approximation of overhead, since total maximum throughput is typically reduced by 5-15% during checkpointing.

As one might expect, the schemes that capture partial snapshots—pCALC and our IPP implementation—complete the checkpointing process extremely quickly when write locality is high and only a small subset of the dataset is updated between checkpoints, while those that don’t take roughly the same amount of time to complete regardless of write locality.

As a small dip in throughput occurs around time \( t = 100 \) associated with the initialization of the collapsing thread. There is also a sharp dip that all three partial snapshotting systems experience at time \( t = 125 \), which corresponds to the completion of the process, flushing of disk buffers, and ending the background process.

G. Recovering a Crashed Node

Since all the checkpointing methods implemented herein capture a set of \( (key, value) \)pairs as they appear at a certain time in the global transaction order, we expect recovery to be approximately the same for each of them. In order to test this, we intentionally crashed a node and brought it back online using a recently captured 800MB checkpoint, then

Although our deployment of Calvin does not use very much disk bandwidth for anything besides checkpointing\(^1\), it is very possible to imagine scenarios in which disk bandwidth is a scarce resource, such as a multitenancy environment or a database system whose data set is not entirely memory resident. In such a system, the benefits of partial snapshots would be amplified.

F. Collapsing Partial Snapshots

For systems that are not replicated (or that are expected to fail often), it is important to collapse checkpoints frequently to minimize the time it takes to recover following a crash. The fact that we can control the times at which checkpoints are merged together allows us to schedule the collapsing process for non-peak load times.

Figure 9 shows the total throughput over time for a system running Microbenchmark with 50% write-locality that is capturing snapshots at times \( t = 90 \) and \( t = 120 \). Additionally, the pCALC and IPP schemes collapse checkpoints; both begin the collapsing process at time \( t = 100 \), when both are nearly finished capturing their current snapshots.

\(^1\) Transactional input is logged to stable storage on a remote server, and all data resides in main memory.
replayed the most recent 50,000 TPC/C transactions since that checkpoint had been captured.

As predicted, regardless of how the checkpoints were taken, recovery took the same amount of time—21.1 seconds (plus or minus about 1 second)—with the exception of IPP when implemented using arrays.

IPP with array based storage instead of our default hash-based storage engine was able to recover more quickly than the other scenarios. Recovery for array-based IPP averaged 15.8 seconds — 30% faster than the other methods. This is primarily because array-based IPP records full snapshots whose records are sorted the same as they are in storage, so writing values out to the array-based storage engine in that order achieved extremely high cache locality. Furthermore, array-based IPP eschewed the overheads associated with maintaining the hash table’s dynamically expandable structure in our system.

We notice also that a significant amount of the time spent recovering from a checkpoint is spent in replaying missed transactions. When recovering from a full checkpoint, on average 4.3 seconds were spent replaying the 50,000 missed transactions—a total of 21% of the total recovery time.

VI. RELATED WORK

There have been several previous attempts to capture checkpoints asynchronously. Dewitt et al. [7] discuss the implementation of a main memory system that uses checkpoints to prevent massive data loss due to system interruptions and crashes. Their scheme begins writing values to stable storage, marking pages in the buffer pool that have been updated after being checkpointed as belonging to a set \( \Delta Mem \). Once the checkpoint completes, the pages in \( \Delta Mem \) are written to their old location on disk. However, this algorithm not only requires a physical point of consistency to record the time at which a checkpoint begins, it also requires at worst a doubling of disk IO in order to reconcile the inconsistent state created by pages in \( \Delta Mem \). Other methods utilizing “fuzzy dumping”, as it is known, exist, but they require that the entire log since before the checkpoint period began be replayed upon recovery, forcing the system to store large physical logs in order to avoid inconsistent states introduced by logical redo [11]. In addition, having to conservatively estimate where to begin replaying the log increases the length of time it takes to recover a failed node.

Granular tuple locking allowing for incremental checkpoint captures has also been previously investigated by Pu [17]. However, this “on-the-fly” checkpointing does not allow transactions that touch checkpointed and non-checkpointed keys to execute. This causes numerous non-deterministic aborts, possibly causing the throughput to drop dramatically during the checkpoint period. Consider, for example, an application with low write-locality and high contention. In this case, few transactions would typically execute at a time. If part of the “hot set” was checkpointed at the beginning of the checkpoint, and part at the end, throughput would effectively drop to zero transactions per second for as long as the checkpointer was running.

Other asynchronous methods have traditionally relied on the availability of a spare hot server to duplicate transactions and intermittently quiesce the database to capture a full snapshot [20]. However, despite the unrealistic assumption that there exists an available spare server to perform no tasks other than checkpointing, this technique is extremely vulnerable to network delays or failures, which could result in unacceptable increases in transactional latencies.

Most recently, Cao et al. introduced Ping-Pong and Zigzag, asynchronous methods for capturing checkpoints that avoid duplicating disk writes, the need for costly physical logging, and quiescing the database for long periods of time [4]. However, as detailed at length in this paper, their work makes inflexible assumptions about the amount of main memory available and the organization of the data layer.

VII. CONCLUSION

As the cost of memory falls, transaction processing systems whose data sets reside entirely in memory are becoming increasingly common. Checkpointing is a vital component to any such system, promising durability of transactional updates upon failures.

We have presented herein CALC, a novel method for checkpointing that requires at most a duplication of the data layer, requires no physical point of consistency in the application state, makes no assumptions about the organization of the database’s storage layer, and incurs modest overhead. In addition, we have documented optimizations that can capture partial snapshots of the system and defer use of CPU resources to times when the system is not at a peak load in order to merge those partial checkpoints into full snapshots.

Although only demonstrated in the context of Calvin’s deterministic execution model, this protocol can be run on any storage system that can expose virtual points of consistency. We expect that it would be possible to implement these schemes on traditional databases using a limited amount of write buffering during checkpointing periods, although particular implementations of these schemes are left for future work.
VIII. ACKNOWLEDGEMENTS

Thanks are due to my co-authors on the first paper our research group published in the fall: Kun Ren, Philip Shao, and Shu-Chun Weng. In addition, thanks to Alexander Thomson for his tireless dedication to the Calvin and with several helpful copy edits for this work. Finally, for a year of unwavering support in pursuing this research, I owe many thanks to my advisor, Professor Daniel Abadi.

REFERENCES


