Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems

Kun Ren, Thaddeus Diamond, Daniel J. Abadi, Alexander Thomson
Yale University
kun.ren@yale.edu; thaddeus.diamond@aya.yale.edu; {dna,thomson}@cs.yale.edu

ABSTRACT

As it becomes increasingly common for transaction processing systems to operate on datasets that fit within the main memory of a single machine or a cluster of commodity machines, traditional mechanisms for guaranteeing transaction durability—which typically involve synchronous log flushes—incur increasingly unappealing costs to otherwise lightweight transactions. Many applications have turned to periodically checkpointing full database state. However, existing checkpointing methods—even those which avoid freezing the storage layer—often come with significant costs to operation throughput, end-to-end latency, and total memory usage.

This paper presents Checkpointing Asynchronously using Logical Consistency (CALC), a lightweight, asynchronous technique for capturing database snapshots that does not require a physical point of consistency to create a checkpoint, and avoids conspicuous latency spikes incurred by other database snapshotting schemes. Our experiments show that CALC can capture frequent checkpoints across a variety of transactional workloads with extremely small cost to transactional throughput and low additional memory usage compared to other state-of-the-art checkpointing systems.

1. INTRODUCTION

As the size of main memory on modern servers continues to expand, an increasingly large number of transactional, Web, and ERP applications can fit their datasets entirely in the physical memory of an individual server or across the main memories of a cluster of servers using a shared-nothing architecture. This has led to the release of a flurry of new main memory database products, including Microsoft Hekaton [4], SAP HANA [10], VoltDB [23], Hyper [7], several main memory NoSQL products, and even a refresh of Oracle TimesTen [8].

A fundamental issue that all main memory database systems face is how to guarantee the 'D'—durability—of ACID when all data sits in volatile main memory. Historically, main memory databases have done this by retaining the logging component of traditional database systems, and forcing log records to stable storage before committing each transaction. Unfortunately, this solution removes several of the advantages of having the data in memory—transactional latency is still bounded by the time it takes to force all log records generated by a transaction to stable storage, and the CPU and memory bandwidth required to generate the log records take a significant toll on the otherwise extremely low overhead main memory transactions. One study indicated that 30% of the CPU cycles of a transaction are devoted to generating log records [6].

Furthermore, as high availability becomes increasingly crucial, the newer generation of database systems tend to build replication into the system from the beginning in order to ensure continued operation of the system in the face of failures. This requirement is reinforced by the fact that another trend is to deploy databases on failure-prone commodity hardware or on virtual machines in the cloud. Once data is replicated, recovering from a log upon a failure is typically far slower than copying database state from a replica [9].

Some main-memory systems, such as H-Store [20] and the early versions of VoltDB [23], and even some disk-based systems like C-Store [19] and HARBOR [9] abandoned logging protocols such as ARIES[14] altogether and rely on "K-safety" mechanisms for guaranteeing durability. As long as the effect of a transaction has made it to K+1 replicas (even if it is only in main memory in all these replicas), the system can survive K failures and still ensure full durability of transactions.

Unfortunately, in many cases, failures are not independent events. Rather, they can highly correlated. Catastrophic failures such as cluster-wide power outages or deterministic bugs in the database code can cause more the K simultaneous failures, which can lead to disastrous losses if none of the data is on stable storage. Indeed, VoltDB moved away from K-safety-based durability and instead released a "command logging" feature [13]. This feature, which is based on deterministic database system research [21], causes the system to log all transactional input. After a failure, the recovery process replays all input transactions that occurred after the latest full checkpoint, which recreates the in-memory database state that existed before the failure. The advantage of such a protocol is that logging of transactional input is generally far lighter weight than full ARIES logging.

It is clear that to avoid traditional logging, modern main memory database systems must implement some sort of checkpointing or backup algorithm in order to avoid catastrophic data loss. For some applications, especially those running

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915966

on NoSQL database systems, checkpointing alone is sufficient: even though transactions that were committed after the checkpoint may be discarded, the application does not require full durability. For other applications, checkpointing combined with k-safety is sufficient—data will only be lost in the case of K+1 correlated failures, and even in this case, the checkpoint limits the amount of data that is lost. For yet other applications, checkpointing combined with a deterministic replay of post-checkpoint transactional input is required to ensure no loss of data at all.

For all three types of applications mentioned above, the database system must be able to checkpoint its state with extremely low overhead in order to receive the benefits of main memory execution without having to log transactional actions to stable storage.

The lack of a log makes database checkpointing much more challenging, as the majority of the high performance checkpointing literature in database systems take the approach of creating "fuzzy" or non-transactionally-consistent checkpoints, and rely on the log along with key data structures found in these checkpoints to recreate a consistent state of the database. Without the log, these fuzzy schemes cannot be used. Instead, traditional techniques such as quiescing the database system (as done in Oracle TimesTen), obtaining read locks on the entire database, or requiring a hot spare server to take the checkpoint can be used. However, the performance of these techniques is far from ideal.

Recent work by Cao et al. [2] presented some promising results for taking low overhead complete checkpoints of database state without requiring a database log. This work focused on applications that have frequent points of consistency, such as massive multiplayer online games that have distinct steps in game states. In particular, Cao et al. presented the Zigzag and Ping-Pong algorithms, which allow an asynchronous thread to capture a snapshot of the complete database state after a point of consistency while simultaneously allowing a separate mutator thread to make concurrent modifications to a different copy of the database state.

Unfortunately, most transactional applications do not guarantee frequent physical points of consistency. Furthermore, the implementation presented by Cao. et al. assumes that the database state is stored in fixed-width vectors, which is not flexible enough for relational tables that have variable length attributes. In addition, it requires multiple copies of the application state at all times.

In this paper, we present CALC, a checkpointing algorithm with significantly lower overhead than the state of the art, and which does not require physical points of consistency. In particular, our algorithms use "virtual" points of consistency. The basic idea is to declare a virtual point of consistency to exist at a particular place in the sequence of transaction commits. All state changes to the database by transactions that commit before this point are considered to be part of a checkpoint, while subsequent transactions' mutations are not reflected in the checkpoint. In order to achieve this, the first state change to each data item by a transaction after this point is redirected to a separate copy of the data item (so that the last value of the item before the checkpoint will not be overwritten, and can be captured by an asynchronous thread which reads this final value).

In addition to our algorithm for taking a low-overhead checkpoint at a virtual point of consistency, we are also able to further reduce the overhead of the checkpointing process by only taking partial checkpoints containing database state which may have changed since the last checkpoint. These partial checkpoints are merged together by a background thread, in order to reduce the checkpoint reconstruction cost at recovery time.

2. ASYNCHRONOUS CHECKPOINTING

When we refer to *checkpoints*, we denote a set of records, each uniquely identified by a primary key, that are written out to stable storage. The set of all of these records represents the full state of the database, also referred to as *application state*, at a given point in time, t.

It is desirable for checkpoint capture algorithms to have the following properties:

1. The process should not significantly impede the system's transactional throughput (even if the system is running at maximum capacity).

2. The process should not introduce unacceptable latencies in transaction or query execution.

3. Although the price of memory is rapidly dropping, it is still relatively expensive to store many OLTP applications entirely in main memory, and main memory becomes a limited resource. Therefore, complete multi-versioning (where database updates do not overwrite previous values, but rather add another version to a new place in memory) is likely to be too expensive in terms of memory resources for many applications. Ideally, the checkpointing process should require minimum additional memory usage.

In order to achieve the first two of these properties, it is desirable for checkpointing to occur asynchronously with respect to other transaction processing tasks. Since records continue to be updated during the process of logging the set of all records to disk, it is therefore important to consider methods of ensuring that a snapshot of the database remains available to the background thread performing the checkpoint. However, in order to achieve the third of these properties, this snapshot should be as small as possible, since complete database multi-versioning or application state replication in memory is not possible for many applications.

2.1 Physical versus virtual points of consistency

A database is defined as being at a *point of consistency* if its state reflects all changes made by all committed transactions, and no change made by any uncommitted, in-progress transaction. A checkpoint taken at a particular *point of consistency* is generally called "transaction-consistent (TC)".

Most database systems achieve asynchronous checkpointing by not taking transaction-consistent checkpoints. They rely on mechanisms such as fuzzy checkpointing that include snapshots of key data structures of database system operation so that it is possible to recreate a transaction-consistent state of the database with the help of the database log during recovery. However, without the help of the log, checkpointing algorithms that do not produce transaction-consistent checkpoints are unable to recreate a consistent state of the database.

Certain applications, such as many massive multiplayer online games, naturally reach points in the process of normal execution in which no transactions are actively executing. This is termed a *physical* point of consistency. Most applications, however, are not guaranteed to reach physical points of consistency at regular intervals. Although it is always possible to force the database system to reach such a point by quiescing it entirely (disallowing new transactions from beginning to execute until all currently active transactions complete), quiescing a database system can have significant detrimental effects on both throughput and latency. Therefore, one of the primary goals of the work discussed in this paper is to achieve fast checkpointing even in the absence of physical points of consistency.

We introduce the notion of a *virtual point of consistency*. A virtual point of consistency is a view of the database that reflects all modifications committed before a specified time and none after—but obtained without quiescing the system. Virtual points of consistency are instead created using full or partial multi-versioning.

Systems implementing snapshot isolation via MVCC implement *full multi-versioning*. In such schemes, a full view of database state can be obtained for any recent timestamp simply by selecting the latest versions of each record whose timestamp precedes the chosen timestamp. Since MVCC is specifically designed such that writes never block on reads, a virtual point of consistency can be obtained inexpensively for any timestamp.

However, as described above, many main memory database systems do not implement full multi-versioning since memory is an important and limited resource in these systems. Therefore, precise partial multi-versioning is preferable.

The general idea of precise partial multi-versioning is the following: while a checkpoint is being captured, track two versions of each record: (a) the *live* version, which represents the most current state of the record, and (b) a *stable* version, which represents the state of the record as of the virtual point of consistency. Any transaction that commits before the point of consistency must update both versions of each record it writes, while transactions that commit after the point of consistency only overwrites the live version. However, a mechanism is needed for ensuring that each transaction updates the correct version(s) of the record.

In the Zigzag and Ping-Pong algorithms presented by Cao et al., a global flag specifies whether the point of consistency has been reached yet. The flag (and certain other corresponding metadata) are only updated at a physical point of consistency—therefore any update operation is guaranteed to precede the point of consistency if and only if the transaction's commit will also occur before the point of consistency. Zigzag and Ping-Pong pre-allocate memory for two or three (respectively) copies of each record to use as live and stable versions of that record.

2.2 The CALC Algorithm

We present here the CALC (Checkpointing Asynchronously using Logical Consistency) algorithm, which is designed to have the advantages of asynchronous checkpointing but with four additional benefits:

- **No requirement for a database log.** CALC is able to create a transaction-consistent checkpoint without relying on any kind of database log except for a simple log containing the order in which transactions commit.
- **Uses virtual points of consistency.** CALC does not require quiescing of the database in order to achieve a physical point of consistency. It therefore is able to avoid throughput drops and latency spikes for transactions that are submitted during and after checkpoints, which is critical for maintaining database service level agreements.
- **Reduced memory usage.** At most two copies of each record are stored at any time. Memory usage is minimized by only storing one physical copy of a record when its live and stable versions are equal or when no checkpoint is actively being recorded.
- **Low overhead.** CALC's overhead is smaller than other asynchronous checkpointing algorithms.

CALC implements a storage structure where each record key is associated with two record versions—one live and one stable. Initially, the stable version is `empty`, indicating that the stable version is not actually different from the live version, and that a checkpointing thread may safely record the live version. CALC also needs to maintain a bit vector called *stable_status*. Each record corresponds to a bit in *stable_status* to indicate whether its stable version is empty[1]. For the purposes of this discussion, we assume that transactions are executed using a pessimistic (lock-based) concurrency control algorithm in a multi-threaded execution environment, since this is the most common concurrency control implementation. We also assume that there exists a `commit-log`, and each transaction commits by atomically appending a commit token to this log before releasing any of its locks.

A system running the CALC algorithm cycles through five states, or phases:

1. the *rest* phase, in which no checkpoint is being taken,
2. the *prepare* phase, immediately preceding a virtual point of consistency,
3. the *resolve* phase, immediately following a virtual point of consistency but before the asynchronous checkpoint capture has started,
4. the *capture* phase, during which a background thread records the checkpoint to disk, and meanwhile deletes stable versions,
5. the *complete* phase, which immediately follows the completion of *capture* phase.

Each transition between phases of the algorithm is marked by a token atomically appended to the transaction `commit-log`. Therefore it can always be unambiguously determined which phase the system was in when a particular transaction committed. Additionally, each transaction makes note of the phase during which it *begins* executing. The algorithm proceeds as follows.

2.2.1 Rest phase

While in the *rest* phase, every record stores *only* a live version. All stable versions are *empty*, and the bits in *stable_status* vector are always equal to *not_available*. Any transaction that begins during the *rest* phase uses only the live version for its read and write operations, regardless of when it commits.

[1]Inserts and deletes are handled via two additional bit vectors, called *add_status* and *delete_status*. However, we only focus on updates in our discussion to keep the explanation simple.

```
Initialized Database status:
    bit not_available = 0;
    bit available = 1;
    bit stable_status[DB_SIZE];
    foreach key in Database
        db[key].live contains actual record value;
        db[key].stable is empty;
        stable_status[key] = not_available;

function ApplyWrite(txn, key, value)
    if (txn.start−phase is PREPARE)
        if (stable_status[key] == not_available)
            db[key].stable = db[key].live;
    else if (txn.start−phase is RESOLVE OR CAPTURE)
        if (stable_status[key] == not_available)
            db[key].stable = db[key].live;
            stable_status[key] = available;
    else if (txn.start−phase is COMPLETE OR REST)
        if (db[key].stable is not empty)
            Erase db[key].stable;
    db[key].live = value

function Execute(txn)
    txn.start−phase = current−phase;
    request txn's locks;
    run txn logic, using ApplyWrite for updates;
    append txn commit token to commit−log;
    if (txn.start−phase is PREPARE)
        if (txn committed during PREPARE phase)
            foreach key in txn
                Erase db[key].stable;
        else if (txn committed during RESOLVE phase)
            foreach key in txn
                stable_status[key] = available;
    release txn's locks;

function RunCheckpointer()
    while (true)
        SetPhase(REST);
        wait for signal to start checkpointing;
        SetPhase(PREPARE);
        wait for all active txns to have
            start−phase == PREPARE;
        SetPhase(RESOLVE);
        wait for all active txns to have
            start−phase == RESOLVE ;
        SetPhase(CAPTURE);
        foreach key in db
            if (stable_status[key] == available)
                write db[key].stable to Checkpoint;
                Erase db[key].stable;
            else if (stable_status[key] == not_available)
                stable_status[key] = available;
                val = db[key].live;
                if (db[key].stable is not empty)
                    write db[key].stable to Checkpointing;
                    Erase db[key].stable;
                else if (db[key].stable is empty)
                    write val to Checkpointing;
        SetPhase(COMPLETE);
        wait for all active txns to have
            start−phase == COMPLETE;
        SwapAvailableAndNotAvailable ();
```

Figure 1: CALC algorithm pseudocode.

2.2.2 *Prepare phase*

When the checkpointing process is triggered to begin, the system enters the *prepare* phase. Transactions that start during the *prepare* phase read and write *live* record versions, just like transactions that started during the *rest* phase. However, before updating a record, if the stable version is currently empty and unavailable, a copy of the live version is stored in the stable version prior to the update. This is done because the system is not sure in which phase the transaction will be committed. For example, a transaction T_P that begins during the *prepare* phase and writes a record R_1—overwriting a previous value R_0—makes a copy of R_0 in its stable version before replacing R_0 in the live version. Immediately after committing, but before releasing any locks, a check is made: if the system is still in the *prepare* phase at the time the transaction is committed, the stable version is removed.

2.2.3 *Resolve phase*

The *prepare* phase lasts until all active transactions are running in *prepare* phase—that is, until all transactions that started during the *rest* phase have completed. At this point, the system transitions to the *resolve* phase (appending a phase-transition token to the `commit-log`). This transition marks the *virtual point of consistency.* All transactions that have committed before this point will have their writes reflected in the checkpoint, whereas transactions that commit subsequently will not.

If a transaction that started during the *prepare* phase commits in *resolve* phase, the stable version is not deleted, and the corresponding bit in *stable_status* is set to *available.* For example, suppose the transaction T_P (discussed above) committed during the *resolve* phase. When it performed its check to see what phase it committed in, it discovers this, and then sets the corresponding bit in *stable_status.* Now subsequent transactions will see R_1 as the live version, but the background checkpoint recorder will see R_0 as the stable value when it runs.

Transactions that start during the *resolve* phase are already beginning after the point of consistency, so they will certainly complete after the checkpoint's point of consistency. Therefore, they always copy any live version to the corresponding stable record version before updating it, unless the record already has an explicit stable version.

2.2.4 *Capture phase*

The *resolve* phase lasts until all transactions that began during the *prepare* phase have completed and released all their locks.

The system then transitions into the *capture* phase. Transaction write behavior is the same during the *capture* phase as during the *resolve* phase, both for new transactions, and for transactions that were already active when the *capture phase began.*

Once this phase has begun, a background thread is spawned that scans all records, recording stable versions (or live versions for record versions that have no explicit stable version) to disk. As it proceeds, it erases any explicit stable versions that it encounters. This can be accomplished with no additional blocking synchronization. The *capture* phase section of the `RunCheckpointer` function outlined in Figure 1 shows the pseudocode for this process. Note that the bit in *stable_status* is set to *available* after the background thread accesses the record. This prevents other transactions running during the *capture* phase from creating the stable version again.

2.2.5 Complete phase

Once the checkpoint capture has completed, the system transitions into the *complete* phase. Transaction write behavior reverts to being the same as in the *rest* phase. Once all transactions that began during the *capture* phase have completed, the system transitions back into the *rest* phase, and awaits the next signal to take a checkpoint.

However, before transitioning back to the *rest* phase, the function *SwapAvailableAndNotAvailable()* is called to swap the variables *not_available* and *available*, in order to reverse their mapping to '1' and '0' values in the bit vector. In one iteration of the checkpointing algorithm, *not_available* maps to "1" and *available* maps to "0"; in the next *not_available* maps to "0" and *available* maps to "1". This allows the system to avoid a full scan to reset the *stable_status* bits, since after the *capture* phase all the *stable_status* bits are set to *available*, but at the beginning of the *rest* phase we want all the *stable_status* bits to be set to *not_available*.

Figure 1 shows pseudocode for the CALC algorithm. In particular, it outlines the initialized database status and the status of variables that are used in CALC algorithm. It also shows the `ApplyWrite` function called directly by transaction logic, the `Execute` function, which outlines all steps taken by a worker thread executing a transaction, and the basic steps taken by the CALC background thread over the course of a checkpointing cycle (in the `RunCheckpointer` function).

2.3 Partial checkpoints

In the CALC algorithm described above, each capture phase takes a complete checkpoint of the entire database. Even though this can be done in an asynchronous background thread which runs with low overhead, if very few records were updated since the last checkpoint (either because there is transaction skew where the same records get updated repeatedly or because the workload is mostly 'read-only'), it is wasteful to take a complete checkpoint if there are very few differences from the last checkpoint.

Therefore, as an alternative to the CALC algorithm, we also propose a pCALC option, that takes partial checkpoints that contain only records that may have been changed since the most recent checkpoint. These partial checkpoints can be merged together to create a complete checkpoint either at recovery time or as a background process during normal operation.

In order to take partial checkpoints, pCALC keeps track of keys that were updated after the most recent virtual point of consistency. We explored three alternative data structures for keeping track of keys that are updated. In the first approach, we stored each updated key in a hash table. In the second approach, we used a bit vector where the ith bit is set if the ith value had been updated since the most recent checkpoint. In the third approach, we used a bloom filter to decrease the size of the aforementioned bit vector. The main disadvantage of the bit vector approach is that it always consumes one bit per record in the database — even if that record has not been updated. The first approach never wastes space on irrelevant records and the third approach falls somewhere in between these two extremes. Nonetheless, in practice we found that the bit vector approach usually outperformed the other two approaches. This is because the entire database already fits in main memory (since this is the application space for which we are designing CALC). If each record consumes 50 bytes (400 bits), then the bit vector only extends memory requirements by $1/400 = 0.25\%$. Although smaller data structures have better cache properties, we found that the additional work required by the other approaches were slightly more costly than the performance savings from improved cache locality. Therefore, we settled on the bit vector approach in our final implementation.

Bit vectors are atomically cleared during the checkpointing period. This can be accomplished with no blocking synchronization by keeping two copies of the structure, and flipping a bit specifying which is active at the beginning of the *resolve* phase, and clearing the inactive one during each checkpointing period. For transactions that start in the *prepare* phase and commit in the *prepare* phase, we update the bit vector associated with current upcoming partial checkpoint; for those that commit in the *resolve* phase, we update the bit vector associated with the later partial checkpoint. For those transactions that start in (or after) the *resolve* phase, the bit vector associated with the later partial checkpoint is updated.

Although the bit vector must be cleared during each checkpoint period, note that pCALC does not require a full scan of the database. This can lead to a significant reduction in overhead of checkpointing relative to CALC.

2.3.1 Background merging of partial checkpoints

The cost at recovery time of taking partial checkpoints can be ameliorated by periodically collapsing pCALC's partial snapshots in a background thread. This is done in a low-priority thread to take advantage of moments of sub-peak load, while refraining from limiting peak throughput. Furthermore, this task can optionally be offloaded to different machines that do *not* stand in the critical path of a high-volume OLTP application pipeline.

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.

3. RECOVERY

Thus far we have discussed two types of checkpoints: CALC and pCALC. We now discuss how to recover from each type of checkpoint.

3.1 Recovery Using CALC

A CALC checkpoint is a complete snapshot of database state as of a particular point it time. It is guaranteed to be physically consistent — it reflects the state of the database after all transactions that committed prior to the checkpoint have been processed and no intermediate or temporary state from ongoing transactions that had not committed prior to the checkpoint. This makes recovery very easy for the use cases discussed in Section 1: for the use cases that find it acceptable to lose transactions that committed after the most recent checkpoint (e.g. the NoSQL and K-Safety use cases mentioned above), the database state is created via loading in the most recent **completed** checkpoint. For use cases that rely on determinism to avoid losing committed transactions [22, 13, 20, 23, 21], the database state is created via a two-step process. First, the most recent completed checkpoint is loaded to create a temporary database state. Then the commit log is read to see which transactions committed after this checkpoint that was loaded. These transactions are then replayed, leveraging determinism to ensure that the serialization order and states are equivalent to the original order before the crash.

Note that all of CALC's data structures — the "stable" record versions, the stable_status bit vector, etc., get wiped out along with the rest of volatile memory upon a crash. So there is no clean-up that has to occur upon recovery. The most recent completed checkpoint is loaded and all of CALC's data structures are initialized as shown in Figure 1.

3.2 Recovery Using pCALC

Recovery from partial checkpoints is slightly more complicated since there is not necessarily a single checkpoint which can be loaded in order to get a consistent database state.

A naive algorithm would be to collapse all partial checkpoints that exist on stable storage into a single checkpoint using the merge process described in Section 2.3.1. Obviously, although correct, this is unacceptably slow, since partial checkpoints are continuously generated since the beginning of the installation of the database system.

A less naive algorithm is to occasionally take a full checkpoint so that only the partial checkpoints that were taken since this full checkpoint need to be merged. This leads to a recovery time vs. runtime tradeoff. Full checkpoints are more expensive to generate than partial checkpoints, so optimizing for runtime implies that full checkpoints should be taken rarely. However, this increases recovery time since on average there will be more partial checkpoints to merge since the last full checkpoint.

Note that if a full checkpoint is merged with all partial checkpoints that were taken subsequent to it, the result is a new full checkpoint that is accurate as of the most recent partial checkpoint. Therefore, as an alternative to occasionally taking full checkpoints, the system can instead collapse a series of partial checkpoints and a older full checkpoint. This has the advantage of being a process that can be run entirely asynchronously, in the background. This is the approach that we take in our implementation. However, the recovery time vs. runtime tradeoff remains — it is now expressed as the size of the batch of partial checkpoints that are allowed to exist before performing this merge process. We revisit this tradeoff experimentally in Section 5.1.3.

Once a full checkpoint has been created via merging partial checkpoints, recovery proceeds as discussed in Section 3.1 above.

4. EXPERIMENTAL SETUP

To evaluate the CALC algorithm side-by-side with other state-of-the-art checkpointing approaches, we implemented a memory-resident key-value store with full transactional support. Transactions in our system are implemented as C++ stored procedures, and are executed by a pool of worker threads, using a pessimistic concurrency control protocol to ensure serializability. In order to eliminate deadlock as an unpredictable source of variation in our performance measurements, we implemented a deadlock-free variant of strict two-phase locking.

All experimental results shown here were obtained on Amazon EC2 using a c3.4xlarge instance, which has 16 virtual cores (Intel Xeon E5-2680 v2 (Ivy Bridge) Processors) running 64-bit Ubuntu 12.04.2 LTS with 30GB RAM and a 160GB magnetic disk that delivers approximately 100-150 MB/sec for sequential reads and writes. Of the 16 cores, we allocated 15 cores to transaction processing and checkpointing threads, and devote the remaining core to shared database components and infrastructure necessary to take statistics for these experiments.

4.1 Benchmarking CALC

In order to compare the performance of CALC under different transactional throughputs and contention rates, we implemented two different versions of our checkpointing scheme. The first, labeled CALC, is the rudimentary version of our algorithm. The second, labeled pCALC, is a version of our algorithm that captures partial checkpoints.

We compare the performance of CALC and pCALC to four other comparison points, described in the following sections.

4.1.1 Naive Snapshot

We implemented a simple version of "naive snapshot" in our database. A naively taken snapshot involves acquiring an exclusive lock on the entire database, iterating through every existing key, and writing its corresponding value to disk. Recent work by Lau et al. favors "round-robin" naive snapshot as a low-cost way of achieving durability in highly-replicated systems [9].

4.1.2 Fuzzy Checkpointing

The naive snapshot algorithm incurs a long interruption of normal processing during checkpointing. The fuzzy checkpointing algorithm was invented to avoid this, as it only interrupts normal processing to write two relatively small data structures to the log, instead of a full snapshot of database state. As a result, it has become an extremely popular (perhaps even the most popular) checkpointing algorithm used in database systems — especially those systems that use the ARIES recovery algorithm.

Although there exist several proposals and optimizations of fuzzy checkpointing, the simple version of the algorithm is done as follows [1, 18]: (1) The DBMS stops accepting new update, commit, and abort operations. (2) It creates a "checkpoint" log record which contains a list of all the dirty pages in the cache, and also a list of all active transactions along with pointers to the last log record that these active transactions have made. (3) It writes this checkpoint record to the log. (4) The DBMS allows normal operation to resume. At this point, the pages marked as dirty in the checkpoint record can be flushed to disk in parallel with the DBMS running new update, commit and abort operations.

It has repeatedly been pointed out in the literature that the traditional version of the fuzzy checkpointing algorithm is designed for disk resident DBMSs, and does not adapt well to main memory DBMSs [12, 5]. The main issue is that while the granularity of a write to disk is a page, the granularity of a write to main memory is typically a single record. Therefore, the dirty page table in the fuzzy checkpoint (along with the pageLSN header of disk pages) need to be modified to work with record-level granularity. This increases the size of these structures. Furthermore, the higher throughput of main memory database systems tend to result in more dirty records per checkpoint than seen in disk-based systems. Combined these two issues cause much more to be written to a checkpoint record, and therefore a longer interruption of normal database processing. There have therefore been several proposals for reducing the size of this interruption (e.g. splitting the checkpoint record to begin_checkpoint and end_checkpoint records) and to generally improve the performance of checkpointing in main memory database systems. We have implemented that which we believe to be the most performant of these proposals (IPP), which we describe in the next section. However, we nonetheless also run a version of basic fuzzy checkpointing since it can serve as familiar comparison point that is well-known to most database researchers.

Our implementation of fuzzy checkpointing uses bit vectors to keep track of dirty keys (like we do in pCALC — see Section 2.3). Furthermore, in order to directly compare this approach to our implementations of CALC and pCALC, we created full and partial versions of the fuzzy checkpointing. The default implementation is the partial version — pFuzzy — since that corresponds to the traditional approach to fuzzy checkpointing. To create the full version, we maintain an extra copy of the database in main memory which is the latest consistent snapshot, so the full checkpoint can be generated by merging dirty records with the latest snapshot.

4.1.3 *Ping-Pong*

As a third comparison point, we implemented the "Interleaved Ping-Pong" (IPP) checkpointing algorithm [2]. 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 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 finds all the values that have been labeled as dirty during the first period, and merges them with the last consistent checkpoint in order to construct a new consistent checkpoint that can be written to disk. After each element is written to disk, 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.

4.1.4 *Zig-Zag*

As a fourth comparison point, we implemented the Zig-Zag checkpointing algorithm [2]. Zig-Zag starts with two identical versions of each record in the datastore: $AS[Key]_0$ and $AS[Key]_1$, plus two additional bit vectors, $MR[Key]$ and $MW[Key]$. $MR[Key]$ indicates which version should be used when reading a particular record, Key, and $MW[Key]$ indicates which version to overwrite when updating Key. New updates of Key are always written to $AS[Key]_{MW[Key]}$, and $MR[Key]$ is set equal to $MW[key]$ each time Key is updated. The bit vector MR is initialized with zeros and MW with ones, and each checkpoint period begins with setting $MW[Key]$ equal to $\neg MR[Key]$ for all keys in the database. Thus $AS[Key]_{MW[Key]}$ always stores the latest version of the record, and the asynchronous checkpointing thread can therefore safely write $AS[Key]_{\neg MW[Key]}$ to disk.

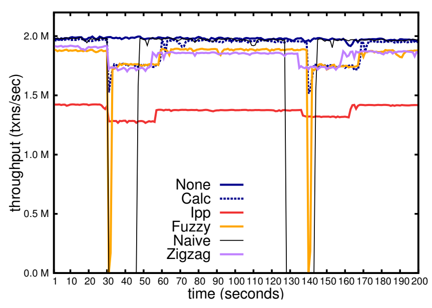
IPP and Zigzag make several assumptions about the application and data layers that do not necessarily apply to all use cases. First, they assume that the application layer has some *physical* point of consistency at which the database does not contain any state that has been written by a transaction that is not yet committed. For applications that do not naturally have such points of consistency, the database can enforce one by blocking any incoming transactions from starting until after all existing operations finish.

Second, IPP and Zigzag assume that the storage layer uses simple array storage using fixed-length values. Although some applications are possible to be implemented over fixed-length array-based storage (including the massive multiplayer online game applications for which IPP and Zigzag were designed), many applications have data that cannot easily be managed via array-based storage (which is why most modern database systems do not use array-storage backends). In order to be general and make an apples-to-apples comparison, we implemented a modified version of IPP and Zigzag that use the same hash-table-based storage engine that is used for CALC. Nonetheless, we maintain the cache optimizations of these algorithms For example, IPP stores all three copies of each record continuously in memory in order to keep as much data as possible for a record in the same cache line. We maintain the same optimization — maintaining all three copies of each record contiguously in the same hash entry of the database.

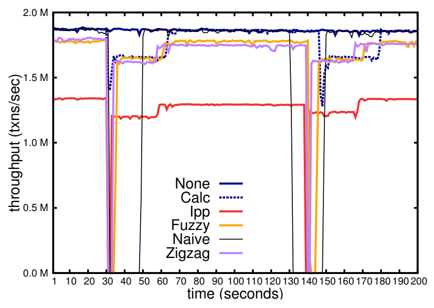
IPP and Zigzag both take full checkpoints and are thus best compared to CALC. However, in order to additionally compare them with pCALC, we implemented a second version of the IPP and Zigzag implementations that take only partial snapshots using the same bit vectors as used for pCALC. In this way, the advantage of pCALC of not having to write entire database snapshots to disk at each checkpoint can also be attained by IPP and Zigzag. These secondary implementations are labeled "pIPP" and "pZigzag" in the experiments.

5. EXPERIMENT RESULTS

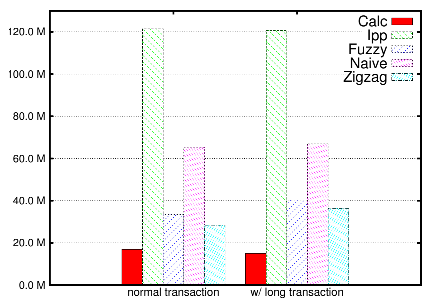
In this section we run experiments to benchmark CALC against the naive snapshot, fuzzy, IPP and Zigzag algorithms described in the previous section. We run our experiments using two benchmarking applications: The first is a microbenchmark similar to the one used in recently published transactional database system papers [22, 16]. The second is TPC-C, which we implemented by writing stored procedures in C++.



(a) Throughput over time, no long xacts



(b) Throughput over time, long xacts



(c) Transactions lost.

Figure 2: Full checkpointing, Microbenchmark experiments.

5.1 Microbenchmark experiments

The microbenchmark operates on a collection of 20 million records, where each record is 100 bytes and has an 8 byte key. We experiment with two versions of the microbenchmark. The first version consists entirely of transactions that read and update 10 records from the database, and do some simple computing operations. The second version contains 99.999% of transactions that are the same type as the first version, but 0.001% of transactions are long-running batch-writes which take approximately two seconds to complete. We keep contention low for both versions of the microbenchmark. This enables worker threads to fully utilize all cores on the experimental system, instead of becoming blocked on acquiring locks. This ensures that the overhead of checkpointing is clearly visible.

5.1.1 Full Checkpointing

We first compare CALC with the other checkpointing algorithms for the "full" checkpoint case, where each scheme must create a complete snapshot of the database before the end of the checkpoint interval. Figure 2(a) shows the system throughput over time for the first version of the microbenchmark (the version without long-running transactions). Figure 2(b) shows the results for the second version of the microbenchmark (the version with .001% long-running transactions). This experiment runs over the course of a 200 second period, where the first checkpoint is taken, starting at time 30 seconds, and the second checkpoint is taken 80 seconds after the first one. We run database system under peak workload (the database system is 100% busy).

For the naive snapshot algorithm, the throughput drops to 0 transactions per second while the checkpoint is being taken. However, the time to take this checkpoint is very small, since all database resources are devoted to creating the checkpoint during this period. After the checkpoint completes, database throughput returns to normal.

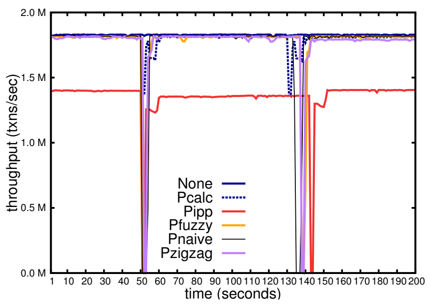
For the fuzzy checkpointing algorithm, the database system is quiesced to write the dirty record table to disk (which results in a sharp drop in database throughput), but then continues to process transactions after the data structure is written. Since the size of the dirty record table is much smaller than the size of a full database checkpoint, the time that the database is quiesced is smaller for the fuzzy case than the naive snapshot case. However, after the database returns to processing transactions, database throughput does not fully return to normal, since some processing resources need to be allocated to asynchronously creating the full checkpoint from the data structures written out in the previous step.

The IPP algorithm's performance (even before the checkpoint period begins) is around 25% worse than the no-checkpoint baseline, since it needs to maintain two copies of the database state at all times, which involves memory copy operations during normal operation. Even though we utilize the IPP optimization of ensuring that both copies of each record are stored contiguously in memory, and thus this copy operation is as cache-local as possible, nonetheless the overhead is noticeable given the write-intensive nature of the workload.

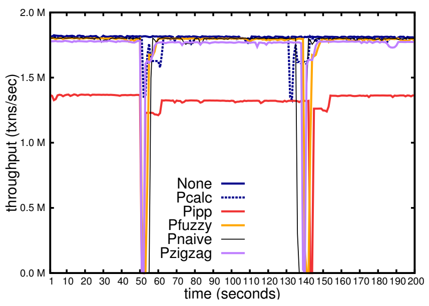
We found that the Zigzag algorithm performed better than IPP. At rest, before the checkpoint period begins, Zigzag results in only a 4% decrease in throughput relative to the baseline. This is because, unlike IPP, Zigzag only has to perform writes once, without any additional copying. The only extra ongoing cost at rest is that it needs to read and update two bit vectors that control access to the two copies of the data. Once the checkpoint period begins, its drop in throughput relative to its baseline is larger than the corresponding drop for IPP. Nonetheless, since its baseline is so much higher than IPP for this workload, its throughput remains higher than IPP throughout the experiment.

The naive algorithm has to quiesce the database system to perform the entire checkpoint, and the fuzzy checkpointing algorithm has to quiesce the system while certain data-structures are written (see above). These needs to quiesce the system are mostly independent from the workload. Therefore, the precipitous drop in throughput during this quiescence period is approximately the same in Figure 2(a) and Figure 2(b) for both the naive and fuzzy algorithms. In contrast, IPP and Zigzag only have to quiesce the system until all active transactions at the start of a checkpoint complete (this enables the system to get into a physical point of consistency which are required by these algorithms). This type of quiescence is very workload dependent. When every active transaction is short (as for the workload in Figure 2(a)), the period of time for which the database must quiesce is essentially invisible. However, where there are long-running transactions in the workload (as for Figure 2(b)), the period of time for which the database has to reject new transactions until these long transactions complete is noticeable. Therefore, there is a visible drop to zero transactions a second at the beginning of the checkpoint period for both IPP and Zigzag in Figure 2(b) but not in Figure 2(a).

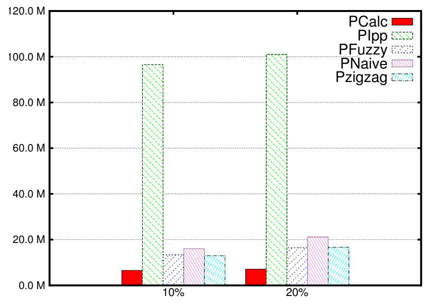
In contrast, CALC does not require physical points of consistency, instead relying on a virtual point of consistency. Therefore, the performance of CALC is similar for both ver-



(a) 10% of records modified since last checkpoint.



(b) 20% of records modified since last checkpoint.



(c) Transactions lost.

Figure 3: Partial checkpointing with Long-running transactions.

sions of the microbenchmark in Figures 2(a) and Figure 2(b). Furthermore, it does not suffer from performance degradation in baseline performance, since unlike IPP and Zigzag, it does not have to maintain multiple copies of the data outside of checkpoint periods — either via making multiple copies of each write, or by controlling access to different copies though bit vectors. It does, however, have a 10% performance degradation relative to its baseline during the asynchronous parts of the checkpoint process. This drop relative to its baseline is slightly larger than the corresponding drops for the other algorithms because in addition to writing the checkpoint, CALC requires some additional memory copy operations during the checkpoint period (in particular, it has to copy the live version to the stable version in the database the first time a record is written after the beginning of the checkpoint period).

In general, the different checkpointing algorithms have different shapes in Figures 2(a) and Figure 2(b). Some have short periods of zero transactions per second being run, while some have longer periods of reduced performance during asynchronous checkpointing processes, and some have both. In general, the total transactions that were completed is the area underneath each line in the graph, and the total cost of checkpointing is thus the difference between these areas. Therefore, in order to present the total cost of checkpointing, we measured, for each algorithm, the total number of transactions that were completed during the 200 second window of our experiment. We then subtracted this number from the total number of transactions that would have completed if no checkpointing code had been performed. The result is the total cost of checkpointing — the number of transactions that were not processed during this period in order to devote database resources to checkpointing.

Figure 2(c) shows the result of this cost summary calculation. It is perhaps more clear from this graph than the previous graphs how much lower overhead CALC has relative to the other algorithms. Even relative to Zigzag for the workload without long transactions (when Zigzag does not have to noticeably quiesce the database), CALC still significantly outperforms Zigzag by almost 40%.

5.1.2 *Partial Checkpointing*

We now compare the partial versions of the checkpointing algorithms. In this case, instead of being required to create a full snapshot of the database at each checkpoint, the algorithms only have to checkpoint the records that had been modified since the most recent checkpoint.

The write locality conditions can make a big difference for the partial checkpointing schemes. High write locality means that most transactions update the same "hot" subset of the records in the database, so that the total number of records modified between two consecutive checkpoints is small. Hence, the size of the partial checkpoints will also be small. It is very common to see this kind of skew in data access patterns in real-world applications.

Therefore we vary the write locality conditions for our partial checkpoint experiments. Figure 3(a), 3(b) shows the performance of the partial checkpointing algorithms where there is 10% and 20% data access skew respectively. The workload we run is the microbenchmark with long-running transactions. Similarly to the previous experiment, the experiment lasts 200 seconds and contains 2 checkpoints.

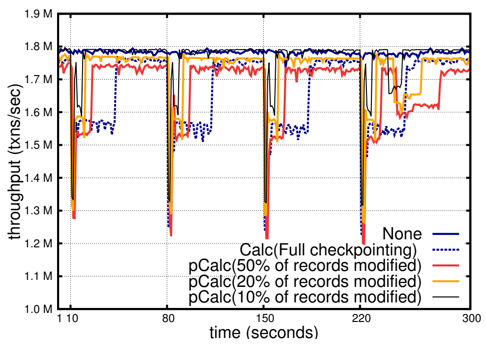
Overall, the relative performance of CALC vs. Zigzag, IPP, fuzzy, and naive are the same for this set of experiments as they were for the previous set of experiments. The only difference is that the amount of time spent writing the checkpoint data is smaller for all 5 algorithms, since only modified records need to be written. For Zigzag, IPP, fuzzy, and CALC, this results in a much shorter window of time that the system is running at 7% to 10% lower than maximum capacity. For naive, this results in a much shorter window of time that the system must be quiesced.

In all five cases, the total cost of checkpointing is proportionally reduced, as shown in Figure 3(c). Interestingly, as data access skew increases, the relative cost of CALC vs. the other algorithms improves, since the time to take a checkpoint gets increasingly small for all four algorithms, and therefore baseline performance and the time required to manufacture a physical point of consistency start to dominate the overall checkpointing cost. Since CALC has almost no overhead at rest and does not require a physical point of consistency, its relative performance to the other algorithms improves.

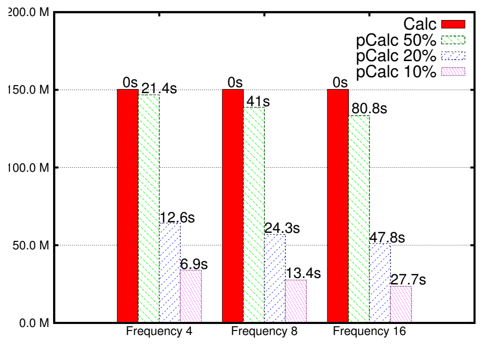
5.1.3 *Recovery Time*

Section 3 discussed the recovery time vs runtime tradeoff for CALC vs. pCALC. In both cases, recovery involves loading a complete checkpoint, and (potentially) replaying committed transactions since this checkpoint. Since both the loading cost and the replaying of committed transactions is a constant for both CALC and pCALC, and their costs are entirely independent of the checkpointing scheme, we ignore them from our recovery time analysis.

Thus, the recovery time for CALC is effectively zero in



(a) Throughput over time.



(b) Transactions lost.

Figure 4: Full checkpointing vs. Partial checkpointing.

our analysis — no additional work has to occur aside from loading the most recent checkpoint and replaying committed transactions. However, if the most recent checkpoint is a partial checkpoint instead of a full checkpoint, additional work must be performed in order to merge all partial checkpoints that have occurred since the most recent full checkpoint. As mentioned in Section 3.2, we can use a background process during runtime to continuously merge together partial checkpoints so that the distance to the most recent full checkpoint is limited. However, there remains a recovery time vs runtime tradeoff — the less frequently this background merge process happens, the less cost there is at runtime, but the longer the recovery time.

To investigate this process further, we experimented with three different configurations for this background process — it runs after 4, 8, and 16 partial checkpoints have been taken.

Figure 4(a) shows the results of this experiment when partial checkpoints are merged with the most recent full checkpoint after every 4 partial checkpoints. The experiment lasts 300 seconds and checkpoints are taken at 10 seconds, 80 seconds, 150 seconds and 220 seconds. For the pCALC algorithm, the four partial checkpoints are merged together with the most recent full checkpoint 5 seconds after the fourth checkpoint completes. We benchmark with three write locality skew cases(10%, 20% and 50% of records modified).

The left 4 bars in Figure 4(b) summarizes the number of transactions lost for this graph, analogously to how Figure 2(c) showed the transactions lost for Figures 2(a) and 2(b). The number of transactions lost is an indication of the runtime cost for each experiment. Above each bar in the graph shows the recovery time needed in the worst case scenario were all four partial checkpoints have to be merged at recovery time in order generate a full checkpoint. The remaining 8 bars in Figure 4(b) summarize the runtime and recovery time costs for the case of partial checkpoints being merged after 8 and 16 partial checkpoints have been taken.

When focusing on runtime, it is clear that despite the additional time to merge the partial checkpoints, the overall cost of checkpointing is much smaller for the partial checkpointing algorithms than the full checkpointing algorithm with higher locality skew(10% and 20%). However, there is less advantage for pCALC with lower locality skew(50%).

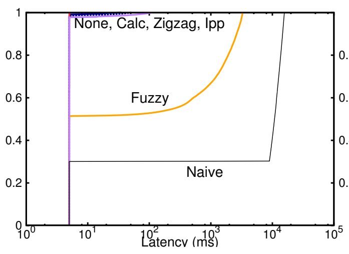
The runtime vs recovery time tradeoff is clear as one compares the three different sets of bars corresponding to the partial checkpoint batch sizes of 4, 8, and 16. pCALC's runtime improvement relative to CALC increases as the batch size of partial checkpoints to merge increases. However, recovery time increases linearly. Nonetheless, we expect that pCALC with batch sizes within our experimental range will be the preferred option for most workloads, since the background merge process keeps the recovery time tractable.

5.1.4 *Transaction Latency*

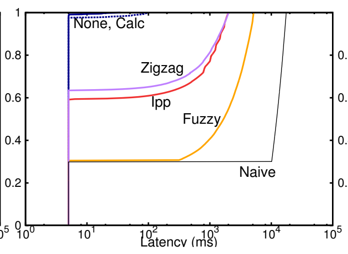
Next, we measure transaction latency for each checkpointing algorithm. We collect transaction latency from the experiments run in Section 5.1.1, in which we run the database system for 100 seconds and take a checkpoint at the 30 second mark. Figures 5(a) and 5(b) show cumulative distribution functions for latencies observed when the database receives transactions at a rate that keeps it running at 90% of its maximum capacity. We also measure transaction latency for a less intense input transactional workload that keeps the database at 70% maximum capacity in Figures 5(c) and 5(d).

When the input transactional workload is intense (keeping the database system at 90% of its maximum capacity) the latencies of the naive and fuzzy schemes are very poor. This is because the naive and fuzzy schemes quiesce the database temporarily during the checkpoint period. All transactions that enter the system during this period thus get queued and wait for the database to resume processing. This queue gets continuously bigger during the checkpoint period. After the database resumes processing, the transactions in this queue get processed. However, new transactions continue to the enter the system and get placed at the end of the queue. Since the input transactional workload is so close to the database system's maximum capacity, the database never gets a chance to "catch up" and shrink the size of the queue. Therefore, all transactions that enter the system after the first time the database is quiesced experience the latency of the quiesce period — even those transactions that enter the system substantially after this period has ended. Thus, the latencies for the naive algorithm are the worst, since its quiesce period is the longest. However, even Zigzag and IPP, which temporarily quiesce the database to create a physical point of consistency when there exist long-running transactions in the database system, experience poor latency in Figure 5(b).

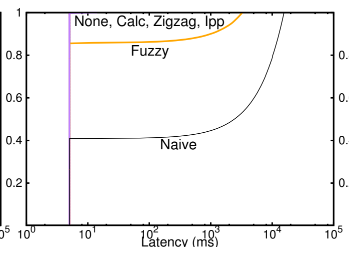
Figures 5(c) and 5(d) show the latencies in perhaps the more realistic scenario, where the database is not running at full capacity, and can use its additional headroom to catch up after transactions get queued during periods of unavailability. In these figures, the latencies of Zigzag, IPP, fuzzy, and naive are much better than than the previous case, but



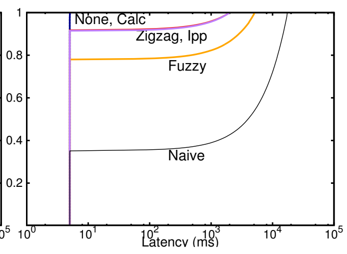
(a) No long xacts, 90% load



(b) Long xacts, 90% load



(c) No long xacts, 70% load



(d) Long xacts, 70% load

Figure 5: Latency distribution.

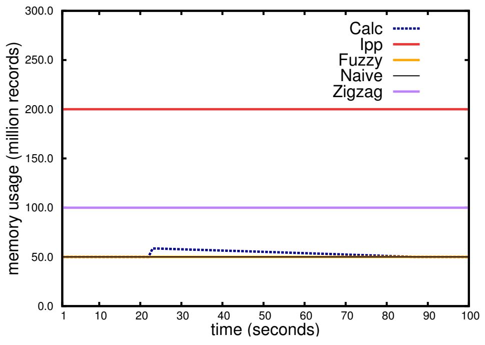


Figure 6: Memory used for record storage over time.

the short period of unavailability while the database was quiesced, in addition to the short catch-up period thereafter, do take their toll on the latency distribution.

For all of these latency experiments, CALC performs very similarly with the "no checkpointing" scheme because CALC does not require physical points of consistency, and never has to quiesce the database system. This experiment shows a major advantage of CALC relative to the other checkpointing schemes. While it is common for main memory database systems to experience hiccups in latency during checkpointing periods, *CALC manages to avoid these hiccups entirely.*

5.1.5 Scalability

We also ran some experiments to see if the database size effects the overhead of CALC. In particular, experimented with database sizes of 10 million records, 50 million, 100 million, and 150 million. We found that the checkpoint duration was linearly correlated with database size. This is not surprising — if the database is larger, so too is the checkpoint. Thus if the database is twice as large, twice as much work has to happen during checkpointing as far as scanning through the stable records and writing them to disk. Accordingly, the total overhead of checkpointing, expressed in terms of total transactions lost, is also linear in database size. Due to lack of space and lack of surprise in the results, we do not present these results in more detail here. However, more detailed results are provided in Appendix A.

5.1.6 Physical Memory usage

One advantage of the naive and fuzzy checkpointing schemes is that they do not need to keep around extra copies of the data in memory.

In contrast, IPP requires keeping 3 copies of all records. Furthermore, during the checkpointing period, IPP must merge the updated records with the last consistent checkpoint in order to construct a new consistent checkpoint that can be written to disk. Therefore, in practice, IPP requires up to 4 copies of the database in memory. Zigzag requires keeping 2 copies of all records.

In the worst case, CALC requires two copies of each record to be stored. However, in practice, the actual memory needed is much less than this. CALC does not require any extra memory until the checkpoint process begins. When CALC enters the *prepare* phase, transactions begin inserting stable record versions on all writes. Once the *capture* phase begins, the checkpointing thread begins erasing these stable versions. Thus CALC only requires extra space for records written during the short period of time in between these two phases, which is generally much smaller than the total number of records updated since the last checkpoint.

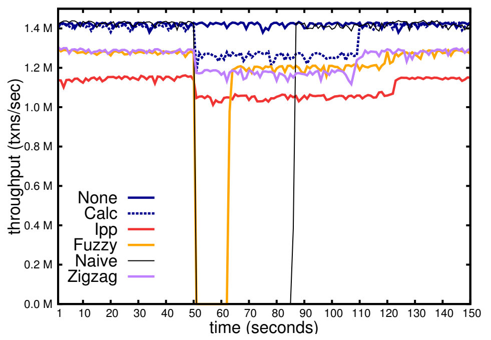
Figure 6 summarizes the space required for each algorithm over the course of a 100 second experiment where a checkpoint is taken at 20 seconds for a 50 million record database. Naive, fuzzy, IPP, and Zig-zag all require constant space. Naive and fuzzy require almost no extra space beyond the original database size. Zigzag and IPP require 2X and 4X database size, respectively. CALC requires no extra space most of the time, but requires around 1.2X database size during its peak requirements during the checkpointing period.

In practice, in order to avoid frequently allocating and erasing stable records, our implementation pre-allocates a pool of space for stable records, so that when a transaction needs to insert stable record, it simply allocates memory for the stable record from the pool. When transactions need to erase the stable record, they simply release the space back into the pool. Therefore, in practice, CALC's memory usage is also flat — it needs as much space as peak requirements — 1.2X database size. Nonetheless, it is clear that its overall memory requirements are much smaller than IPP and Zigzag and similar to Naive and Fuzzy.

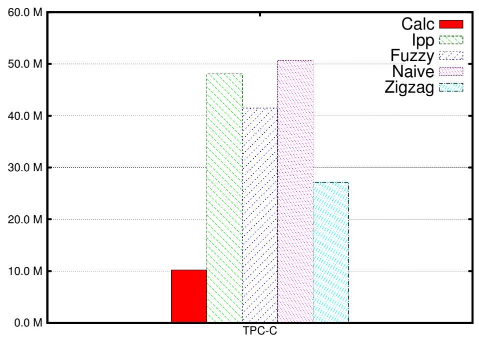
5.2 TPC-C experiments

Finally, we examined the overhead of the checkpointing schemes on the TPC-C benchmark at a scale of 50 TPC-C warehouses. For this experiment, we run a mix of 50% NewOrder and 50% Payment transactions. These two transactions make up 88% of the default TPC-C mix and are the most relevant transactions when experimenting with checkpointing algorithms since they are write-intensive. Figure 7(a) shows TPC-C throughput under maximum load for each checkpointing scheme when a checkpoint is taken at the 50 second mark, and Figure 7(b) plots the total transactions lost during the checkpoint period of Figure 7(a).

The result is very similar to our earlier results for the microbenchmark without long-running transactions, since



(a) Throughput over time.



(b) Transactions lost.

Figure 7: TPC-C workload.

TPC-C does not have a non-read-only transaction of substantial length. However, one interesting difference between these results and the microbenchmark is that Zigzag performs relatively worse relative to CALC on TPC-C. This is because the TPC-C New Order transaction (the most common transaction in the workload) contains many more writes to records than the microbenchmark, which introduces more overhead for Zigzag since it needs to maintain two copies of the record upon each write, even when checkpointing is not actively ongoing.

6. RELATED WORK

There have been several previous attempts to capture checkpoints asynchronously. Dewitt et al. [3] 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 ΔMem. Once the checkpoint completes, the pages in ΔMem 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 ΔMem.

Granular tuple locking allowing for incremental checkpoint captures has also been previously investigated by Pu [15]. 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.

Some 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 [24]. However, despite the 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 costly latency spikes.

Zheng et al. presented SiloR [25], a logging, checkpointing, and recovery subsystem for a very fast in-memory database. However, they do not create transaction-consistent checkpoints.

VoltDB [13] proposed an asynchronous checkpointing technique which takes checkpoints by making every database record ’copy-on-write’. However, like many of the other schemes presented in the paper, it requires a physical point of consistency — a key requirement that CALC drops. Furthermore, their scheme relies on VoltDB’s deterministic design, and is thus not generally applicable to all database systems.

Fuzzy checkpointing has become an extremely popular checkpointing algorithm in database systems. There have been several proposals for applying it to main memory [12, 5, 11]. Salem et al. argues that fuzzy checkpointing is the most efficient [17]. Oracle’s main memory database TimesTen[8] also supports fuzzy checkpointing. However, fuzzy checkpointing requires a database log, a requirement that we avoided in our work.

Microsoft’s main memory database system, Hekaton [4], proposed a fast partial checkpoint algorithm. The basic idea is to continually scan the tail of the log and create data and delta checkpoint files. In contrast, CALC is designed for an entirely different type of main memory system — a system that does not have any redo log at all. Furthermore, the Hekaton technique is specifically designed for multi-versioning systems, while CALC is more general.

Hyper proposed a consistent snapshot mechanism through a UNIX system call to fork(), and OS-based copy-on-update [7]. However, it requires the physical point of consistency and does not support partial checkpoints.

7. CONCLUSION

We have presented CALC and pCALC, novel methods for asynchronous checkpointing that require minimum additional memory usage, no physical point of consistency in the application state, and extremely low runtime overhead. Our experiments show that CALC and pCALC have an overhead that is 2-10X less than alternative approaches — even promising approaches that were introduced recently. Furthermore, it avoids the temporary latency spikes that are common in other checkpointing approaches.

Acknowledgements. This work was sponsored by the NSF under grant IIS-1527118.

8. REFERENCES

[1] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. seconod edition edition, 2009.

[2] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In

Proceedings of the 2011 international conference on Management of data, SIGMOD '11, pages 265–276, 2011.

[3] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[4] C. Diaconu, C. Freedman, E. Ismert, P. ÃĚke Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, 2013.

[5] R. B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Trans. Comput.*, 35:839–843, September 1986.

[6] S. Harizopoulos, D. J. Abadi, S. R. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[7] A. Kemper and T. Neumann. Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. *Proc. of ICDE*, pages 195–206, June 2011.

[8] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *In IEEE Data Engineering Bulletin.*, June 2013.

[9] E. Lau and S. Madden. An integrated approach to recovery and high availability in an updatable, distributed data warehouse. In *Proc. of VLDB*, pages 703–714, 2006.

[10] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka1, H. Plattner, J. Krueger, and M. Grund. High-performance transaction processing in sap hana. *In IEEE Data Engineering Bulletin.*, June 2013.

[11] X. Li and M. H. Eich. Post-crash log processing for fuzzy checkpointing main memory databases. In *Procceedings of the Ninth International Conference on Data Engineering*, Apr. 1993.

[12] J.-L. Lin and M. H. Dunham. Segmented fuzzy checkpointing for main memory databases. In *Procceedings of the ACM Symposium on Applied Computing*, February 1996.

[13] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proc. of ICDE*, 2014.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17:94–162, March 1992.

[15] C. Pu. On-the-fly, incremental, consistent reading of entire databases. In *Proceedings of the 11th international conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 369–375. VLDB Endowment, 1985.

[16] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. PVLDB 7(10): 821-832, 2014.

[17] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *In Proc. ICDE*, 1989.

[18] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts.* 5 edition, 2006.

[19] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, Trondheim, Norway, 2005.

[20] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, Vienna, Austria, 2007.

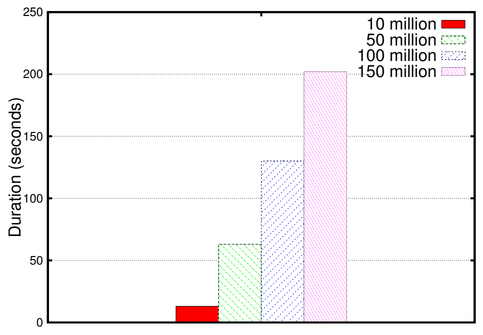
[21] A. Thomson and D. J. Abadi. The case for determinism in database systems. *VLDB*, 2010.

[22] A. Thomson, T. Diamond, P. Shao, K. Ren, S.-C. Weng, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

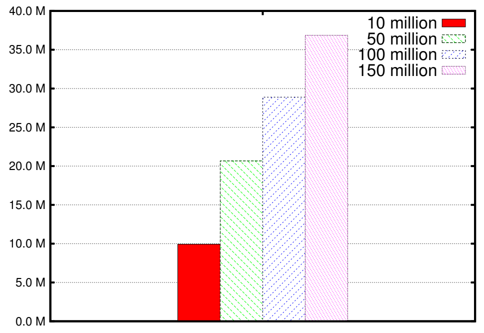
[23] VoltDB. http://voltdb.com.

[24] A. Whitney, D. Shasha, and S. Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++. In *Int. Workshop on High Performance Transaction Systems*, 1997.

[25] W. Zheng and S. Tu. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, 2014.



(a) Checkpoint duration.



(b) Transactions lost.

Figure 8: Scalability.

APPENDIX

A. SCALABILITY RESULTS

Figure 8 shows the result of an experiment in which we examine the scalability for the CALC algorithm. We scale from 10 million records to 150 million records. We measure the checkpoint duration and total transactions lost for each database size setting when running 300 seconds that contains one full checkpoint. As can be seen, the CALC algorithm is linearly scalable. If the database is twice as large, twice as much work has to happen during checkpointing as far as scanning through the stable records and writing them to disk. Accordingly, the total overhead of checkpointing, expressed in terms of total transactions lost, is also linear in database size.

Note that since the recording of a checkpoint is limited by disk bandwidth in our system, the time to complete a checkpoint is a direct measure of total disk IO. Therefore, using higher bandwidth disks will reduce both the checkpoint duration and transactions lost.