Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications

Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue,
Alan Demers, Johannes Gehrke, Walker White
Cornell University
Ithaca, NY 14853, USA
{tuancao,vmarcos,sowell,yaoyue,ademers,johannes,wmwhite}@cs.cornell.edu

ABSTRACT

Advances in hardware have enabled many long-running applications to execute entirely in main memory. As a result, these applications have increasingly turned to database techniques to ensure durability in the event of a crash. However, many of these applications, such as massively multiplayer online games and main-memory OLTP systems, must sustain extremely high update rates – often hundreds of thousands of updates per second. Providing durability for these applications without introducing excessive overhead or latency spikes remains a challenge for application developers.

In this paper, we take advantage of frequent *points of consistency* in many of these applications to develop novel checkpoint recovery algorithms that trade additional space in main memory for significantly lower overhead and latency. Compared to previous work, our new algorithms do not require any locking or bulk copies of the application state. Our experimental evaluation shows that one of our new algorithms attains nearly constant latency and reduces overhead by more than an order of magnitude for low to medium update rates. Additionally, in a heavily loaded main-memory transaction processing system, it still reduces overhead by more than a factor of two.

Categories and Subject Descriptors

H.2.2 [**Information Systems**]: Database Management—*Recovery and restart*

General Terms

Algorithms, Performance, Reliability

1. INTRODUCTION

An increasing number of data-intensive applications are being executed entirely in main memory and eschewing traditional database concurrency control mechanisms in order to achieve high throughput. Examples include applications as diverse as massively multiplayer online games (MMOs) and scientific simulations, as well as certain classes of main-memory OLTP systems and search engines. Many of these systems either serialize operations and execute them sequentially [36, 28] or develop application-specific ways to avoid locking and prevent costly conflicts and rollbacks [34, 35, 29]. These approaches lead to applications with *frequent points of consistency*, which we call *frequently consistent* or FC applications. Unlike traditional database applications, which may never reach a point of consistency without quiescing the system, FC applications reach natural points of consistency very frequently (typically at least once a second) during normal operation.

Since FC applications store data in main memory, durability is an important and challenging property to ensure, particularly given the strict performance requirements of many FC applications. In this paper we leverage frequent points of consistency to develop checkpoint recovery algorithms with extremely low overhead. We start by describing several important use cases.

MMOs are an important class of FC applications that have recently received attention in the database community [27, 11]. MMOs are large persistent games that allow users to socialize and compete in a virtual world. Most MMOs use a time-stepped processing model where character behavior is divided into atomic time steps or *ticks* that are executed many times per second and update the entire state of the game [35]. The game state is guaranteed to be consistent at tick boundaries. Behavioral and agent-based simulations, which are often used by scientists to model phenomena such as traffic congestion and animal motion, also use a time-stepped model and have similar points of consistency [34].

Though we use MMOs as a running example throughout this paper, our techniques can be applied to any application with frequent points of consistency. For example, certain classes of main-memory OLTP systems also have frequent points of consistency. Traditional OLTP systems are heavily mutli-threaded to mask the huge differences between access times to main memory and disk, but the tradeoffs change when all data is stored in memory. New OLTP systems like H-Store [28] (and its commercial version VoltDB [33]) serialize all transactions so that each machine can execute them using a single thread in order to avoid the overhead of concurrency control and increase throughput. In these applications the end of each transaction marks a point of consistency [32]. Other examples of FC applications include new data-parallel systems to program the cloud, such as BOOM [2], deterministic transaction processing systems [30, 36], and in-memory search engines [29].

Many frequently consistent applications must handle very high update rates, which can complicate recovery. For example, popular MMO servers may have to process hundreds of thousands of updates per second, including behaviors such as character movement. Many traditional database recovery algorithms that rely on physical logging simply cannot sustain this update rate without resorting to expensive special-purpose hardware [20, 24]. Another common

SIGMOD'11, June 12–16, 2011, Athens, Greece.

recovery approach, used by some OLTP systems such as H-Store, is to replicate state on several machines and apply updates to all replicas [28]. While this provides high availability, replication is usually still combined with additional mechanisms such as regular checkpointing to protect against large scale failures such as power outages. Checkpointing can also be used to transfer state between nodes during recovery or when a new replica is added to the system.

We can take advantage of frequent points of consistency to take periodic *consistent checkpoints* of the entire application state and use *logical logging* to provide durability between checkpoints. Since we have frequent points of consistency, there is no need to quiesce the system in order to take a checkpoint, and we benefit from not having to maintain a costly physical log. Furthermore, we can avoid the primary disadvantage of logical logging, namely the cost of replaying the log during recovery. Since there are many points of consistency, we can take checkpoints very frequently (every few seconds), so the log can be replayed very fast.

Of course, taking very frequent checkpoints can increase the overhead associated with providing durability. To make this feasible, we need high performance checkpointing algorithms. In particular, we have identified several important requirements. First, the algorithm should have low overhead during normal operation. Ideally, taking a checkpoint should have very little impact on the throughput of the system. Second, the algorithm should distribute its overhead uniformly and not introduce performance spikes or highly variable response times. This is particularly important in applications such as MMOs where high latency may create "hiccups" in the immersive virtual experience. Finally, it should be possible to take checkpoints very frequently so that the logical log can be replayed quickly in the event of a failure. In a recent experimental study, Vaz Salles et al. evaluated existing main-memory checkpointing algorithms for use in MMOs [27], however these algorithms either use locks or large synchronous copy operations, which hurt throughput and latency, respectively.

In this paper we propose and evaluate two new checkpointing algorithms, called Wait-Free Zigzag and Wait-Free Ping-Pong, which avoid the use of locks and are designed to distribute overhead smoothly. Wait-Free Ping-Pong also makes use of additional main memory to further reduce overhead. We evaluate these algorithms using a high-performance implementation of TPC-C running in the cloud as well a synthetic workload with a wide variety of update rates. Our experiments show that both algorithms are successful in greatly reducing the latency spikes associated with checkpointing and that Wait-Free Ping-Pong also has considerably lower overhead: up to an order of magnitude less for low to medium update rates and more than a factor of two in our TPC-C experiments.

In Section 2 of this paper, we review several existing checkpointing algorithms [27]. We then make the following contributions:

1. We analyze the performance bottlenecks in prior work, and propose two new algorithms to address them (Section 3).
2. We explore the impact of data layout in main memory on cache performance and do a careful cache-aware implementation of all of our algorithms, as well as the best previous algorithms [27]. We find that our algorithms are particularly amenable to these low level optimizations (Section 4).
3. We perform a thorough evaluation of our new algorithms and compare them to existing methods. We find that Wait-Free Ping-Pong exhibits lower overhead than all other algorithms by up to an of magnitude over a wide range of update rates. It also completely eliminates the latency spikes that plagued previous consistent checkpointing algorithms (Section 5).

We review related work in Section 6 and conclude in Section 7.

2. BACKGROUND

A *frequently consistent* (FC) application is a distributed application in which the state of each node frequently reaches a point of consistency. A point of consistency is simply a time at which the state of the node is valid according to the semantics of the application. The definition of frequently depends on the application, but we expect it to be less than a second. For example, MMO servers execute approximately 10 ticks/sec, while modern OLTP systems may be able to execute a small transaction in tens of microseconds [28, 17].

Throughout the paper, we use *application state* to refer to the *dynamic, memory-resident state* of an FC application. FC applications may have additional read-only or read-mostly state that can simply be written to stable storage when it is created. For example, in an MMO, each character will have some attributes such as position, health, or experience that are frequently updated and thus part of the dynamic state, but it will also have other attributes such as name, race, or class (the type or job of the character) that are unlikely to change. Additionally, some applications include secondary data structures such as indices that can be rebuilt during recovery. We will focus exclusively on making the primary dynamic state durable.

In this paper, we use *coordinated checkpointing* to provide durability for distributed FC applications at low cost [6]. Many applications already use replication to increase availability and provide some fault tolerance [28], but checkpointing can still be used to copy state during recovery and provide durability to protect against power outages and other widespread failures.

Furthermore, we will focus exclusively on developing robust *single-node* checkpointing algorithms, as these form the core of most distributed checkpointing schemes. There is a large distributed systems literature that explores how to generalize efficient single-node checkpointing algorithms to multiple nodes. Tightly synchronized FC applications that reach global points of consistency during normal operation are particularly easy to checkpoint, as they can reuse their existing synchronization mechanisms to decide on a global point of consistency at which each node can take a local checkpoint. More general distributed applications may need to use techniques such as message logging to create a consistent checkpoint. These distributed approaches are discussed at length in a recent survey by Elnozahy et al [9].

In the remainder of this section we discuss requirements for checkpoint-recovery algorithms targeted at FC applications (Section 2.1) and we present existing checkpoint-recovery methods for these applications (Sections 2.2 and 2.3).

2.1 Requirements for Checkpoint Recovery Algorithms

We can summarize the requirements for a checkpointing algorithm for FC applications as follows:

1. The method must have *low overhead*. During normal operation, FC applications must process very high update rates, and the cost of checkpointing state for recovery should not greatly reduce the throughput.
2. The method should *distribute overhead uniformly*, so that application performance is consistent. Even when the total overhead is low, many applications depend on predictable performance. For example, fluctuations in overhead affect MMOs by interfering with time-synchronized subsystems like the physics engine [16]. This problem is made even worse when the checkpointing algorithm must pause the system in order to perform a synchronous operation like a bulk

Interface 1: Algorithmic Framework

```
Mutator::PrepareForNextCheckpoint()
Mutator::PointOfConsistency()
Mutator::HandleRead(index)
Mutator::HandleWrite(index, update)

AsynchronousWriter::WriteToStableStorage()
```

copy. Such *latency spikes* can also be a problem for distributed applications. If a node sending a message experiences a latency spike, the receiver may block, spreading the effect of the latency spike throughout the system and reducing overall throughput.

3. The method should have *fast recovery* in the event of failure. Ideally, the recovery time should be on the order of a few seconds to avoid significant disruption. We will accomplish this primarily by increasing the frequency of checkpoints.

Traditional approaches to database recovery such as ARIES-style physiological logging [21] and fuzzy checkpointing [12, 26], which uses physical logging, cannot keep up with the extremely high update rates of FC applications without resorting to extremely expensive special-purpose hardware such as battery backed RAM [10, 20, 24]. As the dollar cost per gigabyte of battery-backed RAM exceeds the cost of traditional RAM by over an order of magnitude, it is unlikely that cloud infrastructures or large enterprise clusters will package this technology for use by FC applications.

In the following subsections, we review the framework we previously proposed for checkpointing MMOs on commodity hardware and discuss how it can be extended to support FC applications [27].

2.2 Algorithmic Framework

In this section we discuss the basic algorithmic framework we use for checkpointing. During normal operation, the application takes *periodic checkpoints* of its entire dynamic state and maintains a *logical log* of all actions. For example, in an MMO we would log all user actions sent to the server; in a system like H-Store, we would log all stored procedure calls. Because each logical update may translate into many physical updates, we expect the overhead of maintaining this log to be quite small, and we will focus primarily on the overhead associated with checkpointing. In order to compare with existing algorithms for games, we will use the same algorithmic framework for checkpointing originally introduced by Vaz Salles et al. [27]. Interface 1 lists the key methods in this API.

We model main-memory checkpointing algorithms using a single *Mutator* thread that executes the application logic and synchronously updates the application state. Our techniques can be naturally extended to support applications with multithreaded mutators as long as we include enough information in the logical log to enable deterministic replay. For OLTP applications, for instance, this might include transaction commit order so that transactions can be serialized during replay. This may lengthen the recovery time somewhat, but in practice we have observed that many FC applications are already deterministic or can be made deterministic using existing methods [30].

In addition to the Mutator, we use two threads to write data to disk. The *Logical Logger* thread synchronously flushes logical log entries to disk. This could be done directly in the Mutator thread, but we implement it as a separate thread so that we can overlap computation and process additional actions while we are waiting for the disk write to complete. Note that we must wait for all disk writes to finish before proceeding to the next point of consistency (e.g. reporting a transaction as committed).

The final thread, the *Asynchronous Writer*, writes some or all of the main-memory state to disk. Note that this thread can be run in the background while the Mutator concurrently updates the state. Since the performance of the Asynchronous Writer thread is primarily determined by disk bandwidth, we will focus on the synchronous overhead introduced in the Mutator thread. However, it is important to note that some existing checkpointing algorithms require that the Asynchronous Writer acquire locks on portions of the application state, which can impact the performance of all of the threads [27].

The Mutator thread makes function calls before starting a new checkpoint (`PrepareForNextCheckpoint`), when a point of consistency is reached (`PointOfConsistency`), and during each read and write of the application state (`HandleRead` and `HandleWrite`). The `PointOfConsistency` method must be executed at a point of consistency, but it need not be executed at every point of consistency. If points of consistency are very frequent (e.g., every few microseconds), we can wait for several of them to pass before calling the method.

Different algorithms will implement these methods differently depending on how they manage the application state. Most algorithms will maintain one or more *shadow copies* of the state in main memory. The Mutator may use this *shadow state* during the checkpoint before it is written to stable storage by the Asynchronous Writer. In the rest of the paper, we will refer to the time between successive calls to `PrepareForNextCheckpoint` as a *checkpoint period*. Note that this must be long enough for the Asynchronous Writer to finish creating a checkpoint on disk. These checkpoints may be organized on disk in several different ways. In our implementation, we use a double-backup organization for all of the algorithms, as it was reported in previous work to consistently outperform a log-based implementation [27].

The recovery procedure is the same for all algorithms that implement this framework. First, the most recent consistent checkpoint is read from disk and materialized as the new application state. Then, the logical log is replayed from the time of the last checkpoint until the state is up-to-date. Since we take checkpoints very frequently, the time to replay the logical log is quite small.

2.3 Existing Algorithms

Based on their experimental evaluation for MMOs, Vaz Salles et al. concluded that there was no single checkpointing algorithm that outperformed all the others over the entire range of update rates [27]. Their evaluation included a number of synthetic experiments, and we believe they provide a reasonable model for many FC applications. They concluded that two algorithms, *Copy-on-Update* and *Naive-Snapshot*, performed best for low and high update rates, respectively.

Naive-Snapshot synchronously copies the entire state of the application to the shadow copy at a point of consistency and then writes it out asynchronously to disk. Naive-Snapshot is among the best algorithms when the update rate is very high since it does not perform any checkpoint-specific work in the `HandleRead` or `HandleWrite` functions.

Copy-on-Update groups application objects into blocks and copies each block to the shadow state the first time it is updated during a checkpoint period. During a checkpoint, the Asynchronous writer either reads state from the application state or the shadow copy based on whether the corresponding block has been updated. Since the Mutator is concurrently updating the application state, it must acquire locks on the blocks it references. This may introduce considerable overhead. By varying the memory block size, Copy-on-Update can trade off between copying and locking overhead.

Method	Overhead Factor			
	Bulk Copying	Locking	Bulk Bit-Array Reset	Memory Usage
Naive-Snapshot	Yes	No	No	×2
Copy-on-Update	No	Yes	Yes	×2
Wait-Free Zigzag	No	No	Yes	×2
Wait-Free Ping-Pong	No	No	No	×3

Table 1: Overhead factors of checkpoint-recovery algorithms.

3. NEW ALGORITHMS

In this section, we present two novel checkpoint recovery algorithms for FC applications. We first discuss important design tradeoffs that differentiate our algorithms from state-of-the-art methods (Section 3.1) and then introduce each algorithm in turn (Sections 3.2 and 3.3).

3.1 Design Overview

We have identified four primary factors that affect the performance of checkpoint recovery algorithms:

1. **Bulk State Copying:** The method may need to pause the application to take a snapshot of the whole application state, as in Naive-Snapshot.
2. **Locking:** The method may need to use locking to isolate the Mutator from the Asynchronous Writer, if they work on shared regions of the application state.
3. **Bulk Bit-Array Reset:** If the method uses metadata bits to flag dirty portions of the state, it may need to pause the application and perform a bulk clean-up of this metadata before the start of a new checkpoint period.
4. **Memory Usage:** In order to avoid synchronous writes to disk, the method may need to allocate additional main memory to hold copies of the application state.

Table 1 shows how the factors above apply to both Naive-Snapshot and Copy-on-Update. Naive-Snapshot eliminates all locking and bulk bit-array resetting overhead, but must perform a bulk copy of the whole application state. This introduces a latency spike in the application since it must block during the copy. Copy-on-Update avoids this problem since it does not perform synchronous bulk copies, but incurs both locking and bulk bit-array resetting overhead. This extra overhead is small for moderate update rates, but is significant for higher update rates [27]. Both methods require additional main memory of roughly the size of the entire application state. So the memory usage for the dynamic application state increases to about twice its original size.

Our new algorithms, Wait-Free Zigzag and Wait-Free Ping-Pong, are designed to eliminate all overhead associated with bulk state copying and locking. Unlike Naive-Snapshot, they spread their overhead over time instead of concentrating it at a single point of consistency. Unlike Copy-on-Update, they only require synchronization between the Mutator and Asynchronous Writer at the end of a checkpoint period. This eliminates all locking overhead during state updates. Additionally, the Mutator and Asynchronous Writer are each guaranteed to make progress even if the other is preempted within a checkpoint period. As a consequence, both algorithms are wait-free within a checkpoint period [13]. Table 1 also summarizes the overhead factors our algorithms incur.

To eliminate overhead factors, both of our new algorithms strictly separate the state being updated by the Mutator from the state being read by the Asynchronous Writer. In addition, both track updates at very fine, word-level granularity. However, the algorithms use different amounts of main memory. Wait-Free Zigzag, like Naive-Snapshot and Copy-On-Update, uses additional main memory on the order of the size of the application state. Wait-Free Ping-Pong, however, requires twice that amount. We believe that this is a reasonable tradeoff for most MMOs and OLTP applications, as the size of the dynamically updated state is generally quite small – usually only a small fraction of the whole state.

3.2 Wait-Free Zigzag

The main intuition behind Wait-Free Zigzag is to maintain an untouched copy of every word in the application state for the duration of a checkpoint period. These copies form the consistent image that is written to disk by the Asynchronous Writer. As these copies are never changed during the checkpoint period, the Asynchronous Writer is free to read them without acquiring locks.

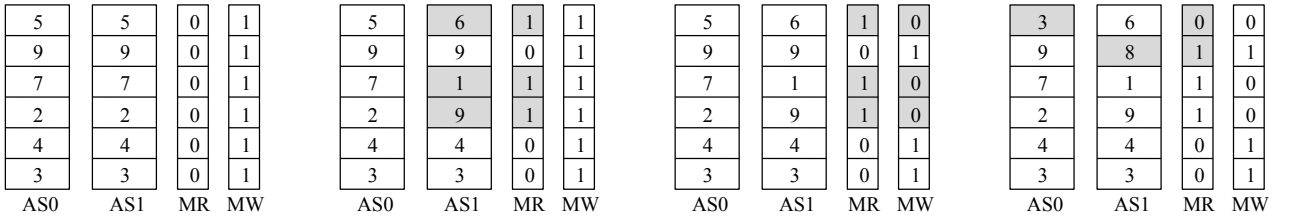
The algorithm starts with two identical copies of the application state: AS_0 and AS_1 (Figure 1(a)). For each word i in the application state, we maintain two bits: $MR[i]$ and $MW[i]$. The first bit, $MR[i]$, indicates which application state should be used for Mutator reads from word i, while the second bit, $MW[i]$, indicates which should be used for Mutator writes. The bit array MR is initialized with zeros and MW with ones.

The bits in MW are never updated during a checkpoint period. This ensures that for every word i, $AS_{\neg MW[i]}[i]$ is also never updated by the Mutator during a checkpoint period. The Asynchronous Writer flushes exactly these words to disk in order to take a checkpoint. To avoid blocking the Mutator, however, we must also apply updates to the application state. Whenever a new update comes to word i, we write that update to position $AS_{MW[i]}[i]$ and set $MR[i]$ to $MW[i]$. Before any read of a word i, the Mutator inspects $MR[i]$ and directs the read to the most recently updated word $AS_{MR[i]}[i]$. Figure 1(b) shows the situation after updates are applied to the shaded words. For example, the value written for the third word by the Asynchronous Writer resides in AS_0 and remains unchanged during the first checkpoint. Any reads by the Mutator after the first update return the value in AS_1.

At the end of the checkpoint period, the Mutator assigns the negation of MR to MW, i.e., $\forall i, MW[i] := \neg MR[i]$. This is done so that the current state of the application is not updated by the Mutator during the next checkpoint period and can be written to disk by the Asynchronous Writer. Figure 1(c) shows the state right after the Mutator performs this assignment in our example. Again, updates during the next checkpoint period follow the same procedure outlined above. Figure 1(d) shows the application state after two updates during the second checkpoint period. Note that the current application state, as well as the state being checkpointed to disk, is now distributed between AS_0 and AS_1.

Algorithm 1 summarizes Wait-Free Zigzag. While most of this algorithm is a straightforward translation of the explanation above, one further observation applies to the `PointOfConsistency` procedure. This Mutator procedure checks whether the Asynchronous Writer has finished writing the current checkpoint to disk. Though the threads communicate, this does not violate the wait-free property of the algorithm, as it can be implemented using store and load barriers instead of locks. Heavier synchronization methods are only necessary at the end of every checkpoint period.

Wait-Free Zigzag does not require any lock synchronization during a checkpoint period. In addition, there is no need to copy memory blocks in response to an update from the Mutator. This eliminates some of the largest overheads of Copy-on-Update. In addition, Wait-Free Zigzag distributes its overhead smoothly and avoids the latency spikes of Naive-Snapshot. On the other hand, Wait-Free Zigzag adds bit checking and setting overhead to both reads and writes issued by the Mutator. It also exhibits a small



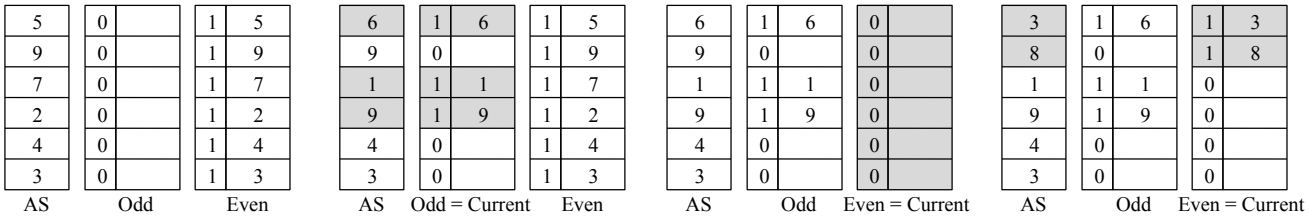
(a) At the beginning of time, AS_0 and AS_1 contain the same information

(b) During the first checkpoint period, some updates from the Mutator are applied

(c) The state right after switching to the second checkpoint period

(d) In the second checkpoint period, the Mutator applies additional updates

Figure 1: Wait-Free Zigzag Example



(a) At the beginning of time, *AS* and *Even* contain the same information

(b) During the first checkpoint period, *Odd* collects updates, while *Even* is flushed to disk

(c) The state right after switching to the second checkpoint period

(d) In the second checkpoint period, *Odd* and *Even* invert roles

Figure 2: Wait-Free Ping-Pong Example

Algorithm 1: Wait-Free Zigzag

input:

```
/* ApplicationState is a vector containing words */
ApplicationState AS_0 ← initial application state
ApplicationState AS_1 ← initial application state
/* size of application state in words */
sizeWords ← |AS_0|
/* reads from the Mutator reference AS_MR[k] */
BitArray MR ← {0,0,...,0}
/* writes from the Mutator affect AS_MW[k] */
BitArray MW ← {1,1,...,1}
```

Mutator::PrepareForNextCheckpoint()

1: **for** $i = 0$ to *sizeWords* **do**
2: $MW[i] \leftarrow \neg MR[i]$
3: **end for**

Mutator::PointOfConsistency()

1: **if** Asynchronous Writer done **then**
2: PrepareForNextCheckpoint()
3: NotifyAsynchronousWriter()
4: **end if**

Mutator::HandleRead(index)

1: **return** $AS_{MR[index]}[index]$

Mutator::HandleWrite(index, newValue)

1: $AS_{MW[index]}[index] \leftarrow newValue$
2: $MR[index] \leftarrow MW[index]$

AsynchronousWriter::WriteToStableStorage()

1: **loop**
2: WaitForMutatorNotification()
3: **for** $k = 0$ to *sizeWords* **do**
4: **write-to-disk** $AS_{\neg MW[k]}[k]$
5: **end for**
6: **end loop**

latency peak associated with negating the bit array at checkpoint boundaries.

3.3 Wait-Free Ping-Pong

All algorithms we have analyzed so far may introduce latency spikes at the end of a checkpoint period due to either synchronous copying or bulk bit-array reset. In this section, we present Wait-Free Ping-Pong, an algorithm that invests extra main memory and extra work per update to avoid these peaks. Wait-Free Ping-Pong uses a total of three versions of the application state. Two of these are used to ensure that the Mutator and Asynchronous Writer always access separate versions of the state and never have to acquire locks. The final copy allows Wait-Free Ping-Pong to do only a constant amount of work at checkpoint boundaries. Rather than performing a large copy or linear time bit-array reset, it only needs to swap two pointers before starting the next checkpoint.

The three copies of the state maintained by Wait-Free Ping-Pong are called *AS*, *Odd*, and *Even*. The Mutator thread reads from *AS* and applies each update to both *AS* and one of the other copies (either *Odd* or *Even*). The Asynchronous Writer uses the other copy to construct a consistent checkpoint that it writes to disk in the background. At the end of the checkpoint period the roles of *Odd* and *Even* are switched so that new updates can be flushed to disk. In order to avoid unnecessary disk writes, each word in *Odd* and *Even* has an associated mark bit that indicates whether it has been updated during the current checkpoint period. The Asynchronous Writer merges those words that have their mark bits set with the previous checkpoint in order to create a new consistent checkpoint.

We show the initial state of Wait-Free Ping-Pong in Figure 2(a). *AS* contains the application state, *Odd* is empty, and *Even* contains a copy of *AS*. The Asynchronous Writer will process *Even* and flush to disk all of the words that have a mark bit set. During the first checkpoint period, this corresponds to the whole state. In the meantime, the Mutator applies updates to *AS*. For every such update, the Mutator must guarantee that the corresponding mark bit for the updated word is set on *Odd* and that the update is also applied to *Odd*. The situation after a few mutator updates is shown in

Algorithm 2: Wait-Free Ping-Pong

```
input:
/* ApplicationState is vector containing words */
ApplicationState AS ← initial application state
ApplicationState currentAS
ApplicationState previousAS ← initial application state
/* size of application state in words */
sizeWords ← |AS|
/* dirty words in the current checkpoint */
BitArray currentBA ← {0,0,...,0}
/* dirty words from the last checkpoint */
BitArray previousBA ← {1,1,...,1}
```

Mutator::PrepareForNextCheckpoint()

```
1: /* pointer swapping */
   swap (previousAS, currentAS)
   swap (previousBA, currentBA)
```

Mutator::PointOfConsistency()

```
1: if Asynchronous Writer done then
2:     PrepareForNextCheckpoint()
3:     NotifyAsynchronousWriter()
4: end if
```

Mutator::HandleWrite(index, newValue)

```
1: AS[index] ← newValue
2: currentAS[index] ← newValue
3: currenBA[index] ← 1
```

AsynchronousWriter::WriteToStableStorage()

```
1: loop
2:     WaitForMutatorNotification()
3:     for k = 0 to sizeWords do
4:         if previousBA[k] then
5:             write-to-disk previousAS[k]
6:             previousBA[k] ← 0
7:             previousAS[k] ← empty
8:         else
9:             write-to-disk word k from previous checkpoint
10:        end if
11:    end for
12: end loop
```

Figure 2(b). Updated words are shaded in the figure – their most recent values are present in both *AS* and *Odd*.

At the end of the first checkpoint period, the Asynchronous Writer will have written all of the marked words in *Even* out to disk. In addition, it will have reset their mark bits. For clarity of presentation, we assume that not only the mark bits but also the contents of those words are reset by the Asynchronous Writer. In an implementation, however, this latter action may be skipped for performance. Note that the Mutator is still applying updates to *Odd* up to this point. Now, the Asynchronous Writer is done with this checkpoint period. The next checkpoint period proceeds similarly with the roles of *Odd* and *Even* inverted. This is shown in Figure 2(c). During the next checkpoint period, the algorithm applies updates to *Even* and the words marked in *Odd* are flushed to disk by the Asynchronous Writer. Figure 2(d) displays the situation after a few updates from the Mutator in the second checkpoint period.

Algorithm 2 presents the logic of Wait-Free Ping-Pong. Note that *currentAS* and *currentBA* point to the copy of the application state collecting updates for the current checkpoint period (either *Odd* or *Even*). From a high-level perspective, *currentAS* and *currentBA* may be seen as an in-memory, compressed implementation of a log of updates for this checkpoint period.

Note that as part of the `WriteToStableStorage` method, the Asynchronous Writer must merge the words updated during the most recent checkpoint (*previousAS*[*k*]) with the last consistent checkpoint in order to construct a new consistent checkpoint that can be written to disk (lines 5 and 9). This merge can be done in one of two ways. In the first method, which we call Copy, the Asynchronous Writer maintains an extra copy of the application state which it "rolls forward" by applying the new updates before flushing the full checkpoint to disk. In the second method, called Merge, the Asynchronous Writer reads the most recent checkpoint from disk and applies the new updates before streaming the new checkpoint to disk. Note that the updates can be applied as the checkpoint is read, so it is not necessary to maintain an additional copy of the state in main memory. We compare these alternatives in Section 5.5

Wait-Free Ping-Pong introduces negligible overhead to the Mutator at the end of a checkpoint period; only simple pointer swaps are needed. Thus, there is no single point in time at which the algorithm introduces a latency peak. On the other hand, this algorithm doubles the number of updates, as each update is applied both to the application state and to a copy.

4. IMPLEMENTATION

Since all of the algorithms we evaluate make frequent access to multiple copies of the application state in main memory, cache and TLB performance are important considerations to reduce overhead. In this section, we describe the important features of our cache-optimized implementations.

4.1 Existing Algorithms

We start by describing our implementations of the two existing algorithms: Naive-Snapshot and Copy-on-Update.

Naive Snapshot (NS). As this algorithm is relatively straightforward, we focused on making the memory copy at the beginning of a new checkpoint period as efficient as possible. With microbenchmarks, we observed that a `memcpy` of a memory-aligned application state was better than our attempts to manually unroll the copy loop.

Bit-Array Packed Copy-on-Update (BACOU). The main data structures used by Copy-on-Update include the primary and shadow states maintained by the algorithm, bit arrays with metadata on dirty memory blocks, and lock information for these blocks. In order to minimize the overhead of bulk bit-array resetting, we packed the bits into (64 byte) cache lines and used long word instructions for all operations. Furthermore, we interleaved blocks of the primary and shadow copies of the application state into one cache line, so that they will be fetched together. This optimized implementation gave us a factor of five improvement over a naive implementation of the algorithm.

4.2 Wait-Free Zigzag

The Wait-Free Zigzag algorithm has two major sources of overhead: the bit array lookups in the `handleRead` and `handleWrite` routines, and the bulk negation in the `prepareForNextCheckpoint` routine. To address these sources of overhead, we devised the following data layout variations.

Naive Wait-Free Zigzag (NZZ). This is the naive translation of the algorithm presented in Section 3.2. We represented each of AS_0, AS_1, *MR*, and *MW* as a separate array in main memory and encoded each bit of *MR* and *MW* as one byte for efficient access.

Interleaved Wait-Free Zigzag (IZZ). In this variant, we interleaved the main-memory layout of AS_0, AS_1, *MR*, and *MW*. Each cache line holds a fixed number of *interleaved records*, containing a word from each data structure, stored in order. Placing all of the

words necessary for `handleRead` and `handleWrite` in the same cache line reduces memory stalls on read and write instructions.

Packed Wait-Free Zigzag (PZZ). This variant is similar to IZZ, but organizes data inside of a cache line differently. Instead of laying out interleaved records row-at-a-time, we laid them out column-at-a-time, in a style reminiscent of PAX [1]. This maintains the benefits for `handleRead` and `handleWrite`, while allowing `prepareForNextCheckpoint` to be implemented more efficiently with long word negation instructions.

Bit-Array Packed Wait-Free Zigzag (BAZZ). We observed experimentally that negating the *MR* bits at the end of each checkpoint period was the major source of overhead in Wait-Free Zigzag (Section 5.2). As with BACOU, we optimized this bulk negation by combining the representation of *MR* and *MW* into a single bit-packed array. We divided each cache line in the array in half, and used the first half for the bits of *MR* and the second half for bits of *MW*. In a system with a cache line size of 64 bytes, each cache line encodes 256 bits from each array. We negated the bits of *MR* efficiently with long word instructions.

4.3 Wait-Free Ping-Pong

Unlike Wait-Free Zigzag, Wait-Free Ping-Pong has a very inexpensive `prepareForNextCheckpoint` routine. On the other hand, it must write to two copies of the application state during each update. With this in mind, we investigate the following two variants of Wait-Free Ping-Pong.

Naive Wait-Free Ping-Pong (NPP). In this variant, we allocated *AS*, *Odd*, *Even*, and their respective bit arrays as independent arrays in main memory. Additionally, we also represented each bit using one byte in order to avoid bit encoding overhead on writes.

Interleaved Wait-Free Ping-Pong (IPP). As in the corresponding variant for Wait-Free Zigzag, we interleaved the memory layout of *AS*, *Odd*, *Even*, and their respective bit arrays. Each cache line contains a set of interleaved records with one word from each data structure in sequence. In this way, the additional writes of `handleWrite` fall on the same cache line as the original write to the application state. We thus expect to eliminate any DTLB or cache miss overheads associated with the additional writes of Wait-Free Ping-Pong using this organization.

We also applied a number of other optimizations, including using different page sizes, eliminating conditionals, and aggressively inlining code. Using large pages resulted in a considerable performance improvement, which we report in Section 5.6.

5. EXPERIMENTS

In this section we compare the performance of Wait-Free Zigzag and Wait-Free Ping-Pong with existing algorithms. We consider several metrics in our evaluation. First, we look at the synchronous overhead per checkpoint. This added overhead indicates the total amount of work done by the checkpointing algorithm during a checkpoint period. We also measure how the overhead is distributed over time in order to see whether the checkpointing algorithms introduce any unacceptable latency peaks.

5.1 Setup and Datasets

We compare Wait-Free Zigzag and Wait-Free Ping-Pong with Naive-Snapshot and Copy-on-Update using two different synthetic workloads and a main-memory TPC-C application.

Synthetic Workloads. For the synthetic workloads, we produced trace files containing the sequence of physical updates to apply to the application. We make these traces sufficiently large to avoid transient effects and keep them in main memory to avoid I/O effects. Updates are applied as fast as possible by our benchmark, but to normalize results for presentation, we group updates into intervals that correspond to 0.1 seconds of simulated application logic. In addition, to meaningfully compare the overhead of checkpointing algorithms, we ensure that the checkpointing interval is the same for all algorithms.

In the first workload, we model the application state as a set of 8 KB objects. Different FC applications may have different data models or schemas, but this is a reasonable general model that allows us to vary the state size by changing the number of objects. We use 25,000 rows as a default, which yields approximately 200 MB of dynamic state. In this workload, we distribute updates by selecting an object and then selecting one of the 2,000 four-byte words of the object using identical Zipf-distributions with parameter α. Using this skewed distribution allows us to model applications in which part of the state is "hot" and is frequently updated. We have found that our results remain consistent across a range of α values, so for space reasons we present all results with $\alpha = 0.5$.

We also experimented with a synthetic MMO workload. Accurately modeling an MMO is challenging as games vary widely, but we have attempted to capture the salient features using a Markov model. Each agent in the game is represented as an object and modeled with a set of five semi-independent probabilistic state machines associated with common gameplay behaviors. We tuned the transition probabilities for each state machine by looking at update rates produced for each type of action. We then adjusted the state-transition probabilities until these update rates corresponded to those we have observed in specific MMOs. For our experiments, we used 2,000 players and a total of 531,530 updates per second. Each player object contains roughly 100 KB of state corresponding to a wide variety of character attributes.

We ran all synthetic experiments on a local Intel Xeon 5500 2.66 GHz with 12 GB RAM and four cores running CentOS. Its Nehalem-based CPU has 32 KB L1 cache, 256 KB L2 cache for each core and a shared L3 cache of 8 MB. We measured disk bandwidth in this server to be roughly 60 MB/s. For the experiments with synthetic workloads, we first ensured that the checkpoint interval was large enough for any of the algorithms to finish writing the entire checkpoint to disk. Then, we normalized checkpoint interval at roughly 4 seconds, so as to provide for short estimated recovery time. Once we established this, we turned off both the Asynchronous Writer and logical logging. These mechanisms perform the same amount of work independently of checkpointing method, and disabling them enabled us to measure the synchronous overhead introduced by different algorithms more accurately.

TPC-C Application. We implemented a single-threaded transaction processing system in main memory. Following the methodology from [28], we implemented the TPC-C workload by writing stored procedures in C++. We drive this application with a memory-resident trace containing transaction procedure calls respecting the transaction mix dictated by the TPC-C specification. As in [28], we do not model think times in order to stress our implementation. As usual, we report the number of new order transactions per second.

Unlike in the synthetic benchmark experiments, we checkpoint as frequently as possible in order to understand the maximum impact of our algorithms in a realistic application. This yields the minimum possible recovery time, as the length of the log since the last checkpoint is minimized. We ran our TPC-C application on an Amazon Cluster Compute Quadruple Extra Large instance with 23 GB RAM, and computing power equivalent to two Intel Xeon X5570 quad-core Nehalem-based processors. We write checkpoints and the log to two separate RAID-0 devices, which

we configured with ten Amazon Elastic Block Storage (Amazon EBS) volumes. We observed that the aggregate bandwidth of these RAID-0 devices depended on the size of the I/O request, ranging from under 10 MB/s for small requests to over 150 MB/s for large requests exceeding 1 GB. In order to completely utilize disk bandwidth, we scale the checkpoint interval with the size of the dynamic application state, so that the Asynchronous Writer always writes data to disk as fast as possible. This differs from our synthetic experiments where we set the checkpoint interval to a constant for all measurements. Nevertheless, to fairly compare the overhead of different algorithms, we set the length of the checkpoint interval to be the same for all methods at each database size.

Since each EC2 instance communicates with EBS over the network, there is a CPU cost to writing out state in the Asynchronous Writer. To limit this effect, we set the thread affinity so that the Asynchronous Writer always runs on a separate core from the Mutator thread. As the number of cores per machine continues to increase, we believe that it will become increasingly feasible to devote a core to durability in this way. A thorough evaluation of these methods on a single core remains future work.

In addition to turning on the Asynchronous Writer, we also enable logging for these experiments. To minimize synchronous I/O effects, we configured the Logger thread to perform group commit in batches of 500 transactions. We also overlap computation with IO operations so that when the Logger is writing the actions of one batch of transactions, the Mutator is processing the next batch.

In standard OLTP systems, it is common to use the ARIES recovery algorithm [21]. As a baseline for our TPC-C experiments, we have implemented an optimized version of ARIES for main-memory databases. As FC applications do not have transactions in flight at points of consistency, checkpointing does not need to be aware of transaction aborts. This eliminates the need to maintain undo information. In addition, since the database is entirely resident in main memory, there is no need to keep a dirty page table. So in our scenario, ARIES reduces to physical redo logging with periodic fuzzy checkpoints. To optimize it as much as possible, we compressed the format of physical log records by exploiting schema information instead of recording explicit offsets and lengths whenever profitable.

In the remainder of this section, we first compare the different implementation options from Section 4 for our algorithms (Section 5.2). After selecting the alternatives with the best performance, we observe how our new algorithms compare to existing methods using both the Zipf and MMO workloads (Sections 5.3 and 5.4). Then, we report on how our methods affect application throughput in our TPC-C application (Section 5.5). Finally, we investigate the impact of a further optimization, using large pages, on the relative performance of all algorithms (Section 5.6).

5.2 Comparison of Implementation Variants

In this section we compare the performance of the different implementations of our algorithms on the Zipf workload. To get a deeper understanding of the runtime characteristics, we profiled our implementation with the Intel VTune Performance Analyzer [15]. Table 2 shows a subset of the statistics we collected from VTune. It includes cycles per instruction (CPI) as well as various measures to characterize the behavior of cache, DTLB, and page walks. For reference, we display these statistics not only for all algorithms, but also for the raw Mutator program without checkpointing.

5.2.1 Wait-Free Zigzag

Figure 3 shows the average overhead per checkpoint period of the different variants. Both IZZ and PZZ are less efficient than NZZ,

Algorithm[a]	CPI	L1D Misses / Update	L1D Miss Rate	L2 Miss Rate	DTLB Miss % (STORE)	# Page Walks / Update
NS	1.79	2.1	6.6%	10.8%	7.1%	0.50
COU	2.91	1.2	8.5%	9.2%	23.5%	0.53
BACOU	2.51	1.1	4.5%	4.5%	6.5%	0.41
NZZ-UP	13.6	0.8	22.3%	24.6%	95.2%	0.78
NZZ-NE	0.7	0.2	3.1%	3.1%	0.8%	-
IZZ-UP	11.1	0.2	7.7%	7.2%	49.2%	0.29
IZZ-NE	0.7	1.0	3.0%	3.4%	1.6%	0.02
PZZ-UP	7.1	0.3	5.0%	5.2%	48.7%	0.37
PZZ-NE	5.0	0.8	22.8%	25.4%	3.3%	0.02
BAZZ-UP	1.5	0.04	6.9%	7.2%	0.03%	-
BAZZ-NE	1.6	0.03	6.9%	8.0%	-	-
NPP	7.1	1.9	23.6%	24.0%	97.4%	0.96
IPP	2.5	1.7	7.4%	7.8%	98.3%	.85
Raw Mutator	2.4	2.0	9.4%	8.0%	96.0%	1.00

[a] -UP: update handling phase; -NE: bulk negation phase

Table 2: Profiling on synthetic workload, 320K updates/sec

except for extremely high update rates. With NZZ and PZZ, the use of long word instructions during negation brings benefits over IZZ, which negates single bytes at a time. However, the benefits are much smaller for PZZ, given that it still interleaves state information between small bit blocks. BAZZ, a variant that focuses on optimizing bulk bit-array negation, dominates all other variants.

In general, there is a tension between accelerating bulk bit-array negation and reducing per-update overhead. Table 2 shows profiling results for 320,000 updates per second. NZZ's update handling phase has an L1D cache miss rate of 22.3% and an L2 cache miss rate of 24.6%. In addition, every update incurs 0.78 page walks on average, given NZZ's independent allocation of data structures. Bulk negation, on the other hand, benefits from prefetching on these data structures. The cache miss rate is about 3% for both the L1D and the L2 cache; the DTLB miss rate is also low. IZZ trades bulk negation performance for better update performance. Its DTLB miss rate is much lower than that of NZZ. The ratio of cache misses to updates for IZZ is much higher during the bulk negation phase, however. PZZ pays more on cache misses and page walks at the update phase, but less during the bulk negation phase. Meanwhile, BAZZ focuses solely on making negation more compact, and dramatically improves performance despite a higher cache miss rate. As Figure 3 shows, this is an important effect at most update rates.

Overall, we have observed that negation is the most significant source of overhead for Wait-Free Zigzag unless the update rate is extremely high. Thus, BAZZ is the best variant for this algorithm under typical workloads. For example, at 320,000 updates per second, BAZZ exhibits about half as much overhead as NZZ and one third the overhead of IZZ.

5.2.2 Wait-Free Ping-Pong

Figure 4 shows the overhead of the two variants of Wait-Free Ping-Pong. NPP's overhead is roughly six times higher than IPP's over all update rates. Like NZZ, NPP runs into similar problems with DTLB and cache performance. Table 2 shows observation. that NPP's cycle-per-instruction ratio (CPI) is 2.8 times higher.

IPP, on the other hand, potentially incurs a cache miss on the write to the application state, but then is guaranteed to find the other words to be written on the same cache line. This has a positive effect in the L1D LOAD hit rate and eliminates most of the stalls on LOAD. IPP pays only a 43% performance overhead on top of the raw Mutator, a great improvement compared to the 258% of NPP. These numbers are very consistent across update rates.

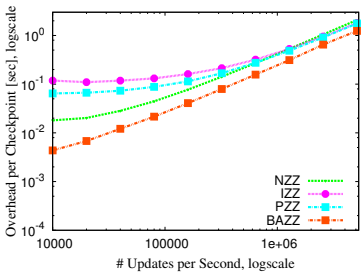


Figure 3: Wait-Free Zigzag Overhead

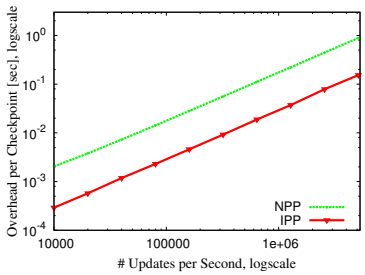


Figure 4: Wait-Free Ping-Pong Overhead

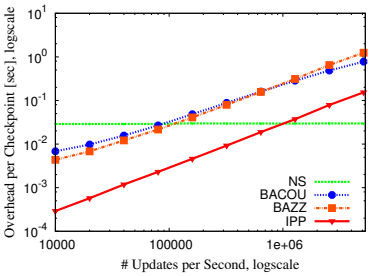


Figure 5: Zipf workload: Overhead

In short, this experiment shows that IPP comfortably dominates NPP over the whole spectrum of update rates evaluated.

5.3 Synthetic Zipf Workload

In this section, we compare the performance of our new algorithms with the best cache-aware variants of Naive Snapshot (NS) and Copy-on-Update (BACOU). We report numbers for the optimized variants described in Section 4. For both BACOU and BAZZ, the overhead numbers we report are lower bounds. Since we turned off the Asynchronous Writer, there is no lock contention between the Mutator and the Asynchronous Writer in BACOU. We also do not model reads, which discounts the small amount of per-read overhead for BAZZ. As shown below, IPP significantly dominates all these algorithms, so we do not explore these effects further.

Checkpointing Overhead. Figure 5 shows the overhead of the algorithms for update rates between 10,000 and 5,120,000 updates per second. As expected, NS is essentially constant regardless of the number of updates, since it always copies the entire state. This is the worst strategy for very low update rates since many unchanged cells get copied, but it dominates the other algorithms, with the notable exception of IPP, for more than 160,000 updates per second. This agrees with the results of [27] – when a large fraction of the words gets updated, taking a checkpoint requires copying most of the state anyway, and NS does this very efficiently.

Among the existing algorithms, BACOU is the best strategy for low update rates [27], and it is four times faster than NS for 10,000 updates per second. Its overhead increases steadily with the update rate, however, since it must lock and copy a memory block the first time the block is updated. As we increase the update rate, more blocks get updated, leading to higher locking and copying overheads. Even though updates are distributed using a Zipf distribution, we have observed only a minor effect from the fact that updates have higher likelihood to hit hot words that fall into the same memory block. This measurement corroborates prior simulation results [27]. We observe that BACOU is never the best algorithm for any update rate. It is always dominated by IPP, and it is also dominated by NS for high update rates.

IPP displays the best performance of any of the algorithms for all but the highest update rates. At 80,000 updates per second, it is nine times better than BAZZ and over an order of magnitude better than NS and BACOU. At 320,000 updates per second, the gap is still a factor of 8.4 with respect to BAZZ, 9.6 with respect to BACOU and three with respect to NS. IPP scales linearly with the number of updates over the entire range of update rates, since the predominant cost is updating each of two copies of the application state. The absolute overhead of the extra updates performed by IPP is extremely low, however, due to its cache-aware data layout and its wait-free operation. Unlike in BAZZ, in IPP the Mutator does not do any bit negations, and the beginning of a checkpoint consists only of swapping pointers.

This experiment shows that, for our default application state size of 200MB, IPP is the method with the lowest overhead for all but the highest update rates. Next, we validate that this trend is robust for a wide range of state sizes.

Scaling the State Size. To understand how our algorithms perform for applications with larger state sizes, we scale the application state from 100 MB to 1.6 GB by adding more objects to the state. Figures 6 and 7 show the overhead per checkpoint period for two different update rates. In each case, we scale the update rate with the state size, so in Figure 6, for every second, we update a number of words equal to 0.08% of the state size. This corresponds to 40,000 updates per second for 200 MB of state. In Figure 7 the update rate corresponds to 2.56%, which is 1,280,000 updates per second for 200 MB of state.

From these graphs we confirm that the trends we described above for 200 MB of application state continue to hold for larger state sizes. When the update rate is fairly low (Figure 6), IPP has roughly an order of magnitude lower overhead than NS regardless of the state size. On the other hand, when the update rate is very high (Figure 7), NS dominates all other algorithms, as it is insensitive to the number of updates. IPP continues to dominate BACOU and BAZZ for larger state sizes regardless of the update rate.

Overhead Distribution. The above experiments already show that IPP dominates BACOU and is preferable to NS for all but the highest update rates, but the overhead does not tell the whole story. As discussed in Section 2.1, we also want the overhead to be uniformly distributed over time. Figures 8 and 9 show the cost of the Mutator thread for 320,000 and 1,280,000 updates per second, respectively. Points on the *x*-axis correspond to time intervals of 0.1 sec, or alternatively the time it takes to execute 32,000 or 128,000 updates. Each point in the graphs indicates the total time taken by the Mutator thread during one such interval. The graphs thus give us an indication of how work is distributed over time.

From these graphs, we see that NS has the worst overhead distribution of any of the algorithms. At 320,000 updates per second (Figure 8) it has a latency peak of 29 ms during intervals when the state is copied. BAZZ and BACOU have much smaller peaks, at 4 ms and 6 ms, respectively. These results indicate that there is a tradeoff between the absolute overhead per checkpoint and the overhead distribution. Recall from Figure 5 that NS has lower overhead than BACOU at 320,000 updates per second, but the latter has a much lower spike. Fortunately, this tradeoff is not present for IPP, which has a nearly constant overhead of 0.8 ms per 0.1 second interval at 320,000 updates per second. This is because IPP only has to swap pointers at the beginning of each checkpoint, and most of the work is distributed evenly among the updates.

Figure 9 tells a slightly different story. The number of updates executed during each point of the graph has increased to 128,000, and this increases the baseline overhead for all of the strategies that do some work per update. BAZZ shows the most dramatic increase with a nearly constant overhead of 10 ms. BACOU and IPP also show modest overhead increases. Additionally, we can see the finer

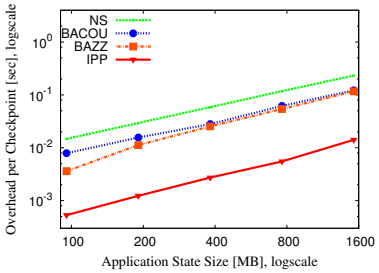


Figure 6: Scaleup: 0.08% updates/sec

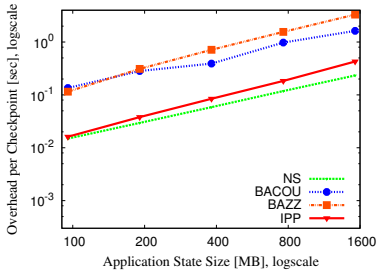


Figure 7: Scaleup: 2.56% updates/sec

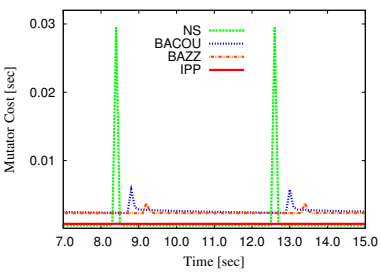


Figure 8: Latency: 320K updates/sec

structure of BACOU. There is a spike in the overhead at the end of each checkpoint period, and then the overhead gradually decreases during the checkpoint. This is because BACOU only has to copy a block the first time it is updated. As the checkpoint progresses, more blocks are already dirty and do not need to be copied.

Significantly, the behavior of NS changes very little at the higher update rate. This is because all of the work done by NS is done at the end of a checkpoint period. The small increase in the figure is due to the time necessary to apply the updates. This indicates a distinction between the overhead incurred by NS and the overhead incurred by the other algorithms. Aside from the small spikes for BAZZ and BACOU, the other algorithms incur most of their cost for work they do at each update. Thus as we increase the number of updates per unit of time, we expect their overhead to increase. Furthermore, the cost shown in each point of Figures 8 and 9 is distributed across all of the updates in the 0.1 second increment. On the other hand, the entire cost of NS occurs between two updates at the end of each checkpoint period. Thus NS is insensitive to the update rate, but it may force the system to block for a considerable amount of time during the synchronous copy. As the height of NS's latency spike is proportional to the application state size, this problem becomes worse when we scale the state size.

Overall, IPP has the most consistent overhead of any tested algorithm, and its total overhead is also lowest for all but the highest update rates. Thus we believe that it best satisfies the requirements for FC applications described in Section 2.1.

5.4 Synthetic MMO Workload

We also ran our experiments on a trace produced using our MMO workload. Figure 10 shows the checkpoint overhead. In this case, both wait-free strategies outperform BACOU. As expected, IPP performs the best, with nearly seven times less overhead than NS. BAZZ is comparable to NS, even though it had higher overhead than NS for 500,000 updates per second in the Zipf experiments. Part of the reason our new algorithms do so well in this case is that many attributes were almost never updated. About 80 percent of the attributes are only updated in response to player actions, which are human initiated and thus occur infrequently. This type of workload is bad for NS, as it has to copy many cells that are never updated.

Figure 11 shows the performance of each algorithm in the MMO simulation over time. The Mutator cost is quite variable compared to Figure 8, due to the more realistic workload, but the trends are the same. NS shows peaks of up to 29 ms, while IPP has a very low, almost uniform latency of at most 0.9 ms. These results suggest that Wait-Free Ping-Pong offers real advantages for MMO workloads.

Finally, we measured the recovery time for the MMO workload. We expected this to be the same for all of the algorithms, as they all store complete checkpoints on disk. In addition, this time is the same as for the synthetic workload, given that the application state size for both scenarios is the same. To measure this time, we observed the time to reread the checkpoint from disk after a crash simulated by server reboot, obtaining an average of 3.9 seconds out of five measurements. To simulate replaying the logical log, we can reapply the updates from the MMO trace file that occurred since the last checkpoint. The maximum time to reapply those updates is equal to our checkpointing interval of 4.2 seconds, resulting in a worst-case recovery time of only 8.1 seconds. Given that current MMO players regularly tolerate downtime [7], we think this is very reasonable for real systems.

In short, the above experiments show that IPP is the method with the lowest overhead and the best overhead distribution for a realistic application with hundreds of thousands of updates per second. In addition, IPP exhibits short recovery times in this scenario.

5.5 TPC-C Application

To validate the usefulness of our techniques in realistic FC applications with logging and checkpoint writing enabled, we compare the total overhead introduced by different checkpointing techniques in our main-memory implementation of the TPC-C benchmark [31]. We stress our implementation by processing as many transactions per second as possible and show results for the two best methods for high update rates: Naive Snapshot (NS) and Wait-Free Ping-Pong (IPP). In addition, we show the performance of our optimized version of ARIES (OPT. ARIES) as a baseline method.

Figure 12 shows throughput as we increase the number of warehouses in TPC-C. Recall that in this measurement we keep the ratio of application state size to checkpoint interval fixed. In other words, we checkpoint as fast as possible to achieve short recovery times while ensuring that the checkpoint interval is equal for all methods so as to allow for direct overhead comparison. Thus the checkpoint sizes grow and so do the costs per checkpoint. The maximum attainable performance is displayed by running the application with checkpointing disabled. Maximum throughput declines as we scale the number of warehouses given that we must operate over a larger database in main memory.

We observe that the relative performance of all methods remains roughly unchanged as we scale the number of warehouses. The variants of IPP using the Copy and Merge methods described in Section3.3 for merging the new updates with the previous checkpoint perform similarly. IPP-Copy always slightly outperforms IPP-Merge at the cost of maintaining an additional copy of the application state. Both IPP variants dominate NS, which in turn dominates OPT. ARIES. At 60 warehouses, application throughput decreases by 10.11% when using IPP-Copy, by 27.92% when using NS, and by 34.21% when using OPT. ARIES.

In order to understand the distribution of overhead in the TPC-C experiment, we also measured the response time of each of the algorithms. Figure 13 reports these results, where each point corresponds to a batch of 500 transactions that are committed together. The response time is measured as the time between the start of one batch of transactions and the start of the next. The large peaks in the response time of NS are due to the synchronous copy time at the

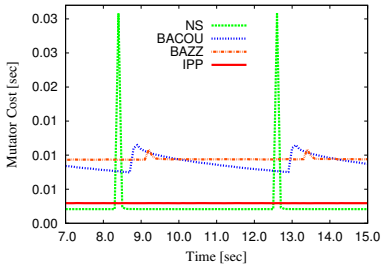


Figure 9: Latency: 1,280K updates/sec

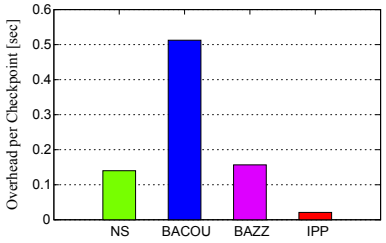


Figure 10: MMO: Overhead

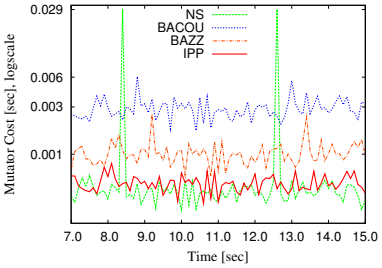


Figure 11: MMO: Latency

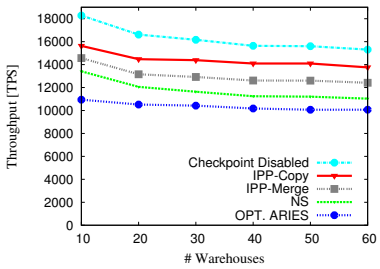


Figure 12: TPC-C Throughput

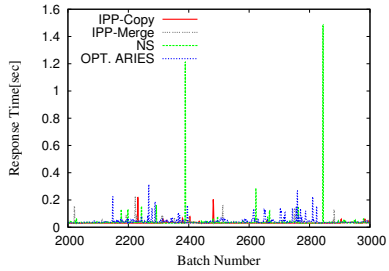


Figure 13: TPC-C Latency

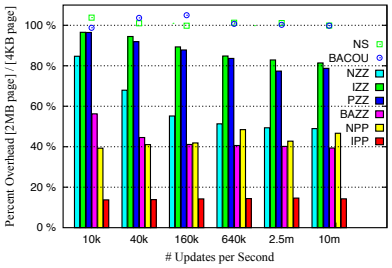


Figure 14: Large Page Overhead

end of each checkpoint period. The remaining peaks in all of the algorithms are due to the cost of logging to EBS. Recall that while the Logger is writing to EBS, the Mutator executes the next batch of transactions. This hides some of the disk latency, but the Mutator still blocks if it finishes the next batch before the Logger. Thus the response time is still quite erratic due to variability in transaction lengths and EBS latency. Since OPT. ARIES uses physical logging, it must write more data to disk, and thus the peaks for OPT. ARIES are both larger and more frequent than the other methods.

Based on these results, we find that IPP allows for frequent checkpointing with significantly lower overhead than the best existing methods. Further, it distributes overhead more evenly than either Naive Snapshot or ARIES, and thus is more suitable for latency-sensitive applications. As the memory capacity and number of cores in a single node continue to increase, we will be able to process even more warehouses within a single machine. Thus, the absolute difference in throughput between using IPP and existing methods will become even more dramatic in the future. Therefore, our experiments suggest that IPP should be the checkpointing method of choice for FC applications.

5.6 Further Optimizations: Large Pages

In order to further increase update rates and stress the limits of all methods, we investigate the effect of an additional optimization. As discussed above, page walks resulting from TLB misses are a significant bottleneck for the Mutator in most algorithms we examined. Large pages may reduce TLB misses because they cover the same region of physical memory with a smaller number of TLB entries. In our scenario, the whole application state and auxiliary data structures are implemented as a few large objects in memory, so using large pages causes very little internal fragmentation.

Figure 14 shows the reduction in overhead when we use different page sizes in the Synthetic Zipf workload. Large pages have little impact on NS and BACOU, since these two methods do not put much stress on TLB (Table 2). In constrast, all variants of Wait-Free Zigzag and Wait-Free Ping-Pong benefit noticeably from large pages. In BAZZ, using large pages yields a 40% to 60% cut in overhead. In IPP, a consistent 80% overhead cut is obtained. These two algorithms benefit the most from using large pages, and remain the best candidates from their respective family of variants. We also increased the update rates to an extremely high value to compare the overhead of IPP and NS when using large pages. Over a range up to ten million updates per second, IPP outperforms NS by up to three orders of magnitude and maintains nearly constant latency.

6. RELATED WORK

There has been extensive work in checkpointing algorithms for main-memory DBMSs [8, 23, 25, 26, 36]. Recently, Vaz Salles et al. evaluated the performance of these algorithms for MMO workloads [27]. Naive-Snapshot and Copy-on-Update came out as the most appropriate algorithms for checkpointing these FC applications. As we have seen in our experiments, Wait-Free Ping-Pong dominates those methods over a wide range of update rates. In contrast to Naive-Snapshot, Wait-Free Ping-Pong distributes overhead better over time, eliminating latency peaks. In contrast to Copy-on-Update, Wait-Free Ping-Pong completely eliminates locking overheads, as it is wait-free within checkpoint periods.

There have been different approaches to integrating checkpoint-recovery systems with applications. One consideration is whether to integrate checkpointing at the system [19, 22] or application level [4, 5]. Additionally, checkpointing may be offered to applications embedded in a language runtime [37], through a library [22], or via compiler support [5]. Our work performs application-level checkpointing and integrates with the application logic through a library API (Section 2.2). Thus we are able to checkpoint only the relevant state of the application, something not achieved by system-level checkpointing schemes. We are also able to exploit application semantics, such as frequent points of consistency, to determine when a consistent image of the state is present in main memory.

In classic relational DBMSs, ARIES is the gold standard for recovery [21]. As we have seen in Section 5.5, approaches that necessitate physical logging, such as ARIES or fuzzy checkpointing [12, 26], exhibit unacceptable logging overheads for the stream of updates produced by FC applications in main memory.

Hot standby architectures have been commonly used to provide fault tolerance on multiple database nodes [3, 14]. Recently, Lau and Madden [18] and Stonebraker et al. [28] propose implementing active standbys by keeping up to K replicas of the state. Systems using this approach can survive up to K failures, and they can

also use those replicas to speed up query processing. Our checkpointing algorithms can be used in tandem with these approaches to bulk-copy state during recovery or when the set of replicas changes. Many replicated systems (such as VoltDB [33]) also include checkpointing in order to ensure durability in the event that all replicas fail (e.g., due to a power outage).

7. CONCLUSIONS

In this paper, we have proposed two novel checkpoint recovery algorithms optimized for frequently-consistent applications. Both methods implement highly granular tracking of updates to eliminate latency spikes due to bulk state copying. Moreover, the wait-free properties of our methods within a checkpoint period allow them to benefit significantly from cache-aware data layout optimizations, dramatically reducing overhead. Wait-Free Zigzag eliminates locking overhead by keeping an untouched copy of the state during a checkpoint period. Wait-Free Ping-Pong improves both overhead and latency even more by using additional main memory space. Our thorough experimental evaluation shows that Wait-Free Ping-Pong outperforms the state of the art in terms of overhead as well as maximum latency by over an order of magnitude. In fact, given that Wait-Free Ping-Pong dominates Copy-on-Update and may have significantly lower overhead than Naive-Snapshot over a wide range of update rates, our new algorithm should be considered as an alternative wherever copy on write methods have been used in the past.

Acknowledgments. We would like to thank our shepherd, Daniel Abadi, for his detailed and insightful comments as well as the anonymous reviewers for their feedback. This material is based upon work supported by the New York State Foundation for Science, Technology, and Innovation under Agreement C050061, by the National Science Foundation under Grants 0725260 and 0534404, by the iAd Project funded by the Research Council of Norway, by the AFOSR under Award FA9550-10-1-0202, and by Microsoft. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

8. REFERENCES

[1] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, 2001.

[2] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM: Data-centric programming in the datacenter. Technical Report UCB/EECS-2009-113, EECS Department, University of California, Berkeley, 2009.

[3] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in tandem computer systems. Technical Report 86.2, PN87616, Tandem Computers, 1986.

[4] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[5] G. Bronevetsky, M. Schulz, P. Szwed, D. Marques, and K. Pingali. Application-level Checkpointing for Shared Memory Programs. In *Proc. ASPLOS*, 2004.

[6] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM TOCS*, 3(1):63–75, 1985.

[7] R. Cortez. World Class Networking Infrastructure. In *Proc. Austin GDC*, 2007.

[8] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. SIGMOD*, 1984.

[9] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[10] H. F. Guðjónsson. The Server Technology of EVE Online: How to Cope With 300,000 Players on One Server. In *Proc. Austin GDC*, 2008.

[11] N. Gupta, A. J. Demers, J. Gehrke, P. Unterbrunner, and W. M. White. Scalability for virtual worlds. In *ICDE*, 2009.

[12] R. Hagmann. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Transactions on Computers*, 35(9):839–843, 1986.

[13] M. Herlihy. Wait-free Synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[14] S.-O. Hvasshovd, O. Torbjornsen, S. Bratsberg, and P. Holager. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In *Proc. VLDB*, 1995.

[15] Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune.

[16] Jason Gregory. *Game Engine Architecture (Section 7.5)*. A K Peters, 2009.

[17] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proc. SIGMOD*, 2010.

[18] E. Lau and S. Madden. An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse. In *Proc. VLDB*, 2006.

[19] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System. Technical Report 1346, University of Winsconsin-Madison, 1997.

[20] T. MacDonald. Solid-state Storage Not Just a Flash in the Pan. *Storage Magazine*, 2007. http://searchStorage.techtarget.com/magazineFeature/0,296894,sid5_gci1276095,00.html .

[21] 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 TODS*, 17(1):94–162, 1992.

[22] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under UNIX. In *Proc. USENIX Winter Technical Conference*, 1995.

[23] C. Pu. On-the-Fly, Incremental, Consistent Reading of Entire Databases. *Algorithmica*, 1:271–287, 1986.

[24] RamSan-400 Specifications. http://www.ramsan.com/products/ramsan-400.htm .

[25] D. Rosenkrantz. Dynamic Database Dumping. In *Proc. SIGMOD*, 1978.

[26] K. Salem and H. Garcia-Molina. Checkpointing Memory-Resident Databases. In *Proc. ICDE*, 1989.

[27] M. V. Salles, T. Cao, B. Sowell, A. Demers, J. Gehrke, C. Koch, and W. White. An evaluation of checkpoint recovery for massively multiplayer online games. In *Proc. VLDB*, 2009.

[28] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proc. VLDB*, 2007.

[29] T. Strohman and W. B. Croft. Efficient Document Retrieval in Main Memory. In *Proc. SIGIR*, 2007.

[30] A. Thomson and D. Abadi. The case for determinism in database systems. In *Proc. VLDB*, 2010.

[31] Transaction Processing Council. TPC Benchmark(TM) C, 2010. http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[32] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

[33] VoltDB. http://voltdb.com/product.

[34] G. Wang, M. V. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White. Behavioral simulations in mapreduce. In *Proc. VLDB*, 2010.

[35] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling Games to Epic Proportions. In *Proc. SIGMOD*, 2007.

[36] A. Whitney, D. Shasha, and S. Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL, or C++. In *Proc. HPTS*, 1997.

[37] G. Zheng, L. Shi, and L. V. Kale. FTC-Charm++: an In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *Proc. CLUSTER*, 2004.