An Evaluation of Checkpoint Recovery for Massively Multiplayer Online Games

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

ABSTRACT

Massively multiplayer online games (MMOs) have emerged as an exciting new class of applications for database technology. MMOs simulate long-lived, interactive virtual worlds, which proceed by applying updates in frames or ticks, typically at 30 or 60 Hz. In order to sustain the resulting high update rates of such games, game state is kept entirely in main memory by the game servers. Nevertheless, durability in MMOs is usually achieved by a standard DBMS implementing ARIES-style recovery. This architecture limits its scalability, forcing MMO developers to either invest in high-end hardware or to over-partition their virtual worlds.

In this paper, we evaluate the applicability of existing checkpoint recovery techniques developed for main-memory DBMS to MMO workloads. Our thorough experimental evaluation uses a detailed simulation model fed with update traces generated synthetically and from a prototype game server. Based on our results, we recommend MMO developers to adopt a copy-on-update scheme with a double-backup disk organization to checkpoint game state. This scheme outperforms alternatives in terms of the latency introduced in the game as well the time necessary to recover after a crash.

1. INTRODUCTION

Massively multiplayer online games (MMOs) are persistent virtual worlds that allow tens of thousands of users to interact in fictional settings [11, 26]. Users typically select a virtual avatar and collaborate with other users to solve puzzles or complete quests. These games are extremely popular, and successful MMOs, such as World of Warcraft, have millions of subscribers and have generated billions of dollars in revenue [3]. Unlike single player computer games, MMOs must persist across user sessions. Players can leave the game at any time, and they expect their achievements to be reflected in the world when they rejoin. Similarly, it is unacceptable for the game to lose player data in the event of a crash. These demands make it essential for MMOs to ensure that their state is durable.

In order to provide durability, MMO developers have turned to database technology. While specific MMO architectures differ significantly, many have adopted a three-tiered architecture to build their virtual worlds. Clients communicate with game servers to update the state of the world, and these servers use a standard DBMS back-end to provide transactional guarantees. This architecture is appropriate for state updates that require full ACID guarantees. For example, many MMOs allow players to trade or sell in-game items, sometimes for real money [35].

In many MMOs, a much larger class of updates does not require transactional guarantees. For example, character movement is handled within the event simulation, and conflicts are resolved using game specific logic such as collision detection. Though these updates do not require transactional guarantees, they occur at a very high rate. For instance, characters move at close to graphics frame rate, and this can lead to hundreds-of-thousands or millions of updates per second. Since most state-of-the-art DBMS use ARIES-style recovery algorithms [21], their update rate is limited by the logging bandwidth, and they are unable to support the extremely high rate of game updates.

To provide some degree of durability, developers typically resort to ad-hoc solutions that rely on expensive specialized hardware, or partitioning schemes that limit the number of characters that can interact at once [5, 11, 22]. For instance, developers often create *shards*, which are disjoint instances of the same game. Though these solutions allow games to scale, they limit the user experience by preventing large numbers of players from interacting.

In this paper we experimentally evaluate a number of traditional main-memory checkpoint recovery strategies for use with MMOs [18, 29, 39]. These strategies advocate using logical logging to reduce disk activity and periodically creating consistent checkpoints of the main-memory state. These techniques are particularly attractive for MMOs, where a single logical action (e.g., a user command) may generate many physical updates. In the event of a crash, the game state can be reconstructed by reading the most recent checkpoint and replaying the logical log.

The real-time nature of MMOs introduces some unique considerations when choosing a checkpointing algorithm. First of all, latency becomes a critical measure. Whereas most traditional recovery work has emphasized throughput, MMOs must execute at frame rate so that users have the experience of a fluid and immersive game. Thus the entire checkpointing process must fit into the game simulation without causing any visible "hiccups" in the game. To achieve this, we can exploit the fact that game updates are applied in a simulation loop. To ensure that all clients see a consistent world, there can be no outstanding updates at the end of an iteration of the simulation loop. This allows us to avoid many of the traditional strategies for aborting ongoing transactions.

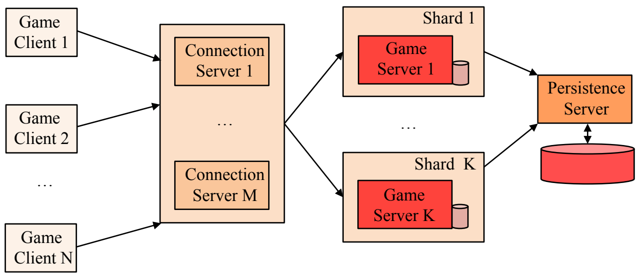


Figure 1: Architecture of a typical MMO.

To our knowledge, this is the first paper to evaluate the applicability of existing checkpoint recovery techniques for main-memory DBMS to MMO workloads. We make two primary contributions. First, we show how to adapt consistent checkpointing techniques developed for main-memory databases to MMOs. In particular, we discuss the interplay of these techniques with the stringent latency requirements of MMOs. Second, we provide a thorough simulation model and evaluation of six recovery strategies. We characterize their performance in terms of latency, checkpoint time, and recovery time. In addition to synthetic data, we test our techniques on a prototype MMO that simulates medieval combat of the type found in many popular games. We also validate the results produced by our simulation model against a real implementation of a relevant subset of the recovery strategies. The Java source code for our simulation is available for download [8].

The remainder of the paper is organized as follows. In Section 2, we present the MMO architecture considered in this paper. In Section 3, we review the set of techniques from main-memory database checkpoint recovery evaluated in this paper. We then describe in detail our implementation, simulation model, and experimental setup in Section 4. Section 5 presents the results of our evaluation of checkpoint recovery techniques. In Section 6, we validate our simulation model against a real implementation of a subset of the recovery techniques. We review related work in Section 7 and conclude with recommendations for MMO developers based on our results in Section 8.

2. ARCHITECTURE OF AN MMO

In this section, we describe the architecture of an MMO [11, 26, 36, 38]. Most MMOs use a client-server architecture, as shown in Figure 1. Clients join the virtual world through a *connection server* that connects them to a single *shard*. Shards are independent versions of the virtual world aimed at improving scalability. Shards are not synchronized, and players on one shard cannot interact with players on another. More recent MMOs such as Eve Online [11] and Taikodom [12] are pioneering *shardless* architectures in which all players interact in the same virtual world.

In this paper we focus on recovery for a single shard, though our techniques could be extended to shardless architectures. We also make the simplifying assumption that a shard consists of a single logical machine. This machine is the *game server* in Figure 1. This may be a large mainframe as used in Taikodom [12] or a cluster with tightly synchronized clocks. This work is a first step towards a more general solution for sophisticated network topologies.

2.1 The Game Logic

The core of an MMO is a discrete-event simulation loop that executes at close to the graphics frame rate, typically 30 or 60 Hz. This provides the illusion of an interactive and continuously evolving virtual world. Conceptually, the state of the MMO is a table containing game objects, including characters and the items they interact with. In order to meet the stringent real-time demands placed on the game, the active state of the game is typically kept in the main memory of the simulation server. Disks are used only to provide persistence or to store information about inactive (e.g., logged off) characters.

During each iteration or *tick* of the simulation loop, portions of the state are updated according to game-specific logic. These updates may be triggered by user actions, elapsed time, or some other event. Note that a single user action may cause several physical updates to the state. For instance, a user-level movement command may translate into a number of smaller movement updates that occur over several ticks.

We make no assumptions about the structure of updates during a tick. Developers may use multithreading to process updates in parallel or leverage a special purpose language like SGL to convert the computation into a relational query plan [37]. The only guarantee is that all updates finish by the end of the tick. The state must be consistent at the end of every tick, and any actions that last longer than a tick must be represented in the persistent state.

2.2 Transactions and Durability in MMOs

Current MMOs focus on providing transactional guarantees for a small subset of updates that need to be globally consistent across servers or communicate with external services. For example, EVE Online processes nine million item insertions per day that require transactional behavior [11]. Other MMOs may include financial transactions that require ACID properties. These transactions frequently involve user interaction or communication with an external system, and thus the update rate is fairly low. Recovery can therefore be handled by a standard DBMS with an ARIES-style recovery manager. This system is the *persistence server* in Figure 1.

In addition to proper transactions, however, MMOs also include a large number of *local updates* that change the game state but do not require complete transactional behavior. For instance, character movement is the single largest source of updates in most MMOs [6, 26], but game-specific logic ensures that these updates never conflict. Nevertheless, we would like to ensure that local updates are durable so that players do not lose their progress in the game in the event of a server failure.

A modern MMO may see hundreds of thousands of local updates per tick, and processing these efficiently is a major challenge [34]. Traditional DBMS cannot sustain this update rate, and existing MMOs have either created ad-hoc checkpointing strategies or resorted to costly special purpose hardware [23]. For instance, EVE Online uses RAM-based SSD drives for their database servers to handle logging and transaction processing [11].[1] While well established MMOs are able to afford such hardware investments, these investments are usually delayed until the MMO builds up a substantial user base. In addition, the cost of any specialized hardware must be multiplied by the number of game server shards. Some MMOs have several hundred shards and this increases their hardware costs accordingly.

2.3 Recovery Requirements for MMOs

We consider a principled approach to durability using a checkpoint recovery approach first proposed for main-memory databases [18, 39, 29] and scientific simulations [30]. The applicability of these approaches to MMOs has never been fully explored, and our investigation focuses on the unique workload requirements found in online games.

[1]These drives, which start at $90,000, have a sustained *random* I/O rate of up to 3GB/s [25, 20]

For example, since MMOs are interactive, it is important for simulation ticks to occur at a uniform a rate: an inconsistent tick rate can distort time in the virtual world, and be frustrating to players [4]. In a choice between uniformity and performance, online game designers typically chose uniformity, especially if performance can be masked by client-side interpolation (e.g. the client interpolates finer-grained values from coarse server states to produce smoother animation). Scientific simulations, on the other hand, tend to be non-interactive [27]. For these applications, individual tick time is unimportant, and total running time is the most important performance measurement. Hence, a high-throughput recovery algorithm designed for scientific simulations may not be ideal for MMOs if it suffers from occasional bouts of poor latency.

Another interesting feature of MMOs is that players are often willing to accept longer recovery times so long as their progress in the game is not lost. These games are designed so that the players can easily take breaks and leave the game without it affecting the experience significantly. The primary problem with server downtime is that it dictates when the player must take a break; as such, it can be an annoyance, but it is accepted as long as it does not occur too frequently and does not last more than several minutes. On the other hand, losing game state is particularly undesirable; if a player has spent a lot of effort trying to beat a difficult challenge, then she might quit in frustration if a server crash forces her to repeat the challenge again. If the challenge is well-defined, like killing a powerful monster, then the player's progress can be preserved by ARIES-style recovery algorithms. However, this can only encompass those challenges that the game designer was able to identify ahead of time. Some challenges, such as jumping up onto a specific hard-to-reach platform, may be overlooked in the original design, and thus do not have transactional guarantees.

Our investigation considers these requirements in analyzing checkpointing algorithms for MMOs, enabling us to better understand the engineering trade-offs involved in implementing recovery algorithms for online games.

3. MAIN-MEMORY DBMS RECOVERY

In this section, we review the main-memory recovery techniques that we evaluate in this paper.

3.1 Checkpointing for MMOs

Before enumerating the algorithms we evaluate, we consider the implications of the architecture described in Section 2 on the design of a recovery solution for MMOs. First, MMOs are amenable to disk-based recovery solutions. Though high-availability techniques that rely on hot standbys may provide faster recovery times, they require hardware over-provisioning and thus are expensive [30]. Furthermore, MMOs can tolerate some downtime. Developers argue that 99.99% uptime is sufficient for their systems, equivalent to about one hour of unplanned downtime per year [36]. At the failure rates observed for current server hardware, there is more than adequate room for a checkpoint recovery solution to deal with fail-stop failures [30].

A checkpoint recovery solution for MMOs must resort to logical logging in order to record state updates. As discussed previously, the rate of local updates may be extremely large, and physically logging this stream could easily exhaust the available disk bandwidth. Instead, we log all user actions at each tick and replay the ticks to recover. This allows us to recover to the precise tick at which a failure occurred. Though logical logging considerably reduces the number of updates written to disk, it adds the overhead of replaying the simulation to the recovery time. To minimize this cost and fully utilize the disk bandwidth, we would like to take checkpoints as frequently as possible.

In order to use logical logging, we must restrict our attention to consistent checkpointing techniques. As it turns out, the structure of the discrete-event simulation loop provides a natural point to do checkpointing. Since all updates complete before the end of the tick, we do not have to worry about aborting ongoing updates if we take checkpoints at tick boundaries.

3.2 Consistent Checkpointing Techniques

We consider several consistent checkpointing algorithms that have been developed in the context of main-memory DBMS. All of these strategies write to stable storage asynchronously, but their memory behavior may be characterized along three dimensions:

In-memory copy timing: some algorithms perform an *eager copy* of the state to be checkpointed in main memory, while others perform *copy on update*. Algorithms that perform eager copies are conceptually simpler, but they introduce pauses in the discrete-event simulation loop of the game.

Objects copied: some algorithms copy only *dirty objects*, while others copy *all objects* in the game state. The amount of state included in a copy will affect the checkpoint time, and as a consequence, the time required to replay the ticks during recovery.

Data organization on disk: some algorithms use a simple *log* file, while others use a *double-backup* organization as proposed by Salem and Garcia-Molina [29]. Depending on the disk organization, we may be able to capitalize on sequential I/O when flushing the checkpoint to disk.

Table 1 shows how existing algorithms fit into this design space. We describe the algorithms in detail below.

Naive-Snapshot. The simplest consistent checkpointing technique is to quiesce the system at the end of a tick and eagerly create a consistent copy of the state in main memory. We then write the state to stable storage asynchronously. Naive-Snapshot can use either a double backup or log-based disk structure. We use the former in our experiments. This strategy is very simple, and a number of real systems have used it [2, 19, 30, 39]. Other than ease of implementation, one advantage of this technique is that it can be implemented at the system level without knowledge of the specific application. On the other hand, copying the full state may introduce significant pauses in the game. As games are extremely sensitive to latency [4], such long pauses may make this algorithm inapplicable.

Dribble-and-Copy-on-Update. This algorithm takes a checkpoint of the full game state without quiescing the system for an eager copy [28]. Each game object has an associated bit that indicates whether it has been flushed to the checkpoint or not. An asynchronous process iterates (or "dribbles") through each object in the game and flushes the object to the checkpoint if its bit is not set. The process sets the bit of any object it flushes. Additionally, when an object whose bit is not set is updated, the object is copied and its bit is set. If an object's bit is already set, it is not copied again until the next checkpoint. This method produces a checkpoint that is consistent as of the time the copy was started. Note that it is not necessary to reset all of the bits before starting the next checkpoint. Instead, we can simply invert the interpretation of the bit attached to each object [24]. In this strategy each object is copied exactly once per checkpoint, regardless of how many times it is updated. This is a critical property for handling the update-heavy workloads of MMOs. Another advantage of this technique is that there is no need to quiesce the system for an eager copy. A disadvantage of this technique is that the decision to include the entire game state in every checkpoint may lead to high main-memory copy overhead and long checkpoint times.

Atomic-Copy-Dirty-Objects. This algorithm refines Naive-

Objects Copied	Eager Copy		Copy on Update	
	Double Backup	Log	Double Backup	Log
All Objects	Naive Snapshot [2, 19, 30, 39]		Dribble and Copy on Update [24, 28]	
Dirty Objects	Atomic Copy Dirty Objects [29]	Partial Redo [29]	Copy on Update [7, 29]	Copy on Update Partial Redo [7, 29]

Table 1: Algorithms For Checkpointing Game State

Snapshot by copying only the "dirty" state that has changed since the last checkpoint. State that was not modified can be recovered from the previous checkpoints. Though previous work has proposed methods to copy this state without quiescence, they rely on aborting transactions that read inconsistent state [24, 29]. To avoid this overhead, we perform our copies eagerly during the natural period of quiescence at the end of each tick. We follow Salem and Garcia-Molina and organize our checkpoints in a double-backup structure on disk [29]. In this technique, two copies of the state are kept on disk and objects in main memory have two bits associated with them, one for each backup. Each bit indicates whether the associated object has already been sent to the corresponding backup on disk. Checkpoints alternate between the two backups to ensure that at all times there is at least one consistent image on the disk. Each object has a well-defined location in the disk-resident checkpoint, allowing us to update objects in it directly. As one optimization to avoid arbitrary random writes, we write the dirty objects to the double backup in order of their offsets on disk. Note that even if an in-memory sort operation is performed for that purpose, it can be done asynchronously and its time is negligible compared to the time spent writing the objects to disk. This sorted I/O optimization is crucial for algorithms that use a double-backup organization, but not necessary for log-based algorithms. The latter methods already write to disk sequentially, and thus as fast as possible. This algorithm will be an improvement over Naive-Snapshot if the dirty object set is generally much smaller than the entire game state.

Partial-Redo. As in Atomic-Copy-Dirty-Objects, Partial-Redo performs an eager copy in main memory of the objects dirtied since the last checkpoint. As a consequence, it also has latency as a potential disadvantage. On the other hand, Partial-Redo attempts to improve the writing of checkpoints with respect to Atomic-Copy-Dirty-Objects. One disadvantage of the double-backup organization is that it may not write dirty objects contiguously on disk. To address this problem, Partial-Redo writes dirty objects to a simple log [9]. Note that while the log organization allows us to use a sequential write pattern, we may have to read more of the log in order to find all objects necessary to reconstruct a full consistent checkpoint. In order to avoid this overhead, we periodically create a full checkpoint of the state using Dribble-and-Copy-on-Update.

Copy-on-Update. We can also refine Dribble-and-Copy-on-Update to copy only dirty objects [7, 29]. In this algorithm the in-memory copies are performed on update, and an object is copied only when it is first updated. We use a double-backup structure on disk as in Atomic-Copy-Dirty-Objects. A similar method was used in SIREN, where the authors found that it may be beneficial to copy at the tuple level rather than the page level [18]. In our experiments, all copies are performed at the level of atomic objects whose size may be configured (Section 4.1). An advantage of this algorithm is that it has a smaller effect on latency since it does not perform an eager copy. In addition, it may have smaller memory requirements because additional main memory is allocated only when an update touches an object that is being flushed to the checkpoint. A potential disadvantage is that the double-backup organization does not allow fully sequential I/O.

Copy-on-Update-Partial-Redo. This algorithm is similar to Copy-on-Update, but uses a log-based disk organization to transform sorted writes into sequential writes. As with Partial-Redo, we periodically run Dribble-and-Copy-on-Update to limit the portion of the log that we must access during recovery. Combining copy on update with logging, we aim to obtain the advantages of both Copy-on-Update and Partial-Redo.

4. EXPERIMENTAL SETUP

In this section, we describe the experimental setup used in our evaluation. We have implemented a detailed simulation to measure the cost of checkpointing and recovery. A simulation model allows us to understand all relevant performance parameters of the algorithms of Section 3. Using the simulation model has three advantages. First, we can use hardware parameters that would be found in server configurations common in MMOs even though we might not own the corresponding hardware. Second, it dramatically reduces the effort necessary to implement all the algorithms described previously. Third, it enables other researchers or MMO developers to easily repeat our results, modify parameters, and understand their effects on the algorithms. In order to fully achieve the latter benefits, we have made our simulation available for download [8]. All of our code is written in Java 1.6.

In the following, we discuss in detail our implementation of the checkpointing algorithms (Section 4.1) and our simulation model (Section 4.2). In addition, we describe the simulation parameters (Section 4.3) and the datasets (Section 4.4) used in our evaluation.

4.1 Implementation

Our implementation is organized into two main parts. The first part is an algorithmic framework that isolates the main costs of the checkpointing algorithms in subroutines. The second part comprises specific implementations of these subroutines for each algorithm, done based on the cost model of our simulator. We describe these two components of our implementation below.

Checkpointing Algorithmic Framework. As discussed in Section 2, at the end of each tick, the state of the game in the discrete simulation loop is consistent in main memory. The core idea of our algorithmic framework is to capture this tick-consistent image of the game state and checkpoint it to disk. We assume in our presentation that updates are handled at the level of abstract game objects, which we call *atomic objects*. In principle, an atomic object can be as small as a single attribute of a row in the game state. However, use of an atomic object size smaller than a disk sector adds significant overheads, especially for double-backup schemes. Techniques to improve I/O by organizing objects into logical pages are orthogonal to our description [18]. We assume they have been applied in order to make the atomic object size equal to a disk sector.

The Checkpointing Algorithmic Framework shows the general method we follow in our implementation. At the end of each game tick, the algorithm checks whether we have finished taking the last checkpoint. If so, a new checkpoint is started. First, we syn-

Algorithm	Subroutine			
	Copy-To-Memory	**Write-Copies-To-Stable-Storage**	**Handle-Update**	**Write-Objects-To-Stable-Storage**
Naive-Snapshot	All objects	All objects, log	No-op	No-op
Dribble-and-Copy-on-Update	No-op	No-op	First touched, all	All objects, log
Atomic-Copy-Dirty-Objects	Dirty objects	Dirty objects, double backup	No-op	No-op
Partial-Redo	Dirty objects	Dirty objects, log	No-op	No-op
Copy-on-Update	No-op	No-op	First touched, dirty	Dirty objects, double backup
Copy-on-Update-Partial-Redo	No-op	No-op	First touched, dirty	Dirty objects, log

Table 2: Subroutine Implementations for Checkpoint Recovery Algorithms

Checkpointing Algorithmic Framework

```
Input: ObjectSet $O_{all}$ containing all objects in game data
1  do synchronous on end of game tick :
2      if last checkpoint finished then
3          //perform synchronous copy actions at end of tick:
4          ObjectSet $O_{copy} \leftarrow$ Copy-To-Memory($O_{sync} \subseteq O_{all}$)
5          if $O_{copy} \neq \emptyset$ then
6              do asynchronous
                   Write-Copies-To-Stable-Storage($O_{copy}$)
7          end
8          // register synchronous handler for update events:
9          do synchronous on each Update u of an Object $o \in O_{all}$ :
10             Handle-Update($u$, $o$)
11         end
12         // perform asynchronous copy of remaining information:
13         do asynchronous
               Write-Objects-To-Stable-Storage($O_{all} \setminus O_{sync}$)
14     end
15 end
```

chronously copy in-memory part (or all) of the game state (Copy-To-Memory subroutine). This portion of the state is then written to the appropriate data structures in stable storage on the background (Write-Copies-To-Stable-Storage subroutine). For the remainder of the game state, we start an asynchronous copy to stable storage (Write-Objects-To-Stable-Storage subroutine) and register a handler routine that reacts to updates while this copy operation is running (Handle-Update subroutine).

Instantiating algorithms. We implement all algorithms of Section 3 by providing appropriate implementations of the subroutines in the Checkpointing Algorithmic Framework. We summarize in Table 2 how each subroutine is implemented for each checkpointing algorithm. We state whether the routine acts on all objects of the game state or only on dirty objects and also whether the routine is implemented or is a no-op. When different from a no-op, the subroutines perform the operations below:

1. **Copy-To-Memory**: this subroutine computes the time necessary to copy either all objects or dirty objects to main memory and advances the simulation time.

2. **Write-Copies-To-Stable-Storage**: this subroutine computes, at every tick, the amount of data from the memory copy that has been flushed to disk. It takes into account whether we are writing to a sequential log or to a double backup. This routine operates on state copied by the Copy-To-Memory routine and thus may be implemented without thread-safety concerns.

3. **Handle-Update**: this subroutine computes the time to perform bit tests, acquire locks, and save the old value of an item in memory. Since we save the old value, this routine is only executed the first time we update an item.

4. **Write-Objects-To-Stable-Storage**: this subroutine computes, at every tick, the amount of data from the objects to be checkpointed that has been flushed to disk. As with the Write-Copies-To-Stable-Storage routine, the amount flushed depends on whether we are writing to a log or to a double backup. Unlike Write-Copies-To-Stable-Storage, this routine reads the actual game state concurrently and thus must be thread-safe.

It is important to emphasize that our simulation does not perform any actual I/O operations or memory copies. Rather, we keep track of which objects have been updated since the last checkpoint and compute the time necessary for these operations based on the detailed simulation model presented in the next section.

4.2 Simulation Model

We describe the main components of the performance model we have used in our simulation below. We assume that the game server is a dedicated machine and, therefore, there is no resource contention with other workloads. MMOs are cognizant of issues that influence performance and avoid running processes that might negatively affect game experience. Scheduled maintenance activities, for example, are typically performed at well-defined times during off-peak hours.

Assume n is the number of atomic objects in the game state.

Duration of synchronous copy. The function Copy-To-Memory in the Checkpointing Algorithmic Framework performs a synchronous, in-memory copy, which introduces a pause in the simulation loop. Given k contiguous atomic objects of size S_{obj} to be included in the synchronous copy and memory bandwidth B_{mem}, we can calculate the time $\Delta T_{sync}(k)$ required to copy these objects by:

$$\Delta T_{sync}(k) \;=\; O_{mem} + \frac{k \cdot S_{obj}}{B_{mem}}$$

Here the constant term O_{mem} represents memory copy startup overhead, including possible cache misses. The total pause time is computed by summing this formula over all contiguous groups of atomic objects to be copied.

Because latency affects the gaming experience, there is a (game-dependent) hard bound on the maximum duration of a pause [4]. Thus it is important to consider the latency introduced in the discrete-event simulation loop in order to understand the applicability of a checkpoint recovery technique.

Duration of asynchronous write. The functions Write-Copies-To-Stable-Storage and Write-Objects-To-Stable-Storage perform asynchronous copies of atomic objects. If we write $k \leq n$ contiguous objects to a log-based organization on disk, I/O is sequential and we can take maximum advantage of the disk bandwidth B_{disk}. Thus, the duration function $\Delta T_{async}(k)$ of the asynchronous copy will be:

$$\Delta T_{async}(k) \;=\; \frac{k \cdot S_{obj}}{B_{disk}} \quad \text{(log-based)}$$

This approximation does not consider the initial seek and rotational delay, but for realistic game state sizes the error introduced is negligible.

For algorithms using a double-backup disk organization we assume a sorted write I/O pattern, in which disk sectors are transferred in order of increasing offset in a contiguously allocated backup file. We assume for simplicity that the objects to be written are uniformly spaced in the file. If more than a tiny fraction of the sectors are written, with high probability, there will be a dirty atomic object to be written in every track of the file. As a consequence, this pattern requires a full rotation for every track of the file, so the cost of writing k sectors can be approximated well by the cost of a full transfer,

$$\Delta T_{async}(k) \approx \frac{n \cdot S_{obj}}{B_{disk}} \quad \text{(double-backup)}$$

As above, this approximation does not consider the initial seek and rotational delay; it also ignores the fact that the expected distance from the first to the last sector transferred is actually shorter than n by $O(n/k)$. Again, for the values of n and k we consider, the error introduced is negligible. This model has the slightly counter-intuitive (but correct) property that, for reasonable values of k, the amount of data written to the backup file is proportional to k, but the elapsed time to write that data is *independent* of k.

Effect of copy on update. All algorithms on the right of Table 1 use a copy-on-update scheme, in which synchronous in-memory copy operations may be performed during checkpointing in order to save old values of game objects being updated. These copy operations add overhead to the simulation loop, potentially over multiple ticks. To estimate this overhead, we examine each atomic object update operation and compute an overhead cost of the form

$$\Delta T_{overhead} = O_{bit} + O_{lock} + \Delta T_{sync}(1)$$

The first term in this expression represents the overhead for the simulation loop to test a per-atomic-object dirty bit. If that test fails, the second term is added, representing the cost to lock out the asynchronous copy thread. The third term is added if necessary to represent the cost of a memory copy of a single atomic object. This detailed simulation is needed because dirty bit maintenance, while comparatively cheap, is embedded in the inner loop of the simulator, so the overhead it introduces can be quite significant and must be modeled.

Recovery time. The recovery time $\Delta T_{recovery}$ of our methods is divided into two components: the time to read a checkpoint, $\Delta T_{restore}$, and the time to replay the simulation after the checkpoint is restored, ΔT_{replay}. It is given by

$$\Delta T_{recovery} = \Delta T_{restore} + \Delta T_{replay}$$

For all schemes except Partial-Redo and Copy-on-Update-Partial-Redo, we have:

$$\Delta T_{restore} = \frac{n \cdot S_{obj}}{B_{disk}}$$

For Partial-Redo and Copy-on-Update-Partial-Redo, the time to restore will be larger whenever updates are concentrated into a subset of the game objects. In the worst case, we will have to read the log backwards until we find the last time all objects have been flushed to the log. Assuming that k objects are written to the log per checkpoint and that we perform a full write of the n game objects every C checkpoints, we may estimate the time to restore by:

$$\Delta T_{restore} = \frac{(k \cdot C + n) \cdot S_{obj}}{B_{disk}}$$

The time to replay ΔT_{replay} is, in the worst case, equal to the time to checkpoint $\Delta T_{checkpoint}$. This situation may happen if the system crashes right before a new checkpoint is finished. In that case, we restore the previous checkpoint and expect the simulation to take exactly the same time between checkpoints to redo its work. Note that the time to checkpoint will be given either by ΔT_{async} or $\Delta T_{sync} + \Delta T_{async}$, depending on the method used.

4.3 Simulation Parameters

parameter	notation	setting
Tick Frequency	F_{tick}	30 Hz
Atomic Object Size	S_{obj}	512 bytes
Memory Bandwidth	B_{mem}	2.2 GB/s
Memory Latency	O_{mem}	100 ns
Lock overhead	O_{lock}	145 ns
Bit test/set overhead	O_{bit}	2 ns
Disk Bandwidth	B_{disk}	60 MB/s

Table 3: Parameters for cost estimation

As described in Section 4.2, we explicitly model the hardware parameters and compute all relevant costs for the algorithms. The parameters we have used in our evaluation are given in Table 3. During each tick, the simulator selects a set of objects to update and then invokes the appropriate checkpointing strategy. We update at the granularity of an atomic object, and we allow an object to be updated more than once per tick. Since we simulate all relevant hardware parameters, our hardware setup is unimportant.

Note that the parameters in Table 3 fall into several different categories. Some parameters, such as F_{tick}, are properties of the game being simulated. Others, such as S_{obj}, are parameters of the algorithms. The most challenging parameters are those representing system performance. The values presented here were measured for one particular server in our lab, using a collection of microbenchmarks written for the purpose.

Memory bandwidth: We measured effective memory bandwidth B_{mem} by repeated memcpy calls using aligned data, each call copying an order of magnitude more data than the size of the L2 cache of the machine.

Memory latency: We measured memory latency O_{mem} using another memcpy benchmark with memory reference patterns mixing sequential and random access. The intent was to take into account both hardware cache-miss latency and the startup cost of the memcpy implementation.

Lock overhead: We measured lock overhead O_{lock} as the aggregate cost of uncontested calls to pthread_spinlock, again with a mixture of sequential and random access patterns.

Bit test overhead: The bit test overhead O_{bit} is intended to model the cost of the dirty-bit check that must be added to the simulation loop for the copy-on-update variants of the algorithms. This benchmark measures the incremental cost of naive code to count dirty bits, roughly half of which are set. The code is added to a loop intended to model the memory reference behavior of the update phase of the game simulation loop.

Disk bandwidth: We measured the effective disk bandwidth B_{disk} by performing large sequential writes to a block device allocated to our recovery disk.

All these benchmarks required considerable care to achieve repeatable results and to eliminate errors due to loop overheads, compiler optimizations, and the cost of timing calls themselves.

4.4 Datasets

The input to our simulator is an update trace indicating which attributes of game objects, termed cells, have been updated on each tick of the game. We consider several methods for choosing which cells to update. In the first set of experiments, we generate updates according to a Zipf distribution with parameter α. We choose the

parameter	setting
number of ticks	1,000
number of table cells	10,000,000
number of updates per tick	1,000 ... **64,000** ... 256,000
skew of update distribution	0 ... **0.8** ... 0.99

Table 4: Parameter settings used in the Zipfian-generated update traces

parameter	setting
number of units	400,128
number of attributes per unit	13
number of ticks	1,000
avg. number of updates per tick	35,590

Table 5: Characteristics of the update trace from our prototype game server.

row and column to update independently with the same distribution. By varying α, we can control the skew of the distribution. As α grows, we are more likely to update a small number of "hot" objects. Our Zipfian generator is from [10]. In addition to the update skew, we also vary the number of updates per tick and analyze latency peaks introduced by the different methods. Table 4 shows the range of parameter values we used. The numbers in bold are the default values we used when varying the other variables.

For each of these three independent variables, we measure the overhead time due to bit testing, locking, and synchronous in-memory copies. This time is extremely important for MMOs, because these overheads must be compensated by MMO developers by reducing the amount of work performed in a given tick. In addition to the runtime overhead of the recovery algorithms, we are also interested in measuring the time to checkpoint and the recovery time. While the behavior of the recovery time will be of great interest to MMO developers, the time to checkpoint gives us insight into the cost breakdown of the recovery time. It is equivalent to the time to replay the simulation (Section 4.2), so the remaining time spent during recovery is due to restoring the last checkpoint taken. We discuss these experiments in Sections 5.1 to 5.3.

To better understand realistic update patterns, we have also implemented a prototype game that simulates a medieval battle of the type common in many MMOs. It is based on the Knights and Archers Game of [37]. The simulation contains three types of characters: knights, archers, and healers, that are divided into two teams. Each team has a home base, and the objective is to defeat as many enemies as possible. Each unit is controlled by a simple decision tree. Knights attempt to attack and pursue nearby targets, while healers attempt to heal their weakest allies. Archers attempt to attack enemies while staying near allied units for support. Furthermore, each unit tries to cluster with allies to form squads.

In typical MMOs, not all characters are active at all times. In the Knights and Archers game, 10% of the characters are active at any given moment and the active set changes over time. Units leave and join the active set such that it is completely renewed every 100 ticks with high probability. We have instrumented this game to log every update to a trace file, which we then use as input to our checkpoint simulator. Table 5 summarizes the basic characteristics of our trace. Note that the update distribution follows the skew determined by the game logic. We consider these experiments in Section 5.4.

5. EXPERIMENTAL RESULTS

In this section, we present the results of our experimental evaluation. The goals of our experiments are twofold. First, we want to understand how the different checkpoint recovery algorithms behave as we scale the number of updates per tick (Section 5.1), what latency peaks we can observe (Section 5.2), and how the skew of the update distribution affects the algorithms (Section 5.3). Second, we want to understand how the checkpoint recovery algorithms perform with an update trace generated by realistic game simulations (Section 5.4).

5.1 Scaling on Number of Updates Per Tick

In this section, we evaluate how the different recovery methods scale as we increase the number of updates per game tick. Recall that an update is a change in one of the cells of the game state table.

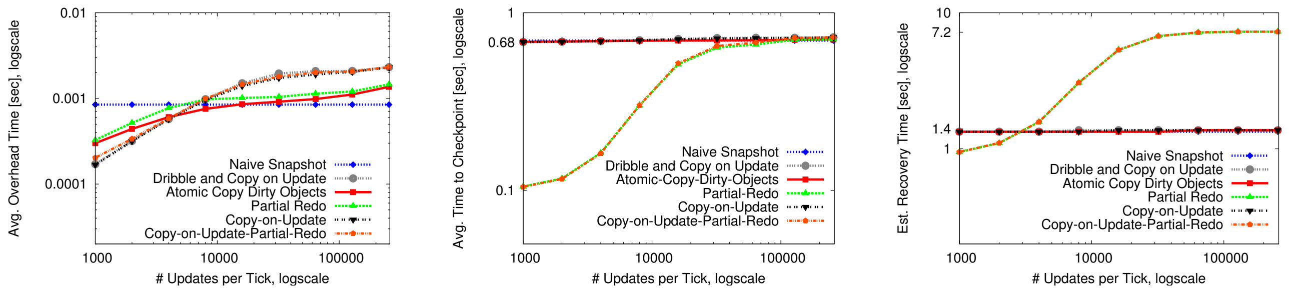
Average Overhead Time. We begin by analyzing the time added by the different algorithms to the game ticks; we call this time the *overhead time* or short *overhead*. Keeping the overhead low is key in MMOs because it represents how much recovery penalizes tick processing. Our simulation assumes that game logic and update processing take the whole tick length. A recovery method introduces overhead that stretches ticks beyond their previous length, reducing the frame rate of the game. In order to compensate for recovery overhead, MMO developers have to diminish the amount of work performed per tick. If the pauses introduced by a recovery algorithm are too high, then the use of that recovery algorithm becomes infeasible.

Figure 2(a) shows the effect of varying the number of updates per tick on the average overhead per tick. Recall that we have one million rows with ten columns each. In typical MMOs, a subset of the game objects are active at any given time. In battle games with very intense action sequences, update rates may reach hundreds of thousands of updates per tick [34]. This extreme scenario is the rightmost point of the graph. Game characters typically have a set of relatively static attributes along with attributes that are more often updated. Thus we expect this extreme situation to occur only in games with very hot objects that receive many updates per tick or in games in which a large set of active objects is moving every single tick.

The average overhead of Naive-Snapshot is 0.85 msec per tick — however, this average is not spread uniformly among all ticks of the game. The method performs a single synchronous copy per checkpoint in which all of the overhead is incurred. This copy takes nearly 17 msec (see Section 5.2), a value in excess of half the length of a tick.

Atomic-Copy-Dirty-Objects and Partial-Redo exhibit similar behavior as both methods perform eager copies of only the dirty objects. We see that these strategies outperform Naive Snapshot for update rates below 10,000 updates per tick. For such low update rates, these two methods achieve lower total memory copy time than Naive-Snapshot, which always copies the whole game state. Above 10,000 updates per tick, however, these methods tend to become worse than Naive Snapshot. At high update rates, most objects in the game state get updated between checkpoints. Thus these methods have to copy as much data in main memory as Naive Snapshot. In contrast to Naive Snapshot however, these methods must also pay for bit checking overhead for each update. At 256,000 updates per tick, this difference amounts to an average overhead of 1.4 msec for Atomic-Copy-Dirty-Objects versus 1 msec for Naive-Snapshot, a 60% difference.

The methods based on copy on update, namely Dribble-and-Copy-on-Update, Copy-on-Update, and Copy-on-Update-Partial-Redo, present an interesting trend. Under 8,000 updates per tick, these methods add less average overhead than Naive-Snapshot by up to a factor of five. In addition, these methods also outperform other eager copy methods. Above 8,000 updates per tick, how-



(a) Updates per tick vs. overhead time (b) Updates per tick vs. time to checkpoint (c) Updates per tick vs. recovery time

Figure 2: Overhead, checkpoint, and recovery times when scaling up on the number of updates per tick.

ever, these methods add more average overhead per tick by up to a factor of 2.7. In spite of adding more overhead when the number of updates per tick is larger, copy on update methods may still be the most attractive even in these situations. Recall that all eager methods mentioned previously, such as Naive-Snapshot, concentrate their total overhead of all the ticks associated with a checkpoint into one single tick. Copy on update methods, on the other hand, spread their overhead over a number of ticks, presenting smaller latency peaks even when the total latency introduced is larger than for eager methods (see Section 5.2). Unfortunately, even copy on update methods will introduce significant latency if most atomic objects in the game state are updated in at least one tick after a checkpoint. This situation happens in our experiment for update rates over 100,000 updates per tick. At these extreme update rates, all methods are undesirable in terms of latency. The strategy of last resort is to take the method with lowest total latency, i.e., NaiveSnapshot and to invest development effort into latency masking techniques for the game [4].

Checkpoint and Recovery Times. Figure 2(b) shows how the checkpoint time changes as we increase the number of updates processed in each tick of the game. Naive-Snapshot, Dribble-and-Copy-on-Update, Atomic-Copy-Dirty-Objects, and Copy-on-Update write the entire game state to disk on every checkpoint. As a consequence, these methods present constant checkpoint time of around 0.68 sec for all update rates. Note that Atomic-Copy-Dirty-Objects and Copy-on-Update copy only dirty objects in main memory. Given that the atomic objects dirtied every checkpoint are a significant fraction of the game state, the fastest way for these strategies to commit their checkpoint to the double backup is a single sequential write of the whole state.

Partial-Redo performs like Copy-on-Update-Partial-Redo. Again the time to flush the checkpoint to disk dominates the checkpoint time. These two algorithms rely on a log-based disk organization and perform sequential I/O. At 1,000 updates per tick, Partial-Redo and Copy-on-Update-Partial-Redo take 0.1 sec to write a checkpoint. That represents a gain of a factor of 6.8 over Naive-Snapshot.

In Figure 2(c), we see how this gain in checkpoint time translates into recovery time. Recall from Section 4.2 that the time to replay the simulation once a checkpoint is restored is roughly equal to the checkpoint time. Moreover, for Naive-Snapshot, Dribble-and-Copy-on-Update, Atomic-Copy-Dirty-Objects, and Copy-on-Update, the time to restore is equal to the time to read the checkpoint sequentially and thus roughly equal to the time these algorithms take to save the checkpoint to disk. As such, the recovery time for these algorithms is nearly twice their checkpoint times, reaching around 1.4 sec for all update rates.

Partial-Redo and Copy-on-Update-Partial-Redo show more interesting behavior. While their time to replay the simulation is again roughly equal to the checkpoint time, the methods differ in the time necessary to restore the checkpoint from disk. Partial-Redo and Copy-on-Update-Partial-Redo have the best checkpoint times, and thus the lowest times to replay the simulation. On the other hand, these algorithms have recovery times that are consistently worse than Naive-Snapshot above 4,000 updates per tick. These methods must read through a log in order to restore the checkpoint, leading to restore times much larger than for the other algorithms. While these restore times could be reduced by flushing the whole state to the log more often, doing so would also make the checkpoint times of these algorithms much closer to a simple sequential write of the whole game state, eliminating the advantage of a reduced time to replay the simulation. At 256,000 updates per tick, these methods spend 7.2 sec to recover, a value 5.4 times larger than the time required by Naive-Snapshot.

In summary, the three methods Dribble-and-Copy-on-Update, Copy-on-Update, and Copy-on-Update-Partial-Redo have the smallest overhead for low numbers of updates per tick. Even when the number of updates per tick is high, these methods may still be preferable given that they spread their overhead over several ticks instead of concentrating it into a single tick. Out of these three methods, Copy-on-Update and Dribble-and-Copy-on-Update are the most efficient in terms of recovery times, being vastly superior to Copy-on-Update-Partial-Redo for high numbers of updates per tick. Eager copy methods only display overhead significantly lower than copy on update methods with very high update rates. In these extreme situations, all methods introduce latency peaks in the game. The method with the lowest overall latency for these cases is Naive-Snapshot.

5.2 Latency Analysis

In order to better understand the latency behavior of the algorithms, we look in depth at a scenario with 64,000 updates per tick. Given the results in Figure 2(a), we expect copy on update methods to introduce nearly twice the average latency of eager copy methods. We have plotted the tick length as we run the algorithms in order to observe how much overhead is introduced on top of the basic tick length of 33 msec. The results are shown in Figure 3 for ticks 55 to 110 of the simulation. The same pattern for all methods repeats itself over the remaining 945 ticks not shown in the figure.

The figure clearly shows that eager copy methods introduce the largest overhead peaks, with some game ticks being lengthened by 17 msec. This value is approximately the same for Naive-Snapshot, Atomic-Copy-Dirty-Objects, and Partial-Redo. As the number of updates is relatively high in this scenario, most atomic objects in the game state are touched between successive checkpoints for all methods. As a consequence, all eager copy methods have identical synchronous copy times. This time is the single latency factor for Naive-Snapshot and dominates latency for the other eager copy

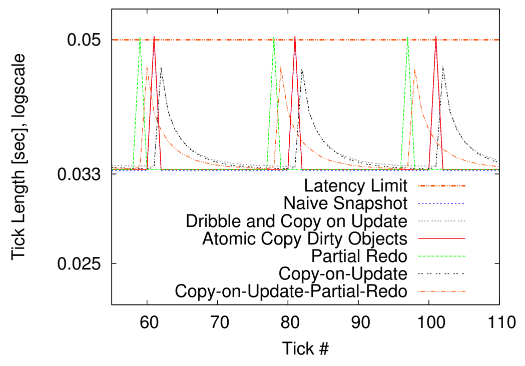


Figure 3: Latency analysis: 10M objects, 64K updates per tick.

methods, given that overheads for bit testing are small.

Considering that the tick frequency of the game is 30 Hz, and thus each tick is about 33 msec long, the eager copy algorithms introduce a pause of over half a tick in the game. As such, the latency impact of these algorithms is undesirable. In fact, we argue that pauses longer than half the length of a tick introduce latency that has to be dealt with by MMO developers via latency masking techniques [4]. In Figure 3, we plot an additional line that represents this latency limit of half a tick. Neither Naive-Snapshot, Atomic-Copy-Dirty-Objects, nor Partial-Redo is able to respect the latency bound.

Copy on update methods present different behavior. Their overhead is spread over a number of game ticks instead of being concentrated into a few ticks. The latency peak for all of these methods is 12 msec for the first tick after a checkpoint is started, dropping to seven msec for the second tick, four msec for the third, and even smaller times for subsequent ticks. In the first tick, no atomic objects are dirty yet and thus copy on update methods must incur locking, memory latency, and copying overhead for many objects. As we move through the checkpoint period, however, many updates will be applied to objects that have already been dirtied in previous ticks. For all of these updates, copy on update methods perform only inexpensive bit tests.

According to what we have discussed above, almost all of the atomic objects in the game state are touched between successive checkpoints for all methods. As copy on update methods perform copies randomly and have to acquire locks for synchronization, their total overhead is larger than the overhead to perform a sequential copy of the state, as done by eager copy methods. In contrast to eager copy methods, however, copy on update methods have a much better distribution of their overhead along ticks of the game. In addition, when smaller fractions of the game state are updated, copy on update methods tend to incur both lower and better distributed overhead than eager copy methods.

5.3 Effect of Skew

Finally, we evaluate the effect of update skew on the various recovery methods. The primary effect of increasing the skew is to decrease the number of dirty objects. This has no effect on the Naive-Snapshot algorithm, but it causes slightly different behavior in the other strategies. The graphs in this section are not log-scaled in order to emphasize the trends.

Average Overhead Time. Figure 4(a) shows the overhead as a function of skew. The relative performance of the strategies is similar to that in the previous experiments with about 64,000 updates per tick. The lowest overhead is introduced by Naive-Snapshot, while other methods fall within a factor of 2.5. Since the number of dirty objects decreases as the skew increases, we expect strategies that copy only dirty objects to perform better with high skew. Atomic-Copy-Dirty-Objects and Partial-Redo benefit less from skew, however, than Copy-on-Update and Copy-on-Update-Partial-Redo. Recall that the large number of updates per tick in the experiment leads to most of the atomic objects in the game state being updated between checkpoints. For Copy-on-Update, extreme skew diminishes the updated portion from roughly 100% to 84% of the atomic objects. All copy on update methods benefit more from a smaller fraction of objects updated than eager copy methods because they do not need to incur locking and latency overheads for these objects.

Interestingly, Dribble-and-Copy-on-Update also gets some benefit from skew, although it writes the whole game state to disk. This method copies fewer objects with high skew because game objects are copied only on the first update. Thus a larger fraction of the updates avoid copying, and more of the objects are flushed to disk directly by the asynchronous process.

Checkpoint and Recovery Times. Figure 4(b) shows the time to checkpoint as a function of the data skew, and Figure 4(c) shows the recovery time. As a large fraction of the atomic objects are updated between checkpoints, most strategies display very similar times to checkpoint. For Copy-on-Update-Partial-Redo and Partial-Redo, the time to checkpoint decreases as the skew increases, because there are fewer dirty objects and these methods benefit from fast sequential writes of only dirty objects to a log.

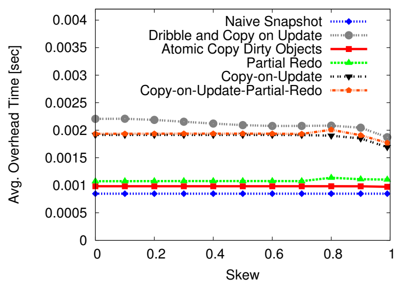
This effect is more clear in the graph that shows recovery time. Here the time for Copy-on-Update-Partial-Redo and Partial-Redo decreases from 7.3 sec to 6.3 sec. This graph also shows some of the same trends observed in the previous sections. Partial redo methods have much larger recovery times than other strategies, while the remaining strategies have similar recovery times given that the whole game state is written by them to disk.

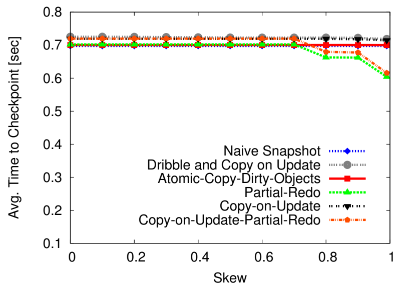
In summary, we conclude that skew has a fairly minor effect on the performance of the algorithms. Copy on update methods benefit relatively more from skew than other methods, as these methods may then avoid locking and memory latency overheads. Copy-on-Update-Partial-Redo and Partial-Redo have very large recovery times, and for this reason are not competitive.

5.4 Experiments with Prototype Game Server

In this section we run our simulation with the update trace of the Knights and Archers Game described in Section 4.4. Our goal is to understand whether the observations we made regarding Zipf distributions carry over to more realistic datasets. Recall that this trace contains 400,128 rows of 13 attributes. The trace applies updates only to 10% of the units, exhibiting an average update rate of 35,590 attributes per tick. This corresponds to the fact that many characters update their position during each tick (possibly only in one dimension), but that other attributes such as health remain relatively stable.

The results in Figure 5 confirm several of our previous observations. At the update rate exhibited by the game, copy on update methods have average per tick overheads that are larger than the overheads of eager copy methods. However, copy on update methods spread this overhead better over several ticks, while eager copy methods concentrate all of their overhead into a single tick. Copy-on-Update-Partial-Redo presents higher overhead than other copy on update methods. It reaches 1.6 msec compared to only 1.2 msec for Copy-on-Update. This effect occurs because Copy-on-Update-Partial-Redo checkpoints more often than Copy-on-Update, as can be seen in Figure 5(b). Recall from Section 5.2 that all copy on update strategies exhibit more overhead in the first few ticks of


(a) Skew vs. overhead time


(b) Skew vs. time to checkpoint

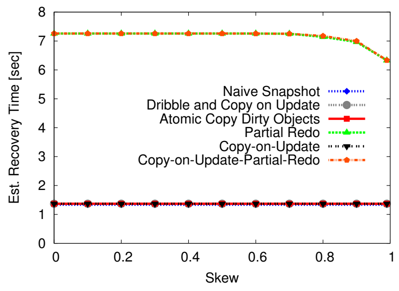
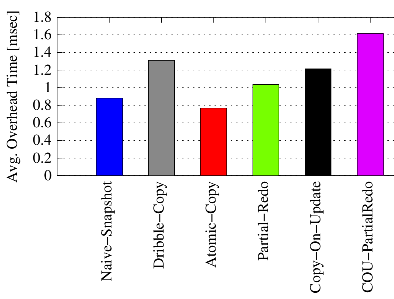
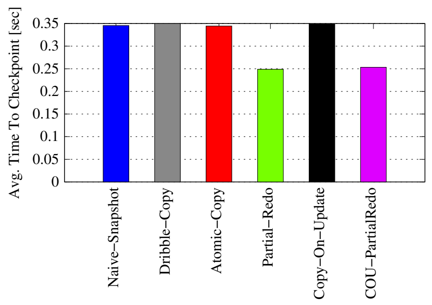

(c) Skew vs. recovery time

Figure 4: Overhead, checkpoint, and recovery times when varying the skew.


(a) Overhead time


(b) Time to checkpoint

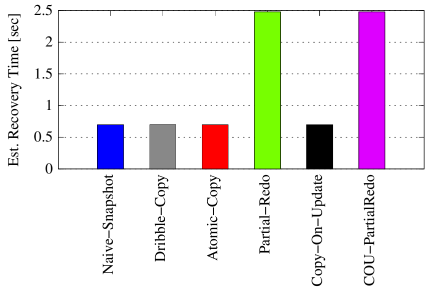

(c) Recovery time

Figure 5: Overhead, checkpoint, and recovery times for a trace with 400,128 units and updates to 10% of the units every tick.

a checkpoint than in later ticks. As Copy-on-Update checkpoints less often, it spends more of its time in the region of lower overhead than Copy-on-Update-Partial-Redo. Typically, one would expect longer recovery times as a consequence of less frequent checkpoints. In the case of Copy-on-Update-Partial-Redo, however, the recovery times shown in Figure 5(c) are also longer than for Copy-on-Update. This confirms our previous observation that methods based on partial redo are slower than other strategies since they have to read more of the log from disk. We saw this behavior previously when we had a very high update rate (Figures 2(c) and 4(c)).

Similar trade-offs can be observed for other strategies. Atomic-Copy-Dirty-Objects checkpoints more often than Partial-Redo, and shows both lower overhead and lower recovery time than the later. Atomic-Copy-Dirty-Objects is in fact the method with lower average overhead time, having a value slightly lower than Naive-Snapshot. Dribble-and-Copy-on-Update exhibits slightly higher overhead than Copy-on-Update, as Dribble-and-Copy-on-Update copies in main memory all objects touched on first update, while Copy-on-Update can restrict its copies to objects that have been dirtied since the last consistent image of the backup currently being written. As observed previously in other experiments, similar checkpoint and recovery times are observed for Naive-Snapshot, Atomic-Copy-Dirty-Objects, and the lazy methods Dribble-and-Copy-on-Update and Copy-on-Update. Overall, the game trace falls comfortably into the range of parameters we explored using synthetic data.

6. EXPERIMENTAL VALIDATION

In this section, we present experimental results that validate our simulation model against an actual implementation of two recovery methods: Naive-Snapshot and Copy-on-Update. These methods are the most relevant methods as identified by our simulation results. For a wide range of update rates, copy on update methods exhibit the best latency behavior and recovery times. When compared to Dribble-and-Copy-on-Update, Copy-on-Update introduces slightly less overhead time on the game as it copies only dirty objects on update. Copy-on-Update also consistently outperforms Copy-on-Update-Partial-Redo in terms of recovery times. For extreme update rates of over 100,000 updates per tick, Naive-Snapshot is the method with the lowest overhead time. Taken together, these two methods ourperform the other algorithms over the entire range of performance metrics included in our simulation model, including memory latency, locking overhead, and memory and disk bandwidths.

Validation Setup. We implemented Naive-Snapshot and Copy-on-Update in C++ and ran them on an Intel Core 2 Quad 2.4 GHz machine with 4MB cache and 4GB of main memory. The operating system used was Ubuntu Linux kernel 2.6.27-14. To avoid fragmentation at the filesystem level, we put the checkpoint files on a dedicated hard drive to which we wrote directly through a Linux block device. The disk was an 80GB 7200 rpm SATA drive with an 8MB cache.

We repeated the experiments from Section 5.1 for validation. Our implementation is driven by trace files with update distributions as described in Section 4.4. It is divided into a mutator thread and an asynchronous writer thread. In order to reproduce the work done in realistic games that tick at 30Hz, the mutator executes each tick in three phases: query, update, and sleep. The *query phase* is adjusted for each update rate in order to fill a tick. It performs a sequence of random lookups in the game state. After the query phase is over, the *update phase* processes the updates from the trace for the given tick. We keep all trace files completely loaded in main memory to avoid introducing disk access delays in the update phase. Finally, the (short) *sleep phase* fills the remaining time so that the game ticks at 30Hz. In our experiments, the sleep phase averaged less than 1 msec and we ensure that this time does not affect the measurement of overhead time. The asynchronous writer flushes consistent checkpoints to disk. The state to be written is either created by the mutator as in Naive-Snapshot, or it is directly accessed by the asynchronous writer using appropriate locks, as in Copy-on-Update.

Validation Results. Figure 6 shows the results from our implementation as we scale the number of updates per tick. For reference, the figure also shows the results predicted by our simulation model, where we calibrated the parameters in the simulation with the micro-benchmarks described in Section 4.3.

The trends predicted by our simulation model are closely matched by the implementation. The real implementation of Naive-Snapshot exhibits practically the same overhead, checkpoint, and recovery times as the simulation. This is expected as the performance of Naive-Snapshot depends essentially on the memory and disk bandwidths.

Copy-on-Update displays similar behavior regarding checkpoint and recovery times. The overhead time shown by the implementation is, however, larger than the simulation model's prediction by up to a factor of 3. This occurs because the real implementation has overheads not accurately modeled in our simulation, for example, lock contention between the mutator and the asynchronous writer and interference on the mutator from the I/O performed by the asynchronous writer. We have observed in a separate experiment that the latter effect is significant and explains why the difference between implementation and simulation increases for higher update rates. In spite of these differences, the simulation model accurately predicts the overhead trends shown by the algorithm.

7. RELATED WORK

Several approaches have been proposed for taking database snapshots [1]. Kähler and Risnes explore how to use logging to refresh a snapshot, proposing a technique that resembles incremental view maintenance [15]. Other work has investigated how to provide efficient access to a large collection of past snapshots [31, 32]. Our focus is different, as we target instead recording the most recent consistent snapshot of the data efficiently and as often as possible. Furthermore, in our scenario all primary data is main-memory resident for performance reasons, while these previous approaches assume disk storage of both the database and the snapshots.

Levy and Silberschatz propose a recovery scheme for databases with large main memories termed log-based backup [17]. Log-based backup works by continuously applying log records to a backup copy of the database. Both a redo log and the backup copy are maintained. In order to decouple transaction processing from log I/O, the authors suggest the use of stable memory to keep the log's tail. However, as previously discussed, schemes based on logging all game updates are infeasible for MMOs in practice.

Fuzzy checkpointing does not produce a consistent checkpoint, requiring the system to keep a physical log [13, 29]. In MMOs, this requirement makes this technique infeasible, given that all game updates would have to be logged to disk whenever a checkpoint is being taken.

A number of systems study how to optimize recovery in a distributed setting. ClustRa uses the idea of hot standbys and log shipping to provide high availability [14]. To achieve higher performance, log records are not written to disk, but rather to the main memory of a neighboring node. Whitney et al. execute transactions in both a primary site and in a number of backup sites [39]. Backup sites use a variant of the Naive-Snapshot algorithm from the previous section to save checkpoints to stable storage. More recently, Lau and Madden [16] and Stonebraker et al. [33] advocate a distributed design in which no logging is performed by the system. In contrast to Whitney et al. [39], all sites are active and thus updates are processed in all sites that replicate a given portion of the data. Data is replicated in a minimum of K sites so that the system will survive up to K site failures. If we apply their model to our scenario, we need to use K servers to execute copies of the discrete event simulation loop. This basic solution is similar to the process-pairs solution for high-performance scientific simulations [30]. While availability is high, system utilization is rather low ($1/K$), given that all active copies of the simulation loop perform redundant work. We follow instead a checkpoint recovery model, which increases utilization at a potential increase in recovery time; a detailed exploration of recovery methods for MMOs inspired by K-safety is future work.

8. SUMMARY AND RECOMMENDATIONS

In this paper, we have experimentally evaluated the performance of main-memory checkpoint recovery techniques for MMOs. Instead of requiring MMO developers to hand-code persistence logic for their games, our experimental study shows that these techniques can be used as a viable alternative to provide durability for local updates.

One important conclusion of our study is that not all checkpoint recovery techniques are equally suited for MMOs. MMOs have stringent latency requirements, ruling out algorithms that introduce excessive pauses in the game's discrete-event simulation loop. In our study, we have quantified the duration of these pauses and observed which algorithms introduce the least overhead. We summarize our principal findings as a set of **recommendations** for MMO developers:

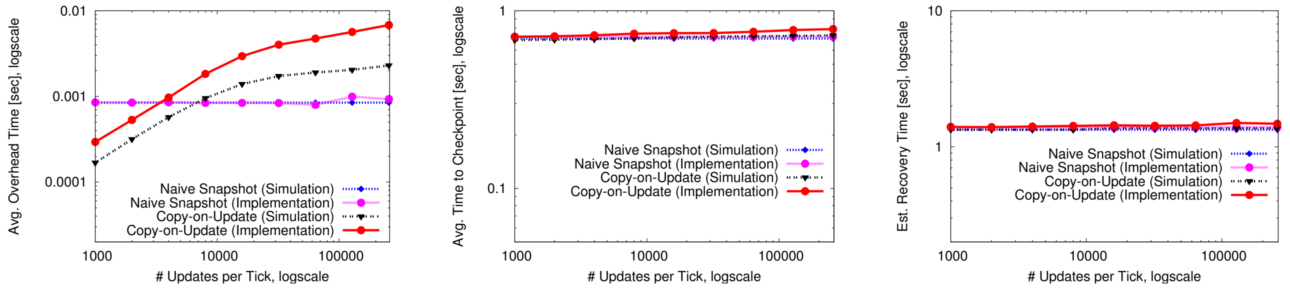
1. Methods based on copy on update that checkpoint only dirty objects have clear advantages over other checkpoint recovery methods for MMOs. These methods introduce up to a factor of five less overhead in the game than methods that perform eager copies in main memory when the number of updates is low. In addition, even when update rates are high, these methods avoid latency peaks by spreading their overhead over a number of game ticks.

2. Some games may have such extreme update rates that they update a huge number of objects within some tick. In these situations, all methods we have evaluated will introduce significant latency peaks in the game. The method that has shown the lowest overall latency under such heavy and skewed load is Naive-Snapshot, as it avoids any overheads associated with locking and performs fully sequential memory copies of the whole game state.

3. Methods based on a double-backup organization that checkpoint only dirty objects exhibit recovery times either better or comparable to other methods across all ranges of parameters we have investigated. In particular, methods based on checkpointing dirty objects to a log file have larger recovery times due to the time needed to process the log when restoring the checkpoint.

4. The best method in terms of both latency and recovery time is Copy-on-Update. This method combines checkpointing of dirty objects with copy on update and a double-backup organization. When compared to Naive-Snapshot, we have observed up to a factor of five gain in latency and no degradation in recovery time.

In the future, we plan to explore how choices for different hardware parameters affect the performance of the various recovery



(a) Updates per tick vs. overhead time (b) Updates per tick vs. time to checkpoint (c) Updates per tick vs. recovery time

Figure 6: Validation of overhead, checkpoint, and recovery times when scaling up on the number of updates per tick. We compare the results of our simulation model with a real implementation of Naive-Snapshot and Copy-on-Update.

algorithms. In addition, we plan to extend our analysis to multi-server MMOs. This will require synchronizing and recovering shared state between servers.

Acknowledgments. This research was supported by the New York State Foundation for Science, Technology and Innovation under Agreement C050061, the National Science Foundation under Grants 0725260 and 0534404, the iAd Project funded by the Research Council of Norway, and by Microsoft. Any opinions, findings, conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] M. Adiba and B. Lindsay. Database Snapshots. In *Proc. VLDB*, 1980.

[2] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent Advances in Checkpoint/Recovery Systems. In *Proc. IPDPS*, 2006.

[3] S. Carless. The Activision/Blizzard Merger: Five Key Points.

[4] M. Claypool and K. Claypool. Latency and Player Actions in Online Games. *Communications of the ACM*, 49(11):40–45, 2006.

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

[6] B. Dalton. Online Gaming Architecture: Dealing with the Real-Time Data Crunch in MMOs. In *Proc. Austin GDC*, 2007.

[7] 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.

[8] Cornell Data-Driven Games Project. http://www.cs.cornell.edu/bigreddata/games .

[9] J. Gray. Notes on Data Base Operating Systems. *Operating systems–an advanced course, Springer Verlag*, pages 393–481, 1978.

[10] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *Proc. SIGMOD*, 1994.

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

[12] E. Guizzo. The Game-Frame Guild. *IEEE Spectrum*, August, 2008.

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

[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] B. Kähler and O. Risnes. Extending Logging for Database Snapshot Refresh. In *Proc. VLDB*, 1987.

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

[17] E. Levy and A. Silberschatz. Incremental Recovery in Main Memory Database Systems. *IEEE TKDE*, 4(6):529–540, 1992.

[18] A.-P. Liedes and A. Wolski. SIREN: a Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for In-Memory Databases. In *Proc. ICDE*, 2006.

[19] G. Lohman and J. Muckstadt. Optimal Policy for Batch Operations: Backup, Checkpointing, Reorganization, and Updating. *ACM TODS*, 2(3):209–222, 1977.

[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] S. Posniewski. Massively Modernized Online: MMO Technologies for Next-Gen and Beyond. In *Proc. Austin GDC*, 2007.

[23] S. Posniewski. SQL Considered Harmful. In *Proc. GDC*, 2008.

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

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

[26] Richard Bartle. *Designing Virtual Worlds*. New Riders, 2003.

[27] Richard Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, 2000.

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

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

[30] B. Schroeder and G. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conf. Ser.*, 78, 2007.

[31] R. Shaull, L. Shrira, and H. Xu. Skippy: a New Snapshot Indexing Method for Time Travel in the Storage Manager. In *Proc. SIGMOD*, 2008.

[32] L. Shrira and H. Xu. SNAP: Efficient Snapshots for Back-in-Time Execution. In *Proc. ICDE*, 2005.

[33] 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.

[34] T. Sweeney. The Next Mainstream Programming Language: a Game Developer's Perspective. In *Proc. POPL*, 2006.

[35] The New New Economy: Earning Real Money in the Virtual World, 2005. http://knowledge.wharton.upenn.edu/article.cfm?articleid=1302.

[36] Thor Alexander, editor. *Massively Multiplayer Game Development 2*. Charles River Media, 2005.

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

[38] W. White, C. Koch, N. Gupta, J. Gehrke, and A. Demers. Database Research Opportunities in Computer Games. *ACM SIGMOD Record*, 36(3), 2007.

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