FAULT TOLERANCE FOR MAIN-MEMORY APPLICATIONS IN THE CLOUD

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Tuan Cao

May 2013

FAULT TOLERANCE FOR MAIN-MEMORY APPLICATIONS IN THE CLOUD

Tuan Cao, Ph.D.

Cornell University 2013

Advances in hardware have enabled many long-running applications to execute entirely in main memory. With the emergence of cloud computing, thousands of machines could be made available to deploy such applications with lowered operational and maintenance costs. While achieving substantially better performance, these applications have encountered new challenges in achieving fault tolerance; i.e., to ensure durability in the event of a crash. In addition, many of these applications, such as massively multiplayer online games, main-memory OLTP systems, main-memory search engine and deterministic transaction processing systems, must sustain extremely high update rates – often hundreds of thousands of updates per second. They also demand extremely high throughput (e.g. scientific simulation) or low latency (e.g. massively multiplayer online games). To support these demanding requirements, these applications have increasingly turned to database techniques. In this dissertation, we propose an approach to provide fault tolerance for main-memory applications without introducing excessive overhead or latency spikes.

First, we evaluate the applicability of existing checkpoint recovery techniques developed for main-memory DBMS. We use massively multiplayer online games (MMOs) as our motivating example. In particular, we show how to adapt consistent checkpointing techniques developed for main-memory databases to MMOs. Furthermore, we provide a thorough simulation model and evaluation of six recovery strategies. Based on our results, we argue that not all state-of-the-art checkpoint recovery techniques are equally suited for low-latency and high-throughput applications such as MMOs. These algo-

rithms either use locks or large synchronous copy operations, which hurt throughput and latency, respectively.

Next, 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.

Finally, we present BRRL, a library for making distributed main-memory applications fault tolerant. BRRL is optimized for cloud applications with frequent *points of consistency* that use data-parallelism to avoid complex concurrency control mechanisms. BRRL differs from existing recovery libraries by providing a simple table abstraction and using schema information to optimize checkpointing.

BIOGRAPHICAL SKETCH

Tuan Cao was born in Hanoi, the capital of Vietnam, to wonderful and lovely parents. Tuan graduated from HUS High School for Gifted Students in 2000, majored in Physics. Before going aboard for higher education, Tuan studied mathematics for one year in School of Applied Mathematics and Informatics, Hanoi University of Science and Technology. In 2005, he graduated from Pune University, India with a B.E. in Computer Engineering. During his undergraduate in India, Tuan participated in many national software contests and won various prizes, namely the 1st prize in Tryst, the 3rd prize in Techkrity and the 1st prize in Showcase, all India open software contests organized by IIT Delhi, IIT Kanpur, and IISc Bangalore respectively. Tuan also won the 2nd prize in Vietnam Intellectuals 2006 in Vietnam. In this contest, he also won the 1st prize in the Network Security group.

Prior to coming to Cornell in 2007, Tuan was a full-time employee at Yahoo! India. At Cornell, he has worked on the Games and Simulation project, focusing on applying database technologies to provide fault-tolerance for main-memory applications running in the cloud. Tuan received an M.S. in computer science in 2010 and completed a graduate minor in business. During his graduate studies, he interned twice at Google, with the Gmail Backend Team and with the Structured Data Research Group, in 2008 and 2012 respectively. Tuan also filed a patent, titled “Searching for Join Candidates”, while interning at Google.

To my parents and especially my sister, who introduced me to math, physics and literature.

ACKNOWLEDGEMENTS

I am especially grateful to my advisor, Johannes Gehrke, who led me into the database research world. Johannes taught me how to be an independent researcher; his endless patience encouraged me to find my own way. From Johannes, I not only learned to do research, but also learned to manage work-life balance. I really admire the fact that Johannes could advise nine PhD students while spending a decent amount of time with his family. Looking at how he played with his kids at the department party made me believe that he is one of the best fathers in the world. For all my PhD years, Johannes has always been a great source of inspiration and I must admit that I am very lucky to have him as my advisor.

I am also grateful to Andrew Myers, Robbert van Renesse, and Levent Orman for their insightful comments and questions while serving on my committee. In addition, Andrew Myers cultivated the love of teaching in me; I will never forget the two classes I took with him. Robbert van Renesse revived my love of playing music; I am proud to say that I learned ukulele from the best ukulele player in town. And finally, interacting with Levent Orman, inside and outside the class, is my great pleasure. In fact, I recommended quite a few students in our department to work with him to do a minor in business.

Besides my committee members, I owe much of my development as a researcher to Marcos Vaz Salles. Marcos was my mentor during his time at Cornell. I still remember the first few months when I joined the Cornell Database Group, I was very nervous since I knew just a little about databases. Marcos, with great understanding, kind encouragement and insightful criticism, taught me all the basic concepts of databases, brought me up to speed in just a short period of time. Marcos was practically my second advisor; without him, I could not reach this far.

I have been fortunate to have worked with many excellent colleagues in the Cornell Database Group. I owe a big debt to Alan Demers and Ben Sowell, who helped me to

build up many skills. Alan Demers has been like a mentor to me, providing much needed support and guidance. His door was always open and I really enjoyed the discussions with him throughout my PhD life. Ben Sowell is a role model I have been following; I learned how to lead a discussion from Ben, and more importantly, how to read papers effectively and how to write papers comprehensively from Ben.

I would like to thank other members of the Cornell Database Group; especially Michaela Götz, Sudip Roy and Tao Zou, with whom I always had enlightening conversations. I would also like to thank my Systems Lab friends, who were an invaluable resource of fun and joy at work. I will really miss the Systems Lab culture after I graduate. I would also like to thank my batchmates, Hussam Abu-Libdeh and Hu Fu for their sharing and accompanying during my PhD. Running marathons with Hussam and discussing political issues with Hu Fu are my never-forgotten memories.

Prior to joining the Cornell Database group, I worked with Paul Francis and his two brilliant students, Hitesh Ballani and Saikat Guha. As a new graduate students, I learned many things from them and I am truly indebted to them. During my journey as a PhD student, I also interned at Google and I would like to thank my internship mentors, Balaji Raghavan and Nitin Gupta. They taught me many lessons which I could not learn in an academic environment. I would also like to thank Varun Kapoor, Kim Cuong Pham, Tudor Marian and Ryan Peterson for their help and fruitful guidance for jobs interview. They made my dream to work at Google become true.

I am fortunate enough to have amazing Vietnamese mentors and friends. I benefited greatly from AnHai Doan when seeking for career advice. Moreover, whenever stressed or depressed, I kept telling myself that Anh Hai would always be there to help me. In retrospect, Anh Hai had an incredible impact on my intellectual development. I can't thank him enough for all of his support and encouragement during my graduate study. The Vietnamese society at Cornell (VITCO) provided a warm atmosphere which made

me feel like living in a big family. I always consider my roommate, Thanh Nguyen, as a big brother who I can share my happiness and sadness and my badminton teacher, Duyen Bui, as a little sister who I enjoyed every single moment we hang out together.

None of this would have been possible without the love and support of my family; especially my sister, who encouraged me to pursue a PhD in Computer Science.

Finally, I would like to thank the agencies that funded my research as a graduate student. The work reported in this dissertation is 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.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Every cloud has its silver lining but it is sometimes a little difficult to get it to the mint. – D. Marquis

Data-driven applications such as OLTP and data warehousing systems, data mining algorithms, search engines, computer games, and large-scale simulations, have traditionally required disks to hold their data. But increasingly, due to the low-cost availability of terabytes of main memory within a cluster and the resulting improvements in speed, these data-driven applications are moving towards completely main-memory based implementations.

Systems with huge main memories were once available exclusively to entities with huge financial resources. But low-cost cloud infrastructure has become a great equalizer: At current prices, the rental cost of a cluster of 600 machines with over 1 TB of aggregate main memory is less than $55 per hour in the cloud [6]. This means that scientists can now afford to run data-driven applications for their terabyte-sized datasets at unprecedented speeds in-memory.

Main memory data-driven computing in the cloud brings new challenges. One of the greatest of these is fault tolerance. If an application resides only in main memory, we need to apply some methodology to persist its state in case of crashes or failures. Most cloud computing services are built from commodity hardware, which is not especially reliable. As the number of nodes in a computation grows, so does the chance of failure; for many data-intensive scientific applications the mean time to interrupt (MTTI) is considerably less than the application running time [23]. This situation is predicted to grow exponentially worse as cluster sizes increase [102]. Naively scaling an application

to 100 nodes in the cloud causes about a 30-fold reduction in its mean time to interrupt. Estimates based on recent cluster growth rates predict that the utilization of a large parallel application relying on conventional checkpoint recovery techniques will fall to *zero* by 2013 – the application will spend all its time writing checkpoints or recovering. As a result, there is a dire need in developing efficient checkpoint-recovery techniques that scientists can easily use to make their data-driven applications fault-tolerant.

In this dissertation, we present three projects. The first project examines the applicability and challenges of using checkpoint-recovery techniques to provide fault tolerance. The second project presents a new class of novel checkpoint-recovery algorithms that trade additional space in main memory for significantly lower overhead and latency. In particular, we leverage three copies of the application state to achieve almost an order of magnitude lower overhead and nearly constant latency. Nevertheless, for applications with large state size, we also present memory-optimized checkpoint-recovery algorithms which only use two copies of the application state. The third project demonstrates the practicality of these algorithms with BRRL, a library for making distributed main memory applications fault tolerant. In the remaining of this chapter, we introduce these projects in more detail.

1.1 An Evaluation Of Existing Checkpoint-Recovery Techniques

We use massively multiplayer online games (MMOs) as a running example throughout this evaluation. MMOs are persistent virtual worlds that allow tens of thousands of users to interact in fictional settings [46, 95]. 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 [20]. 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 [114].

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 [77], 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 [28, 46, 90]. 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 dissertation, we experimentally evaluate a number of traditional main-memory checkpoint recovery strategies for use with MMOs [70, 98, 123]. 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.

1.2 Fast Checkpoint-Recovery Algorithms For Frequently Consistent Applications

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 [109, 59].

There is a wide range of FC applications. For example, certain classes of main-memory OLTP systems also have frequent points of consistency. Traditional OLTP systems are heavily multi-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 [109] (and its commercial version VoltDB [119]) 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 [118]. Other examples of FC applications include new data-parallel systems to program the cloud, such as BOOM [5], deterministic transaction processing systems [115, 123], and in-memory search engines [111].

Many FC 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. Traditional database recovery techniques, such as fuzzy checkpointing and ARIES-style recovery, that rely on physical logging, simply cannot sustain this update rate. Since physical logging records physical addresses and values of updated objects, the number of physical log records is proportional to the number of updates. As a result, applications with high update rates quickly saturate the I/O bandwidth while trying to make these physical log records durable. Therefore, logical logging, which logs high level operations, is the method of choice in designing checkpoint-recovery algorithms for applications with high update rates. Depending on logging granularity, a logical log record could correspond to tens or hundreds of physical log records.

We leverage frequent points of consistency in FC applications to take periodic checkpoints of the entire application state and use logical logging to provide durability between checkpoints. At any time, the combination of a checkpoint plus the logical log from the time the checkpoint is taken contains a complete copy of the application state; the application state can simply be constructed by reapplying the logical log to the checkpoint. 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. Our experimental study has evaluated existing main-memory checkpointing algorithms for use in MMOs [99], however these algorithms either use locks or large synchronous copy operations, which hurt throughput and latency, respectively.

In this dissertation, 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.

1.3 Memory Optimized Checkpoint-Recovery Algorithms

Our approach is inspired by the extensive work in checkpoint-recovery algorithms for main-memory DBMS [33, 72, 92, 97, 98, 101, 123]. In addition, we draw from early work in operating systems, such as the Recovery Box [9]. Surprisingly, little of that work takes advantage of natural points of consistency, even though they are increasingly common in data-driven applications. We observed the overhead of these algorithms to be directly correlated with update rates. For low to moderate update rates, a copy-on-write technique performed best; for high update rates, the best technique focused on eagerly making entire main-memory copies of the game state. Our study demonstrated that previous methods can be adapted to take advantage of natural points of consistency in data-driven applications. In particular, our Wait-Free algorithms achieve significantly better latency and lower overhead. However, they incur a substantial memory overhead cost; for example, the Wait-Free Ping-Pong algorithm uses three copies of the application state.

In this dissertation, we present a series of optimization techniques which we can apply to the existing checkpoint-recovery algorithms to lower their memory usage. In addition, some of these optimization techniques also exploit modern hardware architecture to reduce the number of cache misses as well as to build low-overhead synchronization between the Mutator and the Asynchronous Writer. Our experiments show that these optimization techniques achieve a comparable performance with the existing Wait-Free algorithms while utilize only two copies of the application state.

1.4 BRRL(Big-Red Recovery Library): A Recovery Library For Main-Memory Applications In The Cloud

We have integrated all our techniques in to the Big Red Recovery Library (BRRL), a recovery library for frequently consistent applications in the cloud. Unlike existing checkpointing libraries such as Libckpt [87] and PORCH [93], which are designed solely for single-node applications, BRRL is primarily intended for distributed applications deployed on the cloud or other large shared-nothing clusters. Many of these applications have embraced data-parallelism in order to scale to a very large number of nodes and are frequently designed with limited or periodic synchronization in place of traditional lock-based concurrency control. BRRL takes advantage of these *points of consistency* to take efficient checkpoints without interfering with application throughput or introducing excessive latency. Additionally, BRRL includes an efficient and low-latency implementation of logical logging to ensure that checkpointed applications can recover to the point of failure.

Another key way in which BRRL differs from previous work is that it uses schema information to select the best checkpointing algorithms for mutable and immutable state.

Developers implement their state using a BRRL provided table abstraction, and they annotate the schema with access pattern details to guide BRRL's optimization decisions. For instance, updates to append-only tables can be logged without the need for additional checkpoints. BRRL uses the high-performance Wait-Free Ping-Pong checkpointing algorithm, and it reorganizes the physical storage of tables so that access to checkpointing data is cache optimized. Unlike BRRL, most existing approaches provide a generic memory abstraction and use compiler support to add appropriate checkpointing logic [15, 16]. While these approaches require very little information from the developer, their generality may lead them to choose sub-optimal checkpointing algorithms.

1.5 Contributions of this dissertation

This section summarizes the contributions of each project presented in this dissertation.

1. To the best of our knowledge, we provide the first benchmarking to evaluate the applicability of existing checkpoint recovery techniques for main-memory DBMS to MMO workloads. We make two primary contributions.

   (a) 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.

   (b) 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 [38].

2. We formally define a prevailing class of main-memory applications, namely frequently consistent applications. We take advantage of frequent points of consistency in many of these applications to develop novel checkpoint-recovery techniques. The work makes the following contributions.

   (a) We analyze the performance bottlenecks in prior work, and propose two new algorithms to address them; Wait-Free Zigzag and Wait-Free Ping-Pong are designed to perform well over a wide range of update rates.

   (b) 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 [99]. We find that our algorithms are particularly amenable to these low level optimizations.

   (c) 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 order of magnitude over a wide range of update rates. It also completely eliminates the latency spikes that plagued previous consistent checkpointing algorithms.

   (d) We design and implement a series of optimization techniques which exploit modern computer architectures to design low-overhead operations. When apply these techniques to the existing checkpoint-recovery algorithms, we achieve comparable performance with lower memory usage.

3. We have developed BRRL, a library for making distributed main-memory applications fault tolerant. We make the following contributions.

(a) BRRL provides a simple table-oriented API that makes it easy for developers to make their applications fault tolerant in the cloud.

(b) We present a concrete implementation of a durable main-memory transaction processing system using BRRL. Our experiments show that this system achieves good performance at low cost.

1.6 Dissertation Outline

Chapter 2 presents our experimental evaluation of checkpoint recovery for massively multiplayer online games (MMOs). Chapter 3 describes the design and implementation of our fast checkpoint-recovery algorithms for frequently consistent applications. Chapter 4 presents a series of optimization techniques to enhance existing algorithms. Chapter 5 demonstrates BRRL, a recovery library for main-memory cloud computing applications. Chapter 6 concludes.

CHAPTER 2

AN EVALUATION OF CHECKPOINT-RECOVERY FOR MASSIVELY MULTIPLAYER ONLINE GAMES

Get your facts first, and then you can distort them as much as you please.

– Mark Twain

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 scalability, forcing MMO developers to either invest in high-end hardware or to over-partition their virtual worlds.

In this chapter, we evaluate the applicability of existing checkpoint recovery techniques developed for main-memory DBMS to MMO workloads.

2.1 Architecture of an MMO

In this section, we describe the architecture of an MMO [46, 95, 116, 121]. Most MMOs use a client-server architecture, as shown in Figure 2.1. 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 [46] and Taikodom [47] are pioneering *shardless*

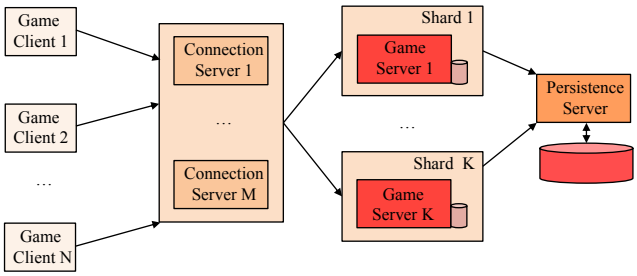


Figure 2.1: Architecture of a typical MMO.

architectures in which all players interact in the same virtual world.

In this dissertation, 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 2.1. This may be a large mainframe as used in Taikodom [47] or a cluster with tightly synchronized clocks. This work is a first step towards a more general solution for sophisticated network topologies.

2.1.1 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 game 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 up-

dated 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 [122]. The only assumption 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.

2.1.2 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 [46]. 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 2.1.

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 [29, 95], 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 [112]. Traditional DBMS cannot sustain this update rate, and existing MMOs have either created ad-hoc checkpointing strategies or resorted to costly special purpose hardware [91]. For instance, EVE Online uses RAM-based SSD drives for their database servers to handle logging and transaction processing [46].[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.

2.1.3 Recovery Requirements for MMOs

We consider a principled approach to durability using a checkpoint recovery approach first proposed for main-memory databases [70, 123, 98] and scientific simulations [101]. 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.

For example, since MMOs are interactive, it is important for simulation ticks to occur at a uniform rate: an inconsistent tick rate can distort time in the virtual world, and be frustrating to players [26]. 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

[1] These drives, which start at $90,000, have a sustained *random* I/O rate of up to 3GB/s [94, 75]

server states to produce smoother animation). Scientific simulations, on the other hand, tend to be non-interactive [96]. 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.

2.2 Main-Memory DBMS Recovery

In this section, we review the existing main-memory checkpoint-recovery techniques.

Objects Copied	**Eager Copy**		**Copy on Update**	
	Double Backup	**Log**	**Double Backup**	**Log**
All Objects	Naive Snapshot		Dribble and Copy on Update	
Dirty Objects	Atomic Copy Dirty Objects	Partial Redo	Copy on Update	Copy on Update Partial Redo

Table 2.1: Algorithms For Checkpointing Game State

2.2.1 Checkpointing for MMOs

Before enumerating the algorithms we evaluate, we consider the implications of the architecture described in Section 2.1 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 [101]. 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 [116]. 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 [101].

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.

2.2.2 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 [98]. Depending

on the disk organization, we may be able to capitalize on sequential I/O when flushing the checkpoint to disk.

Table 2.1 shows how existing algorithms fit into this design space. We describe the algorithms in detail below.

Naive-Snapshot. This is the simplest consistent checkpointing technique, which quiesces the system at the end of a tick and eagerly creates a consistent copy of the state in main memory. Naive-Snapshot then writes 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 [14, 72, 101, 123]. 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 [26], 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 [97]. 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 [92]. 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-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 [92, 98]. 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 [98]. 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 [42]. 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. This algorithm refines Dribble-and-Copy-on-Update to copy only dirty objects [33, 98]. 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 [70]. In our experiments, all copies are performed at the level of atomic objects whose size may be configured (Section 5.2). 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.

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 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 2.2: Subroutine Implementations for Checkpoint Recovery Algorithms

2.3 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 2.2. 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 extensively 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 [38]. All of our code is written in Java 1.6.

In the following, we discuss in detail our implementation of the checkpointing algorithms (Section 5.2) and our simulation model (Section 2.3.2). In addition, we describe the simulation parameters (Section 2.3.3) and the datasets (Section 2.3.4) used in our evaluation.

2.3.1 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 2.1, 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 [70]. 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

Checkpointing Algorithmic Framework

```
Input: ObjectSet O_all containing all objects in game data
do synchronous on end of game tick :
    if last checkpoint finished then
        // perform synchronous copy actions at end of tick:
        ObjectSet O_copy ← Copy-To-Memory(O_sync ⊆ O_all)
        if O_copy ≠ ∅ then
            do asynchronous Write-Copies-To-Stable-Storage(O_copy)
        end
        // register synchronous handler for update events:
        do synchronous on each Update u of an Object o ∈ O_all :
            Handle-Update(u, o)
        end
        // perform asynchronous copy of remaining information:
        do asynchronous Write-Objects-To-Stable-Storage(O_all \ O_sync)
    end
end
```

synchronously 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 2.2 by providing appropriate implementations of the subroutines in the Checkpointing Algorithmic Framework. We summarize in Table 2.2 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.

2.3.2 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 [26]. 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:

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

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,

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

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 counterintuitive (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 2.1 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.

2.3.3 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 2.3: Parameters for cost estimation

As described in Section 2.3.2, 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 2.3. 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 2.3 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 micro-benchmarks 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.

2.3.4 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 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"

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 2.4: 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 2.5: Characteristics of the update trace from our prototype game server.

objects. Our Zipfian generator is from [44]. 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 2.4 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 2.3.2), so the remaining time spent during recovery is due to restoring the last checkpoint taken. We discuss these experiments in Sections 2.4.1 to 2.4.3.

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 [122]. 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.

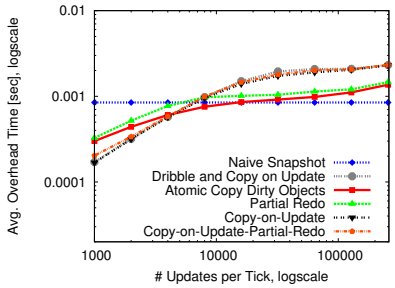
In typical MMOs, not all characters are active at all times. In the Knights and Archers game, 10% of the characters are active at any given moment and the active set changes over time. Units leave and join the active set such that it is completely renewed every 100 ticks with high probability. We have instrumented this game to log every update to a trace file, which we then use as input to our checkpoint simulator. Table 2.5 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 2.4.4.

2.4 Experimental Results

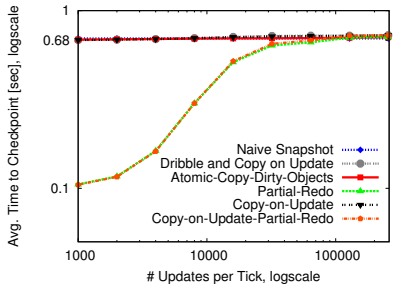
In this section, we present the results of our experimental evaluation. The goals of our experiments are twofold. First, we want to understand how the different checkpoint recovery algorithms behave as we scale the number of updates per tick (Section 2.4.1), what latency peaks we can observe (Section 2.4.2), and how the skew of the update distribution affects the algorithms (Section 2.4.3). Second, we want to understand how the checkpoint recovery algorithms perform with an update trace generated by realistic game simulations (Section 2.4.4).

2.4.1 Scaling on Number of Updates Per Tick

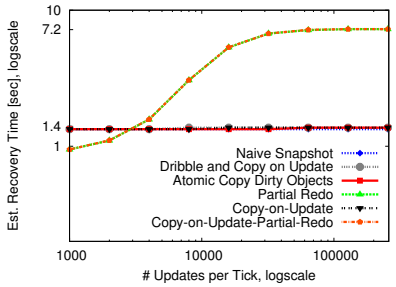
In this section, we evaluate how the different recovery methods scale as we increase the number of updates per game tick. Recall that an update is a change in one of the cells of the game state table.



(a) Updates per tick vs. overhead time



(b) Updates per tick vs. time to checkpoint



(c) Updates per tick vs. recovery time

Figure 2.2: Overhead, checkpoint, and recovery times when scaling up on the number of updates per tick.

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 2.2(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 [112]. 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 2.4.2), 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 over-

head 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, however, 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 2.4.2). 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 [26].

Checkpoint and Recovery Times. Figure 2.2(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 2.2(c), we see how this gain in checkpoint time translates into recovery time. Recall from Section 2.3.2 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.

2.4.2 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 2.2(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 2.3 for ticks 55 to 110 of the simulation. The same pattern for all

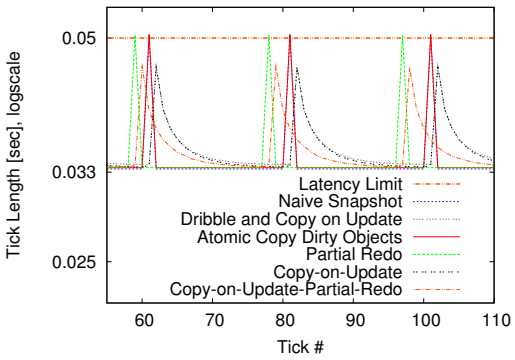


Figure 2.3: Latency analysis: 10M objects, 64K updates per tick.

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 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 [26]. In Figure 2.3, 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.

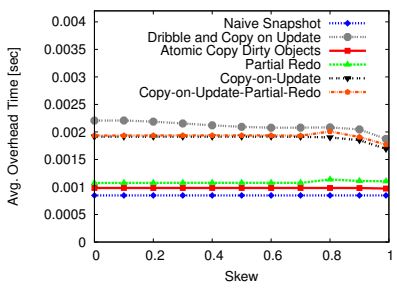
Copy on update methods present different behavior. Their overhead is spread over a

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.

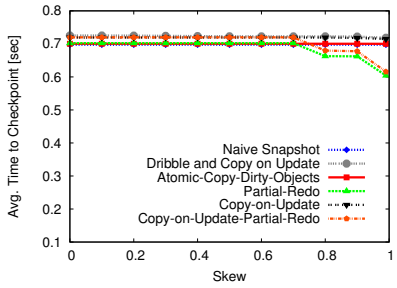
According to what we have discussed above, almost all of the atomic objects in the game state are touched between successive checkpoints for all methods. As copy on update methods perform copies randomly and have to acquire locks for synchronization, their total overhead is larger than the overhead to 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.

2.4.3 Effect of Skew

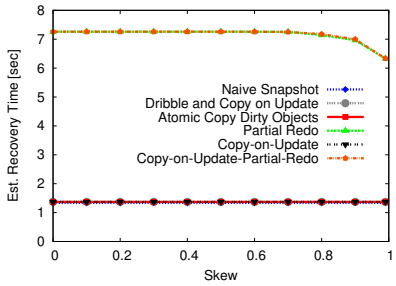
Finally, we evaluate the effect of update skew on the various recovery methods. The primary effect of increasing the skew is to decrease the number of dirty objects. This has no effect on the Naive-Snapshot algorithm, but it causes slightly different behavior in the other strategies. The graphs in this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



(a) Skew vs. overhead time



(b) Skew vs. time to checkpoint



(c) Skew vs. recovery time

Figure 2.4: Overhead, checkpoint, and recovery times when varying the skew.

the trends.

Average Overhead Time. Figure 2.4(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.

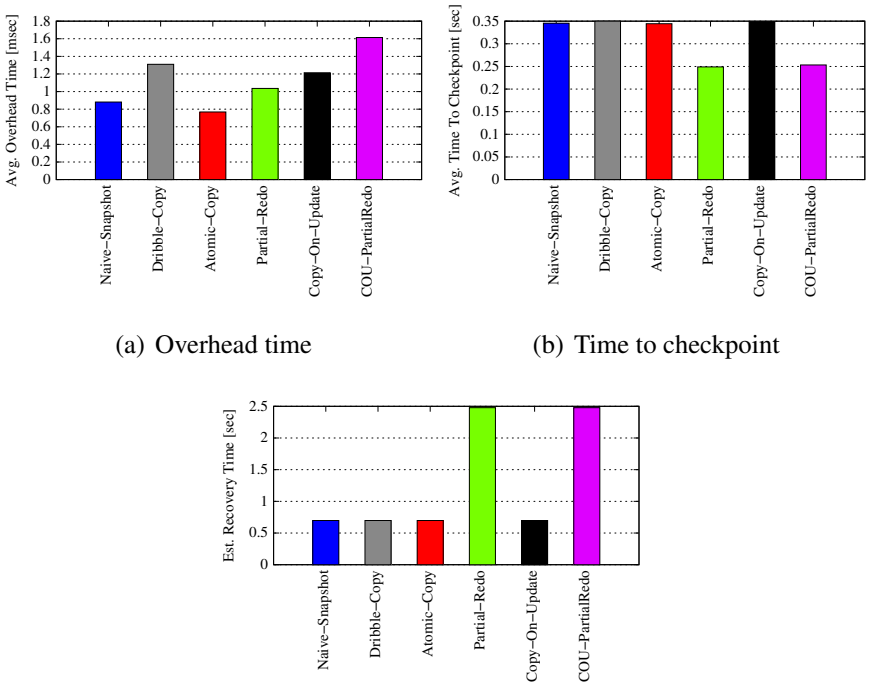
Interestingly, Dribble-and-Copy-on-Update also gets some benefit from skew, although it writes the whole game state to disk. This method copies fewer objects with high skew because game objects are copied only on the first update. Thus a larger fraction of the updates avoid copying, and more of the objects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

Checkpoint and Recovery Times. Figure 2.4(b) shows the time to checkpoint as a function of the data skew, and Figure 2.4(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.

2.4.4 Experiments with Prototype Game Server



(a) Overhead time

(b) Time to checkpoint

(c) Recovery time

Figure 2.5: Overhead, checkpoint, and recovery times for a trace with 400,128 units and updates to 10% of the units every tick.

In this section we run our simulation with the update trace of the Knights and Archers Game described in Section 2.3.4. 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 2.5 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 2.5(b). Recall from Section 2.4.2 that all copy on update strategies exhibit more overhead in the first few ticks of 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 2.5(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 2.2(c) and 2.4(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 latter. 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.

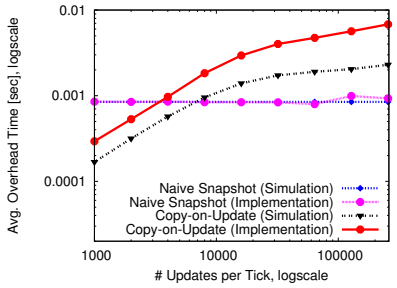
2.5 Experimental Validation

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.

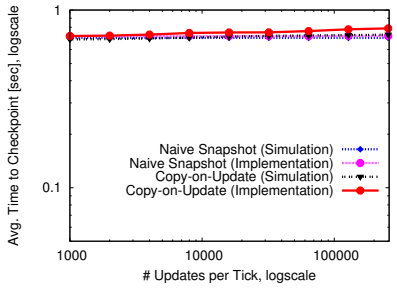
Validation Setup. We implemented Naive-Snapshot and Copy-on-Update in C++ and ran them on an Intel Core 2 Quad 2.4 GHz machine with 4MB cache and 4GB of main memory. The operating system used was Ubuntu Linux kernel 2.6.27-14. To avoid fragmentation at the filesystem level, we put the checkpoint files 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 2.4.1 for validation. Our implementation is driven by trace files with update distributions as described in Section 2.3.4. 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

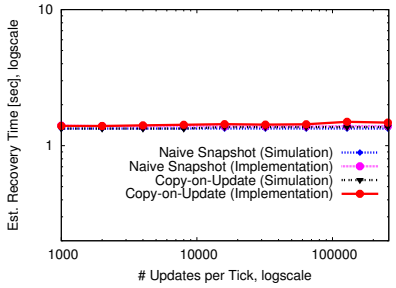
rate in order to fill a tick. It performs a sequence of random lookups in the game state. After the query phase is over, the *update phase* processes the updates from the trace for the given tick. We keep all trace files completely loaded in main memory to avoid introducing disk access delays in the 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.



(a) Updates per tick vs. overhead time



(b) Updates per tick vs. time to checkpoint



(c) Updates per tick vs. recovery time

Figure 2.6: 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.

Validation Results. Figure 2.6 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 2.3.3.

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.

2.6 Related Work

Several approaches have been proposed for taking database snapshots [2]. Kähler and Risnes explore how to use logging to refresh a snapshot, proposing a technique that resembles incremental view maintenance [60]. Other work has investigated how to provide efficient access to a large collection of past snapshots [104, 105]. 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 as-

sume 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 [69]. 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 [50, 98]. 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 [53]. 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 [123]. 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 [67] and Stonebraker et al. [109] advocate a distributed design in which no logging is performed by the system. In contrast to Whitney et al. [123], 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 [101]. 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.

2.7 Summary and Recommendations

In this dissertation, 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.

CHAPTER 3

FAST CHECKPOINT-RECOVERY ALGORITHMS FOR FREQUENTLY CONSISTENT APPLICATIONS

Repetition is the only form of permanence that nature can achieve.

– George Santayana

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 chapter, 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.

3.1 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 [109,

59].

Throughout the dissertation, 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 dissertation, we use *coordinated checkpointing* to provide durability for distributed FC applications at low cost [21]. Many applications already use replication to increase availability and provide some fault tolerance [109], 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 [34].

Before discussing requirements for checkpoint-recovery algorithms targeted at FC applications, we summarize the terminologies we use in this chapter:

- A *point of consistency* during an application execution is a time at which the state of the application is valid according to the semantics of the application. For example, in a single-threaded transactional processing system, an end of each transaction is a point of consistency.
- A *frequently consistent* (FC) application is a distributed application in which the state of each node frequently reaches a point of consistency. For example, new OLTP systems, such as H-Store, data-parallel systems, such as BOOM, and deterministic transaction processing systems, are FC applications.
- *Logical logging* is a logging technique which logs high level operations. For example, each time an application performs any action which changes its application state, logical logging only records the operation which made the changes, such as insert, update or delete.
- *Stable storage* indicates a data repository which guarantees that successfully written data is safe in the event of an application crash.
- A *Checkpoint period* is the time between start and end of a checkpoint.

3.1.1 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 [57]. This problem is made even worse when the checkpointing algorithm must pause the system in order to perform a synchronous operation like a bulk 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 [77] and fuzzy checkpointing [50, 98], 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 [46, 75, 94]. 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 in Chapter 2 for checkpointing MMOs on commodity hardware and discuss how it can be extended to support FC applications.

Interface 1: Algorithmic Framework
Mutator::PrepareForNextCheckpoint() **Mutator::PointOfConsistency()** **Mutator::HandleRead(index)** **Mutator::HandleWrite(index, update)**
AsynchronousWriter::WriteToStableStorage()

3.1.2 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 introduced in Chapter 2. 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 [115].

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 [99].

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 chapter, 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 [99].

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.

3.1.3 Existing Algorithms

Our experimental evaluation for MMOs concluded that there was no single checkpointing algorithm that outperformed all the others over the entire range of update rates [99]. Our evaluation included a number of synthetic experiments, and we believe they provide a reasonable model for many FC applications. We recommended 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.

3.2 New Algorithms

In this section, we present two novel checkpoint recovery algorithms for FC applications. We first discuss important design trade-offs that differentiate our algorithms from state-of-the-art methods (Section 3.2.1) and then introduce each algorithm in turn (Sections 3.2.2 and 3.2.3).

3.2.1 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.

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

Table 3.1: Overhead factors of checkpoint-recovery algorithms.

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 3.1 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. 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 pre-

empted within a checkpoint period. According to Herlihy, an implementation of a concurrent data object that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes, is called a **wait-free** implementation [52]. As a consequence, both of our algorithms are wait-free within a checkpoint period. Table 3.1 also summarizes the overhead factors our algorithms incur.

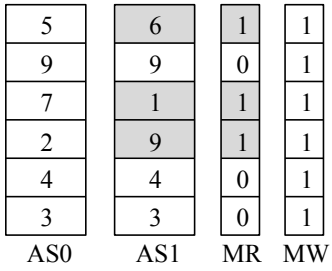
To eliminate overhead factors, both of our new algorithms strictly separate the state being updated by the Mutator from the state being read by the Asynchronous Writer. In addition, both track updates at very fine, word-level granularity. However, the algorithms use different amounts of main memory. 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.

3.2.2 Wait-Free Zigzag

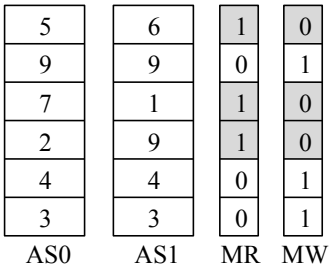
The main intuition behind Wait-Free Zigzag is to maintain an untouched copy of every word in the application state for the duration of a checkpoint period. These copies form the consistent image that is written to disk by the Asynchronous Writer. As these copies are never changed during the checkpoint 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 3.1(a)). For each word i in the application state, we maintain two bits: $MR[i]$

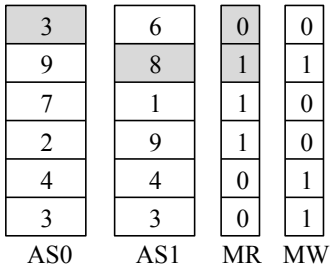
(a) At the beginning of time, AS_0 and AS_1 contain the same information



(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 3.1: Wait-Free Zigzag Example

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.

During a checkpoint period, Wait-Free Zigzag maintains an invariant that 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. These words form an untouched consistent copy of the application state. As such, 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 3.1(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, to maintain the invariant for the next 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 3.1(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 3.1(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 2 summarizes Wait-Free Zigzag. At the very first checkpoint, all the bits in MW are ones, so the Mutator only updates AS_1, and sets the MR bits accordingly. Meanwhile, the Asynchronous Writers can flush AS_0 to a stable storage. The Mutator can read values of the application state using the corresponding MR bits. At the end of this checkpoint, the Mutator pauses the application state, assigns the negation of MR to MW to maintain the invariant for the next checkpoint. The process repeats for the next checkpoint and so on.

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

Algorithm 2: Wait-Free Zigzag

input:

```
/* ApplicationState is a vector containing words */
```

ApplicationState $AS_0 \leftarrow$ *initial application state*

ApplicationState $AS_1 \leftarrow$ *initial application state*

$sizeWords \leftarrow |AS_0|$ `/* size of application state in words */`

BitArray $MR \leftarrow \{0,0,\ldots,0\}$ `/* reads from the Mutator reference` $AS_{MR[k]}$ `*/`

BitArray $MW \leftarrow \{1,1,\ldots,1\}$ `/* writes from the Mutator affect` $AS_{MW[k]}$ `*/`

Mutator::PrepareForNextCheckpoint()

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

Mutator::PointOfConsistency()

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

Mutator::HandleRead(index)

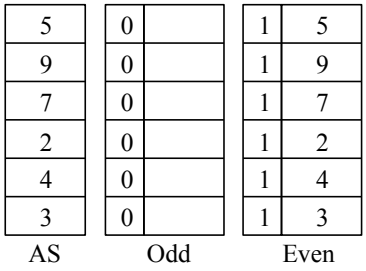
1: return $AS_{MR[index]}[index]$

Mutator::HandleWrite(index, newValue)

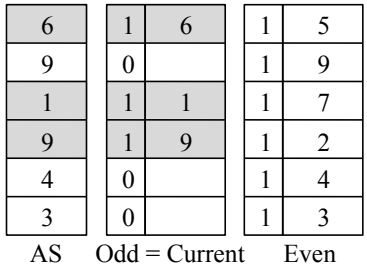
1: $AS_{MW[index]}[index] \leftarrow newValue$
2: $MR[index] \leftarrow MW[index]$

AsynchronousWriter::WriteToStableStorage()

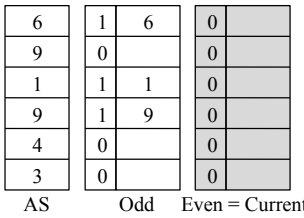
1: **loop**
2: $\quad$ WaitForMutatorNotification()
3: $\quad$ **for** $k = 0$ to $sizeWords$ **do**
4: $\quad\quad$ **write-to-disk** $AS_{\neg MW[k]}[k]$
5: $\quad$ **end for**
6: **end loop**



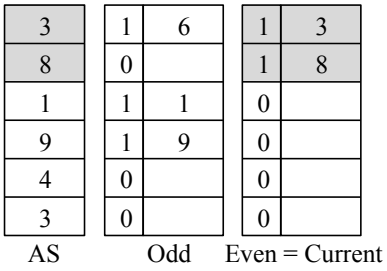
(a) At the beginning of time, *AS* and *Even* contain the same information



(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 3.2: Wait-Free Ping-Pong Example

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

3.2.3 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 3.2(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

Algorithm 3: Wait-Free Ping-Pong

input:

```
/* ApplicationState is vector containing words */
```

ApplicationState $AS \leftarrow$ *initial application state*

ApplicationState *currentAS*

ApplicationState *previousAS* $\leftarrow$ *initial application state*

$sizeWords \leftarrow |AS|$ `/* size of application state in words */`

BitArray $currentBA \leftarrow \{0,0,\ldots,0\}$ `/* current checkpoint dirty words */`

BitArray $previousBA \leftarrow \{1,1,\ldots,1\}$ `/* last checkpoint dirty words */`

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
```

also applied to *Odd*. The situation after a few mutator updates is shown in Figure 3.2(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 3.2(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 3.2(d) displays the situation after a few updates from the Mutator in the second checkpoint period.

Algorithm 3 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 flush-

ing 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 3.4.5

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.

3.3 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.

3.3.1 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.

3.3.2 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 3.2.2. 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 [3]. 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 3.4.2). 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.

3.3.3 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 3.4.6.

3.4 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.

3.4.1 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 [109], 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 [109], 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 [77]. 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 5.2 for our algorithms (Section 3.4.2). 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 3.4.3 and 3.4.4). Then, we report on how our methods affect application throughput in our TPC-C application (Section 3.4.5). Finally, we investigate the impact of a further optimization, using large pages, on the relative performance of all algorithms (Section 3.4.6).

3.4.2 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 [55].

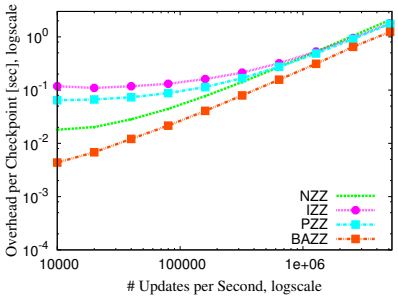


Figure 3.3: Wait-Free Zigzag Overhead

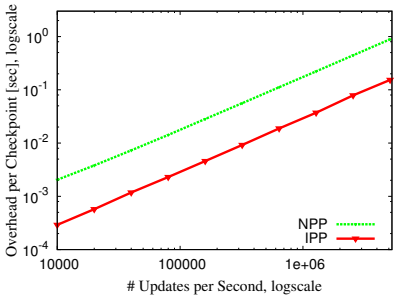


Figure 3.4: Wait-Free Ping-Pong Overhead

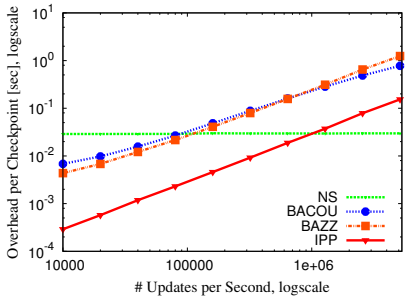


Figure 3.5: Zipf workload: Overhead

Table 3.2 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.

Wait-Free Zigzag

Figure 3.3 shows the average overhead per checkpoint period of the different variants. Both IZZ and PZZ are less efficient than NZZ, 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 3.2 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 3.3 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

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 3.2: Profiling on synthetic workload, 320K updates/sec

of IZZ.

Wait-Free Ping-Pong

Figure 3.4 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 3.2 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.

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

3.4.3 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 5.2. 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 3.5 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 [99] – 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 [99], 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 [99]. 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 3.6 and 3.7 show the overhead per checkpoint period for two different update rates. In each case, we scale the update rate with the state size,

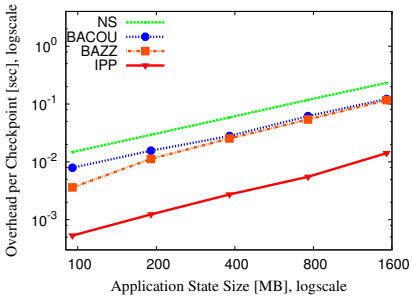


Figure 3.6: Scaleup: 0.08% updates/sec

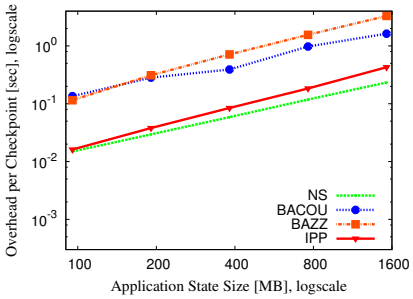


Figure 3.7: Scaleup: 2.56% updates/sec

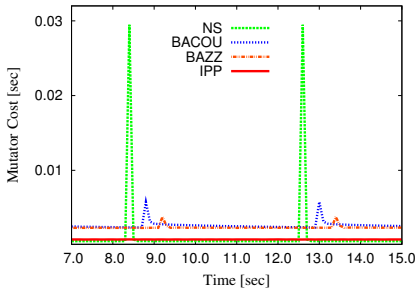


Figure 3.8: Latency: 320K updates/sec

so in Figure 3.6, 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 3.7 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 3.6), 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 3.7), 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 3.1.1, we also want the overhead to be uniformly distributed over time. Figures 3.8 and 3.9 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 3.8) 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 3.5 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 3.9 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 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 3.8 and 3.9 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 3.1.1.

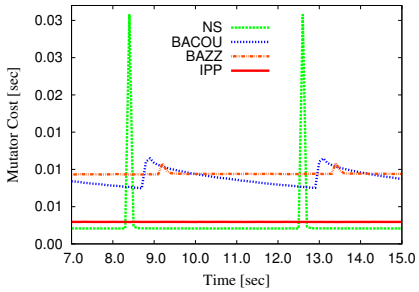


Figure 3.9: Latency: 1,280K updates/sec

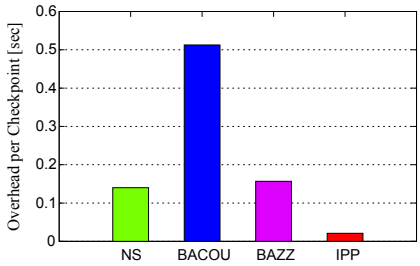


Figure 3.10: MMO: Overhead

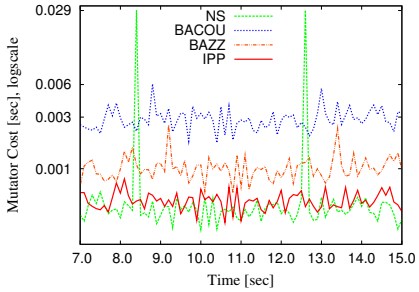


Figure 3.11: MMO: Latency

3.4.4 Synthetic MMO Workload

We also ran our experiments on a trace produced using our MMO workload. Figure 3.10 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 3.11 shows the performance of each algorithm in the MMO simulation over time. The Mutator cost is quite variable compared to Figure 3.8, 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 [28], 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.

3.4.5 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 [117]. 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 3.12 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 Section3.2.3 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.

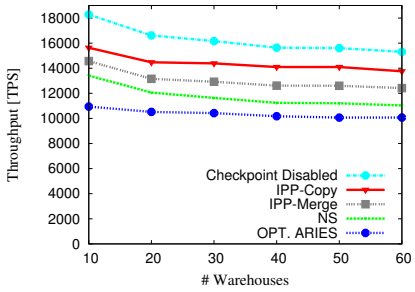


Figure 3.12: TPC-C Throughput

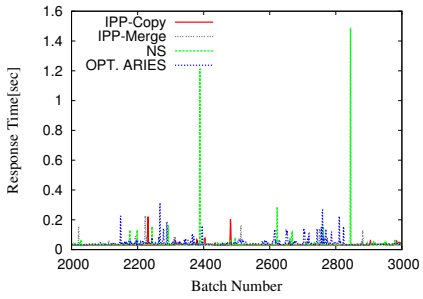


Figure 3.13: TPC-C Latency

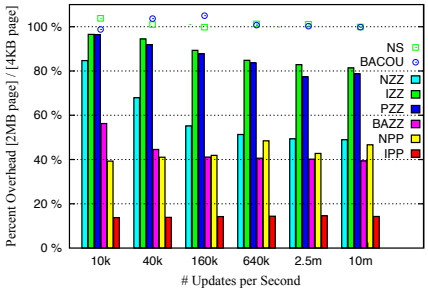


Figure 3.14: Large Page Overhead

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 3.13 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 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.

3.4.6 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 3.14 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 3.2). 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.

3.5 Related Work

There has been extensive work in checkpointing algorithms for main-memory DBMSs [33, 92, 97, 98, 123]. Recently, we have evaluated the performance of these algorithms for MMO workloads [99]. 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 [71, 87] or application level [12, 16]. Additionally, checkpointing may be offered to

applications embedded in a language runtime [127], through a library [87], or via compiler support [16]. Our work performs application-level checkpointing and integrates with the application logic through a library API (Section 3.1.2). 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 [77]. As we have seen in Section 3.4.5, approaches that necessitate physical logging, such as ARIES or fuzzy checkpointing [50, 98], 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 [11, 53]. Recently, Lau and Madden [67] and Stonebraker et al. [109] 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 [119]) also include checkpointing in order to ensure durability in the event that all replicas fail (e.g., due to a power outage).

3.6 Conclusions

In this chapter, 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.

CHAPTER 4

OPTIMIZED CHECKPOINT-RECOVERY ALGORITHMS

Life is like riding a bicycle. To keep your balance, you must keep moving.

– Albert Einstein

Checkpoint-recovery is a crucially important technique to provide fault-tolerance to main-memory applications. As discussed in the previous chapter, we can take advantage of frequent *points of consistency* in many of modern main-memory applications to develop novel checkpoint recovery algorithms that trade additional space in main memory for significantly lower overhead and latency. In particular, Wait-Free Ping-Pong uses three copies of the application state to achieve almost an order of magnitude lower overhead and nearly constant latency. These algorithms work best for applications which have small application states, such as MMOs, whose sizes are in a range of hundreds of megabytes.

The previous chapter has only scratched the surface of the design space of checkpoint-recovery methods to avoid locking and bulk copying. In this chapter, we present a series of optimization techniques which exploit modern computer architectures to reduce memory usage as well as to reduce synchronization overhead.

4.1 Background

A common feature of modern main-memory applications is that there are clear points in time at which the application state is consistent. We consider OLTP systems as an example. OLTP systems used to be heavily multi-threaded since they had to mask the huge differences between access times to main memory versus disk. They sacrificed points of

consistency in order to mask latency differences, but the required concurrency control and recovery subsystem imposed significant overhead [51]. When OLTP systems move to main memory, this overhead and complexity can be avoided if transactions are just completely serialized [109]. In addition, given natural restrictions on OLTP transactions, it is possible to isolate their reads and writes to avoid coordination among multiple processors or nodes. The end of each transaction (or group of transactions) marks a natural point of consistency in these systems [118].

Such points of consistency exist in many other types of data-driven systems. Search engines and data warehouse systems both keep read-optimized data structures for performance. They do not directly apply updates to these data structures, but rather evolve them by a differential indexing approach [103]. The merge of a new read-optimized data structure in main memory marks a natural point of consistency. MMOs and large-scale simulations follow a time-stepped processing model. Here, the effects of all actions of characters are determined inside of a time step, or tick, and state updates are applied at the end of a tick by aggregating the effects over all characters [107, 122]. The end of each tick is again a natural point of consistency.

Data mining applications that can be implemented through iterated MapReduce computations in main memory [25] are yet another example. This processing model exhibits a natural point of consistency at the end of each MapReduce step [32]. The same observation can be made about the time-stepped dataflow processing model proposed by BOOM/Overlog [5, 27]. These processing models hold promise as new abstractions to write a large number of future data-driven applications in the cloud.

In Chapter 2, we evaluated several existing main memory checkpointing algorithms to persist the large number of non-transactional updates in MMOs. We recommended two recovery algorithms: Copy-on-Update for low to moderate update rates, and Naive

Snapshot for extremely high update rates. These algorithms try to balance the overhead of checkpointing with the latency disruption caused by synchronous memory operations. In order to avoid latency spikes, Copy-on-Update utilizes copy-on-write and requires locking portions of the game state. Though this works for very low update rates, lock contention becomes a major problem as the update rate increases. On the other hand, Naive Snapshot involves large synchronous copy operations, and thus introduces large latency spikes.

To avoid locking and synchronous copying, we designed two new algorithms: Wait-Free Zigzag and Wait-Free Ping-Pong. As their names suggest, neither of these algorithms requires locking, and both are designed to reduce latency spikes. Wait-Free Ping-Pong takes advantage of the fact that the dynamic state is usually quite small, in a range of a few hundred megabytes, so that it can use additional memory to reduce latency. Wait-Free Zigzag is designed for more memory constrained environments, and achieves overhead similar to existing algorithms with considerably smaller latency spikes.

In this chapter, we explore the impact of data layout in memory on cache performance and present a series of low level optimizations. We find that our algorithms are particularly amenable to these optimizations.

4.2 Optimizations for Copy-On-Update

In this section, we present the optimization techniques for Copy-On-Update.

4.2.1 Copy-On-Update Revisited

The logic underlying Copy-On-Update is to group objects into blocks. During a checkpoint period, when an object is updated, the block containing this object, if not yet copied, is copied to the shadow state. There are two approaches to implement this logic; system-level approach and application-level approach.

System-Level Copy-On-Update

System-level Copy-On-Update is widely used in all modern operating systems. This approach uses page-level hardware-supported virtual memory snappshoting to create consistent-snapshots. In particular, an Asynchronous Writer process is forked from the Mutator process at a point of consistency. The forked child process obtains an exact copy of the parent address space. The snapshot captures the state at the point of consistency when the *fork* system call is invoked. When an object is updated, the hardware-supported copy-on-update mechanism replicates the virtual memory page where the object resides. Replicating a page is highly efficient, taking only 2 micro-second to replicate a 4kb page [62]. Since the replication is carried out at the granularity of entire pages, which usually have a default size of 4kb, it is difficult for system-level copy-on-update to exploit memory layout to improve caching performance.

Application-Level Copy-On-Update

Algorithm 4 shows the pseudo-code of application-level Copy-on-Update. A set of bit arrays is used to keep track of which blocks have been updated during the current checkpoint period. The Mutator must check this bit array every time it updates an object. Meanwhile, to bring the checkpoint up to date, the Asynchronous Writer

Algorithm 4: Copy-On-Update algorithm

input:

```
/* ApplicationState is vector of blocks */
```

ApplicationState *theApplicationState*

ApplicationState *shadowApplicationState*

numBlocks $\leftarrow \frac{|theApplicationState|}{BLOCKSIZE}$ `/* number of blocks */`

BitArray *currentBA* `/* current checkpoint dirty bits */`

BitArray *previousBA* `/* last checkpoint dirty bits */`

BitArray *changedBA* `/* last two checkpoints dirty bits*/`

Mutator::PrepareForNextCheckpoint()

1: **for** $i = 0$ to *numBlocks* **do**
2: $changedBA[i] \leftarrow currentBA[i] \mid previousBA[i]$
3: $previousBA[i] \leftarrow currentBA[i]$
4: $currentBA[i] \leftarrow 0$
5: **end for**

Mutator::EndOfTick()

1: **if** asynchronous writer done **then**
2: PrepareForNextCheckpoint()
3: NotifyAsynchronousWriter()
4: **end if**

Mutator::HandleWrite(index, update)

1: **if** $\neg currentBA[index]$ **then**
2: **lock** $theApplicationState[index]$
3: **if** changedBA[index] **then**
4: $shadowApplicationState[index] \leftarrow theApplicationState[index]$
5: **end if**
6: $currentBA[index] \leftarrow 1$
7: **unlock** $theApplicationState[index]$
8: **end if**
9: $theApplicationState[index] \leftarrow update$

Algorithm 5: Copy-On-Update algorithm (Continue)

AsynchronousWriter::WriteToStableStorage()

```
1:  loop
2:    WaitForMutatorNotification()
3:    if it is the first checkpoint then
4:      write-to-disk theApplicationState
5:    else
6:      for k = 0 to numBlocks do
7:        if changedBA[k] then
8:          lock theApplicationState[k]
9:          if currentBA[k] then
10:           unlock theApplicationState[k]
11:           write-to-disk shadowApplicationState[k]
12:         else
13:           write-to-disk theApplicationState[k]
14:           unlock theApplicationState[k]
15:         end if
16:       end if
17:     end for
18:   end if
19: end loop
```

either reads from the application state or the shadow state depending on whether the corresponding block has been updated. Since the Mutator is updating the application state concurrently, it must be isolated from the Asynchronous Writer using locks on the blocks it references. Note that this may introduce significant overhead if the number of updates to the application state is large. By varying the block size, Copy-on-Update can tradeoff between copying and locking overhead.

In the remaining of this section, we present a set of optimization techniques to make application-level Copy-On-Update lock-free.

4.2.2 Observations

We observe that the Asynchronous Writer always writes to the stable storage sequentially. So, if the Asynchronous Writer is currently working on block i, then all the blocks whose indexes are less than i have already been written. As a result, the Mutator can freely update those blocks without acquiring a lock. The only information the Mutator needs to know is the block index which the Asynchronous Writer is currently working on. Hence, we can simply use less expensive operations, e.g. using memory barriers or atomic Test-and-Set instruction, to synchronize the Mutator and the Asynchronous Writer, completely removing the use of expensive locking.

In modern computer architecture, independent memory operations are effectively performed in random order; for example, *writes* can be seen out-of-order, *reads* can be speculative and return future values. In order to impose some orderings, program developers have to use some types of memory-barriers. There are four basic types of memory-barriers [76].

1. **Store barrier** A store barrier gives a guarantee that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system. A store barrier is a partial ordering on stores only; it is not required to have any effect on loads.
2. **Data dependency barrier** A data dependency barrier gives a guarantee that if two loads are performed such that the second depends on the result of the first, e.g. the

first load retrieves the address to which the second load will be directed, then the target of the second load is updated before the address obtained by the first load is accessed. A data dependency barrier is a partial ordering on interdependent loads only; it is not required to have any effect on stores, independent loads or overlapping loads.

3. **Load barrier** A load barrier is a data dependency barrier plus a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system. A load barrier is a partial ordering on loads only; it is not required to have any effect on stores.

4. **Full barrier** A full memory barrier gives a guarantee that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system. A full memory barrier is a partial ordering over both loads and stores.

4.2.3 Copy-On-Update with Naive Memory-Barriers

Algorithm 6 and Algorithm 7 describe a naive usage of memory-barriers to coordinate the Mutator and the Asynchronous Writer. The key design is to assign each block a status value; zero for not being updated, one for being updated and two for already updated by the Mutator.

Initially, the status values are all zeros. Before applying the first update to a block which has not been copied by the Asynchronous Writer, the Mutator sets the status bit to one and copies the block to the Shadow Copy. After the copying is done, the Mutator

Algorithm 6: MB-Copy-On-Update algorithm

input:

```
/* ApplicationState is vector of blocks */
```

ApplicationState *theApplicationState*

ApplicationState *shadowApplicationState*

numAtomicBlocks $\leftarrow \frac{|theApplicationState|}{BLOCKSIZE}$ `/* number of blocks */`

StatusArray *statusArray* `/* 0:unchanged,1:being updated,2:updated */`

iW $\leftarrow 0$ `/* the async writer's loop index */`

LDBAR `/* Load Barrier */`

STBAR `/* Store Barrier */`

AsynchronousWriter::WriteToStableStorage()

1: **loop**
2: WaitForMutatorNotification()
3: **for** $k = 0$ to *numBlocks* **do**
4: **if** $statusArray[iW] == 0$ **then**
5: LDBAR
6: **write-to-backup** $theApplicationState[iW]$
7: LDBAR
8: **end if**
9: **if** $statusArray[iW] != 0$ **then**
10: **while** $statusArray[iW] == 1$ **do**
11: no-op
12: **end while**
13: LDBAR
14: **write-to-backup** $shadowApplicationState[k]$
15: **end if**
16: LDBAR
17: increment *iW*
18: **end for**
19: **end loop**

Algorithm 7: MB-Copy-On-Update algorithm(Continue)

Mutator::PrepareForNextCheckpoint()

```
1: for i = 0 to numBlocks do
2:     clear statusArray[i]
3: end for
```

Mutator::EndOfTick()

```
1: if asynchronous writer done then
2:     PrepareForNextCheckpoint()
3:     NotifyAsynchronousWriter()
4: end if
```

Mutator::HandleWrite(index, update)

```
1: blockIndex = index * sizeof(int)/BLOCKSIZE
2: if blockIndex >= iW then
3:     LDBAR
4:     if statusArray[blockIndex] == 0 then
5:         statusArray[blockIndex] ← 1
6:         STBAR
7:         block-copy
           shadowApplicationState[blockIndex] ← theApplicationState[blockIndex]
8:         STBAR
9:         statusArray[blockIndex] ← 2
10:    end if
11: end if
12: STBAR
13: theApplicationState[index] ← update
```

sets the status bit to two. Store-barriers are put between these steps to ensure this order of execution.

Concurrently, the Asynchronous Writer iterates through all the blocks, checks the status value of each block to decide whether it should read the block from the Application State or from the Shadow Copy. If the status of a block is zero, the Asynchronous Writer reads from the Application State. It is possible that during this time, the Mutator also updates this block. Hence, in order to copy the correct block, the Asynchronous Writer has to wait for the Mutator to finish its update; i.e. copies the block from the Application State to the Shadow Copy and then changes the block status to two and applies the update. The Asynchronous Writer achieves that by spinning checking the status value if it has been changed to two. And then, it reads the block from the Shadow Copy and writes it to the stable storage.

4.2.4 Copy-On-Update with Atomic Operators

Algorithm 8 and Algorithm 9 describe an enhanced version of Copy-On-Update using the Test-And-Set instruction.

In this version of the algorithm, each block is associated with a dirty bit. This dirty bit serves as a token for the Mutator and the Asynchronous Writer; who has the token is responsible to copy the block from the Application State to the Shadow Copy. The Test-And-Set instruction guarantees that if there is race between the two threads, only one of them will get it.

However, simply obtaining the dirty bit may not guarantee that the Asynchronous Writer will copy a consistent block since the Mutator can apply the update to that block

Algorithm 8: TaS-Copy-On-Update algorithm

input:

```
/* ApplicationState is vector of blocks */
```

ApplicationState $theApplicationState$

ApplicationState $shadowApplicationState$

$numAtomicBlocks \leftarrow \frac{|theApplicationState|}{BLOCKSIZE}$ `/* number of blocks */`

DirtyBitArray $D[2]$ `/* dirty bit arrays */`

$current \leftarrow 0$ `/* indicate the current working bit array */`

Mutator::PrepareForNextCheckpoint()

1: $current \leftarrow 1 - current$

Mutator::EndOfTick()

1: **if** asynchronous writer done **then**
2: PrepareForNextCheckpoint()
3: NotifyAsynchronousWriter()
4: **else**
5: **if** isFlagSet() **then**
6: chkptNum++
7: STBAR
8: resetFlag()
9: signal the AsyncWriter to flush the shadowApplicationState
10: **end if**
11: **end if**

Mutator::HandleWrite(index, update)

1: $blockIndex = index * sizeof(int)/BLOCKSIZE$
2: **if** $TaS(D[current][blockIndex]) == 0$ **then**
3: **block-copy** $shadowApplicationState[blockIndex] \leftarrow theApplicationState[blockIndex]$
4: **end if**
5: $theApplicationState[index] \leftarrow update$

Algorithm 9: TaS-Copy-On-Update algorithm(Continue)

AsynchronousWriter::WriteToStableStorage()

```
1:  loop
2:    WaitForMutatorNotification()
3:    for i = 0 to numBlocks do
4:      if D[current][i] == 1 then
5:        continue
6:      else
7:        block-copy tempBuf ← theApplicationState[i]
8:        if TaS(D[current][i]) == 0 then
9:          block-copy shadowApplicationState[i] ← tempBuf
10:       end if
11:     end if
12:   end for
13:   Clear the dirty bit array D[1-current]
14:   setFlagAndWait() /*inform the Mutator */
15:   Flush the shadowApplicationState
16: end loop
```

concurrently. To avoid this problem, the Asynchronous Writer always copies the block to its temporary buffer and then tries to obtain the dirty bit. If the dirty bit is not set, then the temporary copy is a consistent copy of that block. Therefore, the Asynchronous Writer can safely move the temporary copy to the Shadow Copy. If the dirty bit is set, then the Mutator is responsible to copy the block to the Shadow Copy.

After iterating through the application state, the Asynchronous Writer waits for a signal from the Mutator to ensure that the Mutator has finished copying all the blocks it is responsible for. This wait and signal mechanism can be done at the end of a tick, and hence, introducing little overhead to the Mutator.

Since the Test-And-Set operator may implicitly introduce some memory fences, we can reduce the number of Test-And-Set operations in the Mutator by simply checking the dirty bit with normal *if* statement. This optimization is achieved by adding $if(D[current][blockIndex] == 0)$ between the line 1 and 2 in the Mutator.

4.2.5 Copy-On-Update with Memory-Barriers Simulated Test-And-Set operation

This version of the algorithm simulates the Copy-On-Update with Atomic Operators using memory-barriers. Algorithm 10 and Algorithm 11 present this implementation, where store-barriers are used to enforce the execution order.

4.3 Optimizations for Wait-Free algorithms

The previous chapter presented two Wait-Free algorithms which eliminate locking and bulk bit manipulation. These Wait-Free algorithms exhibit significant lower overhead and achieve nearly constant latency. However, the best algorithm uses 3x copies of the applications. In this section, we present a set of optimizations to reduce the memory usage while maintaining a comparable performance and latency to the best existing algorithm.

Algorithm 10: MB Simulated TaS-Copy-On-Update algorithm

input:

```
/* ApplicationState is vector of blocks */
```

ApplicationState $theApplicationState$

ApplicationState $shadowApplicationState$

$numBlocks \leftarrow \frac{|theApplicationState|}{BLOCKSIZE}$ `/* number of blocks */`

DirtyBitArray $D[2]$ `/* dirty bit arrays */`

$current \leftarrow 0$ `/* indicate the current working bit array */`

Mutator::PrepareForNextCheckpoint()

1: $current \leftarrow 1 - current$

Mutator::EndOfTick()

1: **if** asynchronous writer done **then**
2: PrepareForNextCheckpoint()
3: NotifyAsynchronousWriter()
4: **else**
5: **if** isFlagSet() **then**
6: chkptNum++
7: STBAR
8: resetFlag()
9: signal the AsyncWriter to flush the shadowApplicationState
10: **end if**
11: **end if**

Mutator::HandleWrite(index, update)

1: $blockIndex = index * sizeof(int)/BLOCKSIZE$
2: **if** $D[current][blockIndex] == 0$ **then**
3: $D[current][blockIndex] = 1$
4: STBAR
5: **block-copy** $shadowApplicationState[blockIndex] \leftarrow theApplicationState[blockIndex]$
6: **end if**
7: $theApplicationState[index] \leftarrow update$

Algorithm 11: MB Simulated TaS-Copy-On-Update algorithm

AsynchronousWriter::WriteToStableStorage()

```
1:  loop
2:    WaitForMutatorNotification()
3:    for i = 0 to numBlocks do
4:      if D[current][i] == 1 then
5:        continue
6:      else
7:        block-copy tempBuf ← theApplicationState[i]
8:        LDBAR
9:        if D[current][i] == 0 then
10:         D[current][blockIndex] = 1
11:         block-copy shadowApplicationState[i] ← tempBuf
12:       end if
13:     end if
14:   end for
15:   Clear the dirty bit array D[1-current]
16:   setFlagAndWait() /*inform the Mutator */
17:   Flush the shadowApplicationState
18: end loop
```

4.3.1 Observations

Wait-Free Ping-Pong exhibits excellent low overhead and low latency compared to other algorithms for a wide range of update rates. However, Wait-Free Ping-Pong uses 3x the amount of memory usage. The Mutator always reads and writes to the main application state. At the same time, all the writes are replicated both at the main application state and one of the two copies; Wait-Free Ping-Pong can always determine which copies it needs to write to. Therefore, the main application state is practically used for reading

the latest data. So, it may be possible to use extra bit arrays to keep track of which copy has the latest data in order to eliminate the main copy of the application state.

Wait-Free Zigzag requires two copies of the application states. However, it incurs higher overhead since the mutator has to *zigzagly* write and read from these two copies. As a result, the room to exploit special memory structure to layout these two copies is limited. In fact, the previous chapter showed that the best implementation of Wait-Free Zigzag is to optimize the negation of the bit-array and implement each copy of the application as a continuous array, i.e. no memory interleaving. Therefore, it is greatly beneficial if the Mutator just access write to a single copy and leverage a some meta data to track the latest data for reading.

4.3.2 Optimized-Write Wait-Free Algorithm

Optimized-Write keeps two copies of the application state, namely AS_0 and AS_1. During a checkpoint period, the Mutator only writes to one of the copies of the application state. Optimized-Write maintains a current flag indicating which copy is being updated by the Mutator during a checkpoint period.

To keep track of the updates, OptimizedWrite maintains two dirty bit-arrays, BA_0 and BA_1, each is associated with one of the application state copy, AS_0 and AS_1 respectively. Whenever the Mutator updates a word, it also sets the corresponding dirty-bit.

The key design is that OptimizedWrite maintains two stable copies, corresponding to AS_0 and AS_1, on a stable storage and propagates the update stream to the stable storage so that in the event of a crash, it knows which stable copy has the latest values.

In addition, OptimizedWrite maintains two more bit-arrays, namely *asValid* and

Algorithm 12: Optimized-Write Wait-Free Algorithm

input:

```
/* ApplicationState is a vector containing words */
```

ApplicationState $AS_0 \leftarrow$ *initial application state*

ApplicationState $AS_1 \leftarrow$ *initial application state*

$sizeWords \leftarrow |AS_0|$ `/* size of application state in words */`

BitArray $asValid \leftarrow \{0,0,\ldots,0\}$ `/* keeps track of which copy contains the latest data*/`

BitArray $ckptValid \leftarrow \{0,0,\ldots,0\}$ `/* keeps track of which ckpt contains the latest data*/`

BitArray $BA[2] \leftarrow \{\{0,0,\ldots,0\},\{0,0,\ldots,0\}\}$ `/* dirty words bit arrays */`

$current \leftarrow 0$ `/* the copy to be written by Mutator */`

Mutator::PrepareForNextCheckpoint()

1: $current = 1 - current$

Mutator::PointOfConsistency()

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

Mutator::HandleRead(index)

1: return $AS[asValid[index]][index]$

Mutator::HandleWrite(index, newValue)

1: $AS[current][index] = newValue$;
2: $BA[current][index] = 1$;
3: $asValid[index] = current$;

Algorithm 13: Optimized-Write Wait-Free Algorithm(Continue)

AsynchronousWriter::WriteToStableStorage()

```
1:  loop
2:    WaitForMutatorNotification()
3:    Bit which = 1 − current;
4:    for i = 0 to sizeWords do
5:      if BA[which][i] then
6:        ckptValid[i] = which;
7:        BA[which][i] = 0;
8:      end if
9:    end for
10:   write-to-disk AS[which]
11:   write-to-disk ckptValid
12: end loop
```

ckptValid.

The *asValid* is only updated by Mutator. Whenever the Mutator updates a word of the application state, it sets the corresponding bit in the *asValid* bit-array so that the next read of that word will get the latest value.

The *ckptValid* is only accessed by the AsynchronousWriter. At the beginning of the checkpoint period, the AsynchronousWriter merges all the updates happened in the previous checkpoint period into this *ckptValid* bit-array. This operation is feasible since the Mutator only writes to one of the copies; consequently, the dirty-bit array of that copy records all the updates. In particular, at the beginning of a checkpoint period, if the Mutator writes to AS_0, then the AsynchronousWriter merges all the bit set from the dirty-bit array of AS_1 to *ckptValid*, and simultaneously clears that dirty-bit.

The AsynchronousWriter also flushes the copy which is currently not being updated

by the Mutator to the stable storage.

As the result, this technique uses two copies of the application state and four bit-arrays, namely BA_0, BA_1, $asValid$, and $ckptValid$. On the stable storage, it maintains two intact checkpoint files (for AS_0 and AS_1) and one intact bit-array file (for $ckptValid$), one working application-state copy file (temporarily receiving the flush from either AS_0 or AS_1) and one working bit-array file (temporarily receiving the flush from $ckptValid$).

4.4 Implementation

We leverage the modern computer architecture to implement the optimizations introduced in the previous section. In particular, we leverage low-level operations and exploit different memory layouts to improve cache performance.

4.4.1 Optimized Copy-On-Update

Since System-Level Copy-On-Update operates at page granularity, it is hard to derive cache granularity implementation. Therefore, this section focuses on cache-optimized implementation for Application-Level Copy-On-Update. Our key design for these optimizations is to use low-overhead memory-barriers instead of locking to synchronize the operations between the Mutator and the Asynchronous Writer. There are two standard ways to implement a memory barrier; using gcc-atomic-builtins and using asm-operators.

1. **gcc-atomic-buitins** GCC version 4.x or higher provides a list of builtin functions for atomic memory access. However, these functions apply only on a limited set

of primitive types, such as *int*, *long*, *long long* and their unsigned counterparts. In addition, they allow only integral scalar or pointer type that is 1, 2, 4 and 8 bytes in length. According to the GNU specification, in most cases, these builtins are considered a full barrier. That is, no memory operand will be moved across the operation, either forward or backward. Further, instructions will be issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation [39].

2. **asm-operators** Instructions can be reordered at different levels; at compiler level and at hardware(CPU) level. In a general case, memory operation ordering is very relaxed, and a CPU may actually perform the memory operations in any order it likes, provided program causality appears to be maintained. Similarly, the compiler may also arrange the instructions it emits in any order it likes, provided it doesn't affect the apparent operation of the program. As a result, the Linux kernel (version 3.7 and above) provides different barriers that act at different levels; i.e. a compiler barrier which prevents the compiler from moving the memory accesses across it and basic CPU memory barriers which prevent the CPU to reorder instructions. Moreover, the Linux kernel enforces that the CPU memory barriers also imply a compiler barrier [76]. Therefore, we only use CPU memory barriers in our implementation. In particular, we use the GNU inline assembler statements, *asm volatile("mfence" :::"memory")*, *asm volatile("lfence" ::: "memory")*, and *asm volatile("sfence" ::: "memory")*, for full barrier, load barrier and store barrier respectively.

Copy-On-Update with Naive Memory-Barriers

We implement this algorithm using full memory-barriers instead of fine-grained load and store barriers. We also implement two versions, one uses *gcc-atomic-builtins*, i.e. $\_\_sync\_synchronize(...)$, and the other uses *asm-operators*, i.e. *asm volatile("mfence" :::"memory")*, to invoke a full memory barrier.

Copy-On-Update with Atomic Operators

We implement the Test-and-Set operator using gcc-atomic-builtins. In particular, we use the $\_\_sync\_lock\_test\_and\_set$ builtin. This builtin is not a traditional test-and-set operation, but rather an atomic exchange operation [39]. We also implement a version in which we use the gcc-atomic-builtin Compare-and-Swap, i.e. $\_\_sync\_bool\_compare\_and\_swap$, to represent the Test-and-Set operator.

Copy-On-Update with Memory-Barriers Simulated Test-And-Set Operation

We implement this algorithm using fine-grained load and store barriers. In particular, we use *asm volatile("lfence" :::"memory")* and *asm volatile("sfence" :::"memory")* to invoke a load barrier and a store barrier respectively.

4.4.2 Optimized-Write Wait-Free Algorithm

The key design for this algorithm is to keep two copies of the application state as contiguous arrays so that the Mutator has a direct access to the state it wants to update. This design exhibits two advantages. First, it preserves the data locality as designed by the

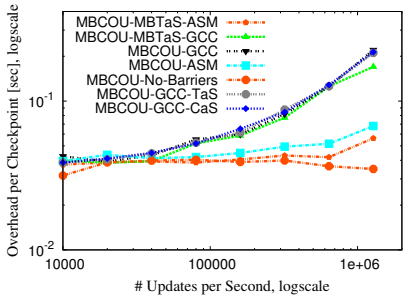


Figure 4.1: Optimized Copy-On-Update Overhead

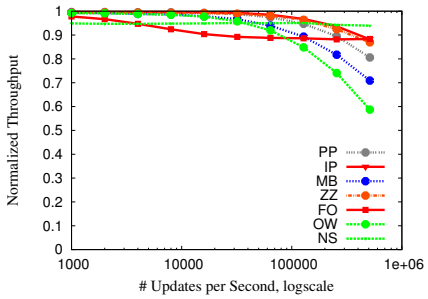


Figure 4.2: Throughput Comparison

application developer. Second, the Asynchronous Writer can flush the copy which is not currently being updated by the Mutator sequentially to a stable storage. To conform with this design, we implement the two copies of the application state as separate arrays. Unlike the Wait-Free Ping-Pong bit array implementation, which is interleaved with the application states, we keep the *asValid* bit array as a contiguous array to maintain a direct access for the Mutator.

The recovery procedure for this algorithm is substantially different from other algorithms, which requires reading two checkpoints in order to recover the most recent consistent state. We leave this implementation as a future work.

4.5 Experiments

In this section we compare the performance of Optimized Copy-On-Update algorithm with existing algorithms. We mainly use two metrics in our comparison; added overhead and normalized throughput. We also compare the best implementation of Optimized Copy-On-Update and Optimized-Write Wait-Free algorithm with the existing algorithms. We also include Fork-Copy-On-Update, a System-Level Copy-On-Update using the *fork* system call in our comparison.

4.5.1 Setup and Datasets

We use the similar synthetic workloads which was described in the previous chapter. In particular, we use the workload which models an application state of 8 KB objects. This application state contains 25,000 rows, corresponding to approximately 200 MB state size. Updates are selected according to a Zipf-distribution with the skew parameter $\alpha = 0.2$. We also group updates into intervals that correspond to 0.1 seconds of simulated application logic. Our workload generator produces traces with various number of updates per interval. These traces are loaded into main-memory before testing the algorithms. Each trace file contains a minimum of 50 s. of data.

We first compare all the variants of Optimized Copy-On-Update algorithms introduced in Section 4.4. We then select the best variant of Optimized Copy-On-Update to compare the added overhead with existing algorithms. Finally, to better understand how this added overhead affects the overall performance of an application, we measure the throughput of the most selective algorithms. In the throughput measurement experiment, we also enable the Asynchronous Writer.

We ran all our experiments on a local Quad-core Intel Xeon 5400 series processor with 32GB of RAM running Centos 5.4. The machine has 32 KB L1 cache and 12 MB L2 cache for each core and no L3 cache. The disk bandwidth in this server is roughly 100 MB/s and the memory bandwidth is around 1 GB/s.

4.5.2 Optimized Copy-On-Update Implementation Variants

Figure 4.1 shows the overhead per checkpoint period of different variants.

- **MBCOU-No-Barriers**. In this variant, we define all the barriers as no-op. This variant demonstrates the best possible performance of Optimized Copy-On-Update.
- **MBCOU-GCC** and **MBCOU-ASM**. These are the implementations of Optimized Copy-On-Update with Naive Memory-Barriers, in which the barriers are implemented using gcc-atomic-builtins and asm-operators respectively.
- **MBCOU-GCC-TaS** and **MBCOU-GCC-CaS**. These are the implementations of Optimized Copy-On-Update with Atomic Operators, in which the atomic operators are implemented using gcc-atomic-builtins test-and-set and compare-and-swap respectively.
- **MBCOU-MBTaS-GCC** and **MBCOU-MBTaS-ASM**. These are the implementations of Optimized Copy-On-Update with Memory-Barriers Simulated Test-And-Set Operation, in which the barriers are implemented using gcc-atomic-builtins and asm-operators respectively.

As the Figure 4.1 clearly shows, all the algorithms using gcc-atomic-builtins have higher overhead than algorithms using asm-operators for a wide range of update rates.

The main reason is that gcc-atomic-builtins use instructions with a lock prefix, e.g. lock cmpxchgb, to implement the atomic operators. These lock prefix instructions lock the system bus which prevent other cores from using it while the operations are taking place.

Among the algorithms using asm-operators, the MBCOU-MBTaS-ASM algorithm is the best algorithm; it is slightly less efficient than MBCOU-No-Barriers, which has the best possible performance among the Optimized Copy-On-Update implementations.

4.5.3 Throughput Comparison

While overhead measurement is an effective way to compare algorithms against each other, it does not show the real effect on applications. In this experiment, we compare the throughput of our synthetic workload application for each algorithm. Our experiment includes the MBCOU-MBTaS-ASM algorithm, the best variant of Optimized Copy-On-Update implementations, the Optimized-Writes algorithm, as well as a System-Level Copy-On-Update implementation using the *fork* system call. Since our synthetic workload models the application state as arrays of words and updates are also at word-level, we derive a version of Wait-Free Ping-Pong which implements the best memory layout for this workload, called Integer-Only Wait-Free Ping-Pong.

Figure 4.2 shows the throughput comparison for the following implementations.

- **PP**. A generic implementation of Wait-Free Ping-Pong which can be used for all types of applications.
- **IP**. A specific implementation fo Wait-Free Ping-Pong which can be used only for the applications whose application states are arrays of words.
- **MB**. The best variant of Optimized Copy-On-Update.

- **ZZ**. The best variant of Wait-Free Zigzag.
- **FO**. An implementation of the System-Level Copy-On-Update algorithm using the *fork* system call.
- **OW**. An implementation of the Optimized-Writes algorithm.
- **NS**. A generic implementation of the Naive-Snapshot algorithm.

NS exhibits a constant throughput drop of 5% for the whole range of update rates. This is expected since the Naive-Snapshot algorithm is agnostic to update rates. Among all the implementations, FO has the worst throughput except for extremely low update rates. Starting from 4,000 updates per second, FO performance is worse than NS performance. FO throughput continues to drop as the update rate increases upto 12,000 updates per second. However, the FO throughput remains constant after that, achieving 90% throughput of the application running without checkpointing. The reason FO has a constant throughput drop when the update rate reaches a certain point is that all the pages are dirtied at that update rate, and therefore increasing the update rate does not cause any more page to be copied.

The Wait-Free algorithms still display the best throughput of any of the algorithms for all but the highest update rates. They dominate NS for all update rates below 100,000 updates per second. When the update rate increases to 256,000 updates per second, the Wait-Free algorithms perform worse than FO. The optimized algorithms, MB and OW, which utilize lesser memory usage than the Wait-Free algorithms, have a comparable throughput with the Wait-Free algorithms for a small and medium range of update rates. For the update rates greater than 10,000 updates per seconds, MB and OW throughput drop more sharply than the Wait-Free algorithms; in particular, MB and OW throughput becomes worse than NS throughput at 16,000 updates per second, and become worse than FO throughput at around 100,000 updates per second.

Overall, this experiment shows that the Wait-Free algorithms and the optimized algorithms, such as MB and OW, are the methods with the highest throughput for low and medium update rates. They achieve almost 98% of the throughput of the application without checkpointing. For higher update rates, the Naive-Snapshot is the method of choice not only for its simplicity in implementation but also for its achievement in throughput.

4.6 Related Work

Levi and Silberschatz [69] enumerated the deficiencies of conventional approaches and suggested that checkpoint recovery activities should be perform incrementally and in parallel with transaction execution. Since then, various aspects of conventional checkpoint recovery schemes have been exploited to improve the performance; some focused on parallelism [65], some focused on shadowing [70], some focused on organizing pages [105, 104], which are briefly summarized in the following section.

Fast-Start Oracle proposed the Fast-Start checkpointing algorithm [65] which incrementally write dirtied blocks in the cache to disk. Compared with conventional checkpointing techniques, Fast-Start checkpointing does not incur bulk write and resultant I/O spikes. Moreover, Fast-Start provides tunable parameters so that a DBA can control the rate of checkpointing. With an aggressive setting, experimental results show that fast-start incur negligible overhead, less than 1% of the baseline performance. With persistent locking, i.e. row locks are stored persistently in the same block as rows of a table or an index , fast-start implements the Deferred Rollback strategy which allows new transactions to start before the the rollback phase of the database recovery completes. In addition, the rollback activities can be parallelized and happen in the background along

with normal processing. As a result, Fast-Start significantly reduces the recovery time and hence makes the database available more quickly.

Most of the optimizations which Fast-Start proposed, for examples, incrementally write dirtied blocks, rollback transactions in parallel, could apply to all techniques using write-ahead-logging. Other optimizations, such as leveraging persistent write-locks, dynamically adjusting configuration parameters, are special features of Oracle, which could be explored by Oracle-like database checkpoint-recovery.

SIREN Analogous to shadow paging, Liedes and Wolski designed SIREN, a consistency-preserving and memory-efficient checkpointing method [70]. The key design is to use tuple shadowing as opposed to page shadowing and logical pages instead of physical pages. Using tuple shadowing, SIREN can achieve a non-blocking checkpointing behavior. Basically, SIREN freeze all the pages that are to be backed up in that checkpoint to produce a consistent checkpoint. During the checkpoint period, all operations which are going to modify tuples in those pages are pending. Tuples with a pending operation are not included in the checkpoint image and can thus be committed and removed directly. The pending operations are not directly coupled with transactions, and hence adding or removing tuples are not visible to the ongoing transactions, providing a non-block behavior. In addition, logical page helps to organize tuples efficiently which in turn facilitates partial and incremental checkpoints. We can split and join pages without actually copying the tuples. It is also possible to create overfull pages than a page capacity. Also, adding and removing tuples are simple and low cost operations, just involving some linked list manipulation. Overall, SIREN achieves up to 30% higher transaction throughput compared with a fuzzy checkpoint method with undo/redo log.

Using logical page structure is the unique feature of SIREN. It provides a flexible

way to achieve consistent checkpoints. The main drawback of this technique is that it has to allocate additional memory for each updated tuple. For frequently updated applications, SIREN would double the amount of memory usage.

SNAP, Skippy To support on-line "back-in-time" execution, a DBMS usually needs to version the data and provides an efficient mechanism to checkpoint a consistent image of the database overtime [73, 74]. To prevent blocking the transactions while taking a checkpoint, SNAP [105], a snapshot service for object storage system, used a copy-on-write scheme to snapshot all the modified pages in a versioned database. To keep track of pages in a snapshot, SNAP implemented an extra page table which maps each modified page to the its slot in the archived snapshot. As a consequence, SNAP trades extra memory for better performance, i.e. it does not prevent applications from accessing the storage system even when snapshots are frequent. Experimental evaluation shows that the overhead SNAP is small, however, it is very hard to support back-in-time execution, which resembles an "as-of" query in a transaction time database, against long-lived, disk page level, copy-on-write snapshots. The problem is that finding much less frequently updated pages must pass over large number of duplicate mappings that corresponds to frequently modified pages. To overcome that problem, Skippy [104] augmented SPAN with an efficient indexing method over snapshots. Basically, the Skippy structure allows to skip over unneeded repeated mappings of frequently modified pages which experimentally shows up to a 19-fold performance improvement in highly skewed workloads.

In short, SNAP and Skippy used a traditional copy-on-write shadowing technique for taking consistent snapshots. They improve the checkpointing speed by using a more advanced Modified Object Buffer [40] technique which supports versioning. However, this technique is specific to the underneath storage system, and it is hard to exploit this technique in the context of other DMBS.

4.7 Conclusion

In this chapter, we have presented a series of optimization techniques which can be applied to the existing algorithms. In particular, we can use low-level memory-barriers to implement Copy-On-Update, which avoids expensive use of locking. Our experiment shows that the memory-barrier implementation using asm-operators has a higher performance than the implementation using gcc-atomic-builtins. With these optimizations, we implemented an optimized version of Application-Level Copy-On-Update. This optimized Copy-On-Update implementation achieves better throughput than the System-Level Copy-On-Update for a wide range of update rates. It also performs better than the Naive-Snapshot algorithm for low and medium update rates.

We have also showed an optimized implementation for Wait-Free algorithms, which uses only two copies of the application state. This implementation, called OW, has showed a comparable performance with the Wait-Free algorithms introduced in the previous chapter. A drawback of this optimization is that it increases the recovery time since the recovery procedure needs to read the most recent two checkpoints in order to rebuild the consistent state of the application at the time of the crash.

CHAPTER 5

BRRL: A RECOVERY LIBRARY FOR MAIN-MEMORY APPLICATIONS IN THE CLOUD

Our greatest glory is not in never failing, but in rising up every time we fail.

– R. W. Emerson

In this chapter we present BRRL, a library for making distributed main-memory applications fault tolerant. BRRL is optimized for cloud applications with frequent *points of consistency*. BRRL differs from existing recovery libraries by providing a simple table abstraction and using schema information to optimize checkpointing.

5.1 Overview of BRRL

BRRL is a library for making application state durable. Unlike existing recovery libraries that present a generic memory abstraction [15, 16], BRRL uses semantic information about the application to optimize logging and checkpointing. BRRL is suitable for a wide variety of data-intensive applications, but it does have several requirements. BRRL applications must store their data in tables and access them only through BRRL API functions. They must also have relatively frequent points of consistency at which the application is consistent and can be safely checkpointed. These points of consistency can be either *local* or *global*. In a distributed application, a local point of consistency is a point at which the state at a single machine is consistent, whereas a global point of consistency marks a time when the state of the entire application is consistent. For example, a single-threaded partitioned DBMS such as H-Store [109] would have local points of consistency since different nodes execute different transactions, while a time-stepped simulation [19] would have global points of consistency at

the end of each time-step. As more applications move to the cloud for scalability, we expect points of consistency to become increasingly common since cloud applications are often data-parallel and divide their work into relatively independent partitions.

BRRL uses *logical logging* to persist state during the time between checkpoints in order to roll the application forward to a consistent state in the event of a failure. Though logical logging can increase recovery time compared to physical logging, it introduces less overhead during normal operation, and we can reduce the replay time by taking very frequent checkpoints. To facilitate logging, applications must produce log records for any non-deterministic behavior necessary for re-execution. For instance, in a DBMS, the logical log would include the type or code and parameters for each transaction as well as their serialization order. Additionally, the application should be able to parse and apply these log messages during recovery. We leave this intentionally vague in order to give applications some degree of flexibility, but in most cases log entries will correspond to transactions or other client requests. In deterministic applications, logging is not necessary, and BRRL can simply be used as an efficient way to take periodic checkpoints.

BRRL uses semantic information about the application to optimize logging and checkpointing. Figure 5.1 shows a typical distributed system deployment with BRRL.

- The schema defines application data structures in terms of tables. BRRL uses C++-like syntax with per-table and per-attribute annotations.
- Developers implement application logic using standard C++. Developers manage application data through BRRL APIs.
- The BRRL compiler compiles the schema and application code to binaries which can be deployed to run on application servers.

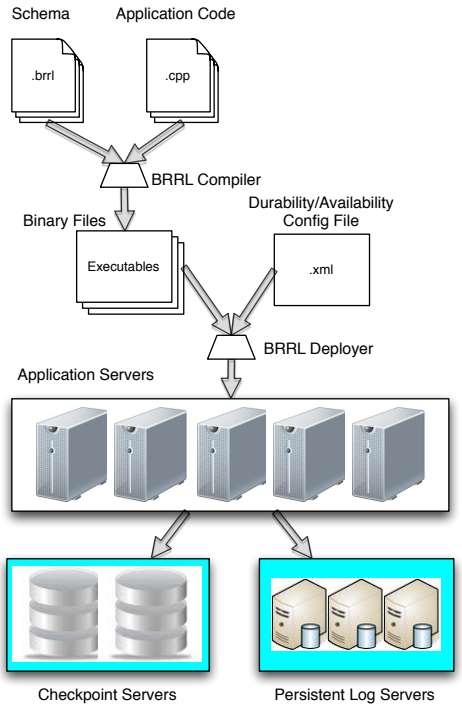


Figure 5.1: The System Overview

- Developers can specify availability and durability requirements in the configuration file. BRRL deployers uses the information in the configuration file to setup an optimal number of machines which satisfy the requirements.
- BRRL deploys the executables on a pool of application servers. This pool is shared by other applications with different availability/durability requirements.
- Logical logs are transfered to Persistent Log Servers and application states are checkpointed to Checkpoint Servers. This is done transparently to the users by BRRL.

The salient features of BRRL are:

- Application developers can define their data structure using simple table abstractions which support per-table and per-attribute annotations. Based on these anno-

tations, BRRL automatically chooses optimal checkpoint-recovery algorithms for their application. For example, BRRL decides to apply Naive-Snapshot algorithm to append-only tables and Wait-Free Ping-Pong to tables with no mutable string attributes.

- BRRL provides simple APIs which application developers can use to manage data. All getters and setters are named according to the corresponding attributes. For example, application developers can call *GetCustomerId()* and *SetCustomerId(id)* to get and set the *CustomerId* attribute.
- BRRL shields programmers from the complexity of low-level optimizations, such as cache-aware checkpoint-recovery implementations. The only requirement is that application developers need to call the function *PointOfConsistency* at points of consistency during application execution.
- BRRL separates application state into static and dynamic parts. Application state is made durable by storing the static state at the very beginning and maintaining a consistent copy of the dynamic state while the process is running. BRRL achieves high availability by fast recovery; a failure could be treated as a small "hiccup" while running an application.

5.1.1 The BRRL API

BRRL is implemented as a pre-compiler and a C++ library with a simple interface. Application developers specify their state in a set of BRRL tables using a syntax similar to structure definitions in C with scalar types. All of the state in these tables is checkpointed automatically so that it can be recovered in the event of a failure. Figure 5.2 shows how two tables in TPC-C would be specified in BRRL. Each attribute in these tables is classified as either `mutable` or `immutable` based on whether it is ever updated.

```
class Customer {
immutable:
    int c_id;
    double c_phone[16];
    ...
mutable:
    double c_balance;
    double c_ytd_payment;
    ...
};
```

```
class OrderLine {
immutable:
    int ol_o_id;
    int ol_w_id;
    int ol_quantity;
    double ol_amount;
    char ol_dist_info[24];
    ...
};
```

Figure 5.2: Example BRRL Tables

Method	Description
Status init(*RecoveryConfiguration rc*)	Initializes BRRL with a given configuration.
Status registerTable(*TableHandle th*)	Registers a given table (*th* is constructed at the time of table creation).
Status startCheckpointing()	Informs that the initialization and registration steps are done.
LSN writeLog(*LogRecord lr*)	Logs a new record as an uninterpreted string of bytes, returns the corresponding lsn.
LSN queryLog()	Gets the log sequence number of the most recently persisted log record.
Status persistLog(*LSN lsn*)	Persists the log up to a given *lsn*.
Status pointOfConsistency(*LSN lsn*)	Indicates that the application state is consistent at a given lsn.
Status recoverTables()	Populates a registered set of tables from the most recent consistent checkpoint.
LogIterator getLogIterator()	Gets an iterator to the logical log stored with the most recent consistent checkpoint.

Table 5.1: The BRRL API

Immutable attributes can never be updated unless the entire tuple is deleted. In Figure 5.2, all of the attributes in `OrderLine` are `immutable` since the table is append only, but the `c_balance` and `c_ytd_payment` attributes in the `Customer` table are `mutable` as they are updated whenever a customer makes a purchase or a payment. The BRRL compiler uses these annotations to compile BRRL tables into a set of C++ classes that are optimized for checkpoint-recovery. As BRRL may reorganize the storage layout, applications access their state using get and set methods generated by the compiler. Additionally, rows have no public constructors and must be created and deleted by calling methods on the appropriate table.

Table 5.1 shows the main set of the BRRL API which application developers can use to register tables, identify points of consistency, write log records, and recover state in the event of a failure. Since application developers have to access application data via

Method	Description
TableHandle createTable(*TableName tn*)	Creates the table which is defined in a brrl file with a given table name.
RowHandle addRow(*TableHandle th*)	Allocates a row for a given table.
RowHandle getRow(*TableHandle th, RowNum rn*)	Gets the row handle for a given row number.
YYY getXXX(*RowHandle rh*)	Gets a field (XXX is the field name and YYY is the field type).
Status setXXX(*RowHandle rh, YYY value*)	Sets a field (XXX is the field name and YYY is the field type).

Table 5.2: The Table API

the BRRL-generated methods, the BRRL API also provides the Table API which can be used to manage application data. Table 5.2 shows a subset of the Table API.

When the library is initialized (*init*), the application developer must specify high-level information about the application, including the checkpoint interval, the locations where checkpoints and logs will be written, and whether or not to use group commit. The application developers can then create their tables using the *createTable* method in the Table API. The passed-in table name is the same table name of the corresponding table defined in a brrl file. The application developers are responsible to register all of their tables to the BRRL library via the *registerTable* API. After that, the application developers can call *startCheckpointing* to inform the BRRL library that the initialization and registration steps have finished. During normal processing, the application developers can perform logical logging via the *writeLog* function. Note that the *writeLog* returns immediately even though the log record has not been persisted. The application developers can call *queryLog* to find out the most recently persisted log record or can persist their log records using the blocking function call *persistLog*. All log records are uniquely identified by their log sequence numbers. If a log record is determined to be persistent, then all the log records with lower log sequence numbers than this log record are guaranteed to be persisted.

The main requirement for application developers is that they should call *pointOfConsistency* at every point of consistency. Note that BRRL does not necessarily take a checkpoint each time this method is called, but every checkpoint is guaranteed to start

at a point of consistency. When a checkpoint is started, it also records the log sequence number from which the logical log can be replayed upon this checkpoint. During execution, the application developers access their application data using the Table API getters and setters. The recovery portion of the API consists of a method to recover all the registered tables (*recoverTables*) and an iterator to access the logical log (*getLogIterator*). It is the application's responsibility to use these methods to recover to a consistent state.

Taking TPC-C implementation as an example, we summarize BRRL usage as the following steps:

- Setup
    1. For each table in TPC-C, creates a brrl file specifying mutable and immutable attributes.
    2. Compiles these tables to .h and .cpp files using BRRL compiler.
- Initialize a BRRL instance
    1. Defines a recovery configuration, e.g., defines checkpoint and log servers.
    2. Initializes a BRRL instance with the recovery configuration using the BRRL API, i.e. call *BRRL::init(...)*
    3. Creates tables using the Table API, i.e. call *Table::createTable(...)*
    4. Registers all tables using the Table API, i.e. call *Table::registerTable(...)*
    5. Populates the registered tables, either
        (a) Call *recoverTables(...)*, get a logical log iterator and replay the logical log, i.e. *BRRL::getLogIterator(...)*, or
        (b) Put some initial data to tables manually by using setter functions.

6. Informs the BRRL library that the initialization and registration steps are done, i.e. call *BRRL::startCheckpointing(...)*

- Normal execution

  1. Builds a log record per transaction. For each log record, performs the following sequence of execution:

     (a) Write the log record, i.e. call *BRRL::writeLog(logRecord)*, get back an lsn.

     (b) Execute the application logic for this log record, i.e. call getters and setters through the Table API.

     (c) Inform the BRRL library that the execution is at a point of consistency, i.e. call *BRRL::pointOfConsistency(...)*, with the given lsn.

  2. Calls *BRRL::queryLog()* at an appropriate time to get the lsn which have been persisted, mark the corresponding transactions to be committed.

- Recover from a crash

  1. Performs the initialization and registration steps.

5.1.2 Checkpointing and Recovery in BRRL

BRRL includes components for efficient logging and checkpointing. Logging is implemented using a set of shared *persistent log servers*. When the application calls `writeLog(lr)`, the BRRL library sends the record to one or more log servers (the number is configurable). When one of these servers receives a log record, it stores the record in main memory and flushes it asynchronously to disk. This allows for very low-latency

logging since the log server returns as soon as the log record has been written to memory, and we expect the network latency to be much lower than the cost of a disk seek. This approach is similar to the *buffered logging* technique proposed by the RamCloud project [81]. For applications that cannot tolerate the loss of any of the log tail in the event of a catastrophic failure or power outage, the application servers can use group commit and wait until the log servers have flushed each record to disk.

Notice that since the persistent log servers do not have to perform any computation and can write to disk asynchronously, they can be shared by many applications. This sharing is particularly beneficial for cloud computing environments where many applications are multiplexed on the same infrastructure. This sharing can yield better resource utilization than approaches such as *k*-safety that rely on many additional replicas [109].

By default BRRL checkpoints data to a shared disk, though any stable-storage media would be suitable. We use the Wait-Free Ping-Pong checkpointing algorithm, which we introduced in Chapter 3, for frequently consistent main-memory applications. This algorithm maintains two copies of the application state in memory and collects updates in one of them for each checkpoint period (thereby "ping-ponging" between them). BRRL automatically interleaves attributes from the original state and the two copies in order to minimize the number of cachelines that must be accessed during an update. This was shown to improve performance dramatically. BRRL only applies these optimizations to mutable state in order to reduce the memory and update overhead. Immutable state is only written to the checkpoint once when it is first created. Deletes are persisted by writing a delete record with the checkpoint that can be replayed during recovery.

5.2 Implementation

This section describes the BRRL implementation in detail and shows the performance of the integrated algorithms.

5.2.1 Implementation Details

Based on the annotated information at the time of creating tables, BRRL implements the best algorithm recommended in Chapter 3. To guarantee reliable communications, all checkpoints and logical log are transferred through totally-ordered communication channels among the machines. Each channel is responsible by a separate thread for high concurrency.

BRRL has its own memory allocator to provide fast memory allocation. Internally, BRRL maintains a list of preallocated chunks of memory. At the time of creation, each table is assigned a vector of pointers to free chunks. Free chunks of memory are dynamically assigned as tables grow. Periodically, a background service inspect each table to regain the memory chunks are marked deleted by tables. The deleted chunks are then given back and will be assigned to other tables when needed.

BRRL is designed to work well with cloud infrastructure. The persistent servers are shared among various applications. To provide high availability, each machine in a partition is replicated. A proxy is responsible to set up separate communication channels to replicas. In order to provide latency, the logical logs are checkpointed to the main memory of a persistent server; meanwhile, the application states are checkpointed asynchronously.

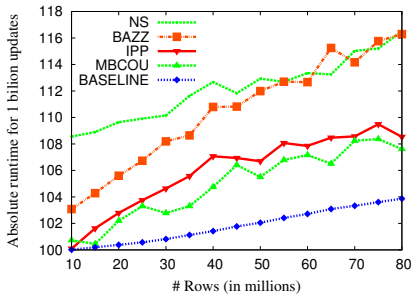


Figure 5.3: Zipf workload: Overhead

BRRL explores the impact of data layout in memory on cache perform and does a careful cache-aware implementation with various low level optimizations. In particular, data structure with low and moderate update rates, BRRL either applies Wait-Free Ping-Pong or Wait-Free Zigzag algorithm, depending on the size of the application state; for extremely high update rates, BRRL uses Naive-Snapshot.

5.2.2 Integrating Checkpoint-Recovery Algorithms

In order to verify that the integrated checkpoint-recovery algorithms, we run an experiment on a synthetic workload. In this experiment, we model an application state as a 2-D array of words. We fix the number of columns and vary the number of rows to experiment applications with different state size. The number of columns is fixed as four. The number of rows is varied from 10 millions to 80 millions. So, effectively, we vary the application state size from 1.2 GB to 9.6 GB. Updates are applied randomly to the application state. We measure the time to apply one billion updates.

Figure 5.3 displays the performance for some selective algorithms, namely NS (Naive-Snapshot), BAZZ (Wait-Free Zigzag), IPP (Wait-Free Ping-Pong) and MBCOU(Application-Level Copy-On-Update). BASELINE is the run when checkpoint

algorithms are disabled. As we expected, IPP, BAZZ and MBCOU exhibit better performance than NS for most of the time. IPP and MBCOU have very comparable performance and are better than BAZZ at every state size.

5.3 Related Work

There have been different approaches to integrating checkpoint-recovery systems with applications. A first consideration is whether checkpointing should be implemented at the system [71, 87] or application level [12, 15, 16]. In addition, checkpointing may be offered to applications embedded in a language runtime [127], through a library [87], or with compiler support [15, 16]. Our work performs application-level checkpointing and integrates with the application logic through a library API. 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 natural points of consistency, to determine which images of the state to persist.

There is a large distributed systems literature that explores how to generalize efficient single-node checkpointing schemes to multiple nodes. A recent survey by Elnozahy et al. describes these distributed checkpointing approches [34]. *Uncoordinated checkpointing* applies single-node algorithms independently, but may have to deal with cascading rollbacks in order to reconstruct a valid global snapshot. *Coordinated checkpointing* avoids that problem at the cost of extra synchronization among nodes during checkpointing. *Message-logging* approaches combine uncoordinated checkpointing with logging of message exchanges to be able to replay the computation from the last checkpoint taken from any single node. We can again take advantage of natural points of consistency when selecting the best technique to use in distributed applications. For example,

applications that have global natural points of consistency, such as time-stepped simulations, can piggyback coordinated checkpoint requests on their natural synchronization barriers. If the application is only loosely synchronized, but still exhibits local points of consistency, a message-logging approach promises better performance. These methods can be transparently layered on top of the efficient single-node algorithms developed as part of our initial work. However, they contain assumptions that restrict their use on heterogeneous cloud environments, a subject to be explored in our future work.

In classic relational DBMSs, ARIES is the gold standard for recovery [77]. Unfortunately, ARIES relies on physiological logging, implying unacceptable logging overheads for the stream of local updates produced by high-throughput, in-memory data-driven applications. The same observation applies to fuzzy checkpointing, which also requires physical logging of the update stream [50, 98].

Recently, the Berkeley/Stanford Recovery-Oriented Computing (ROC) project has explored techniques to reduce time to recover of general applications, under the premise that failures due to hardware malfunction, software bugs or human error are inevitable [82, 18, 17]. This premise also holds for large computing infrastructures [101], and we expect it to be the norm on cloud environments. Checkpoint recovery is an important building block for systems operating in failure-intensive environments. For example, in ROC it has been used to implement a generic undo system applied to an email store [17]. Our approach is similar, but allied with the notion of points of consistency and targeted to the specific challenges of a cloud environment.

Replication. Replication has been extensively used to provide fault-tolerance and high availability in distributed systems. In particular, recent distributed file systems such as Amazon S3, Hadoop and the Google File System (GFS) automatically replicate each fragment of a file, with replication degree $K = 3$ being typical for large clusters [41, 49,

7]. A background repair mechanism creates new segment copies to maintain K replicas in response to disk failures.

Recently, data-driven applications such as data mining have begun to be expressed as iterated MapReduce jobs [25]. We argue that the approach for fault tolerance in MapReduce implementations will need to be rethought for these applications. In particular, most implementations save input and output of each job to a reliable replicated file system [32, 41]. This use of replication is not sufficient to guarantee fault-tolerance, as the minimum running time of even a moderate size Map-Reduce job can be much greater than its MTTI. Map-Reduce implementations rely on the repeatability of Map and Reduce tasks to reschedule tasks assigned to a failed node. In Google's published implementation, which takes no intermediate checkpoints, failure of a Map node may require (possibly parallel) re-execution of all Map tasks previously assigned to that node [32]. In contrast, data-driven applications following our approach will have their state checkpointed at natural points of consistency. The checkpointing interval can be set by balancing the overhead of taking checkpoints with the mean time to failure of the nodes involved in the computation [101, 30, 72, 126].

In the context of classic RDBMSs, transactional, update-anywhere, synchronous replication has been shown to be fundamentally not scalable [43]. Most recent work has focused on asynchronous replication or on small server clusters with hot-standby architectures, with a few notable exceptions [61, 125, 58]. Asynchronous replication is the technique of choice for database caching [4, 8, 31, 48, 66, 106], given the acceptability of occasional data loss in this application. As asynchronous replication offers better performance than eager replication, it is also offered to clients by many commercial DBMS vendors [80, 54, 108]. Recently, Ganymed uses asynchronous replication internally for DBMS scale-out, but hides it from clients by offering a transactional mid-

dleware interface [89]. Of course, mechanisms for durability must still be provided by the primary-copy DBMS.

ClustRa uses the idea of hot standbys and log shipping to provide high availability [53]. To achieve higher performance, log records are written not 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 [123]. Backup sites save frequent checkpoints of their main memory to stable storage. More recently, Lau and Madden [67] and Stonebraker et al. [109] advocate a distributed design in which no logging is performed by the system. In contrast to Whitney et al. [123], all sites are active and thus updates are processed in all sites that replicate a given portion of the data. Data is replicated at a minimum of K sites so that the system will survive up to K site failures. This solution is similar to process-pairs in high-performance scientific simulations [101]. While availability is high, system utilization is rather low ($1/K$), given that all active copies of the data-driven application perform redundant work. In contrast, our checkpoint recovery model increases utilization at a potential cost in recovery time.

Erasure-resilient Codes. Erasure-resilient codes protect data against failures at lower storage and bandwidth cost than replication [35, 120]. The downside is higher latency and overhead for computing the coding scheme. A well-known use of erasure-resilient codes in data storage is for building Redundant Arrays of Inexpensive Disks (RAID) [83]. When multiple disks operate as a single storage device, the mean time to failure is drastically reduced unless redundancy techniques are employed. RAID systems typically rely on parity to tolerate single-disk failures and more complex schemes, such as Reed-Solomon codes, to tolerate multiple disk failures [86].

Plank and Li present a diskless checkpointing scheme based on RAID-like techniques [88]. Their basic scheme survives single-node failures, but can be generalized

to survive m failures given $2m$ extra nodes. These techniques have also been extended to disk-based checkpointing [85]. Due to assumptions such as frequent communication with a parity node, these schemes are restricted to small clusters with homogeneous, high-performance network interconnects. In our proposal, in contrast, we plan to investigate techniques that work on cloud infrastructures composed of heterogeneous commodity hardware components. addition, single-node techniques that ignore effects from processor caches and pipeline stalls must be revisited.

Other systems, such as OceanStore [64] and Intermemory [22], use erasure-resilient codes in the design of large-scale archival services. These systems are designed to be deployed over widely distributed, untrusted servers. As such, they must deal with Byzantine failures. Unlike these systems, we expect data-driven applications to be run on clusters of servers co-located in the same data center. While machines may exhibit widely different performance due to heterogeneity, the failure modes experienced in this architecture are closer to fail-stop failures than to fully general Byzantine failures [102].

5.4 Conclusion

In this chapter, we have presented BRRL, a checkpoint-recovery library, for distributed systems in the cloud. BRRL provides data durability and data locality transparently to application developers. Every operation on data in our library is done using light-weight operations. That enables various optimizations which have been studied extensively in this dissertation. Also, BRRL allows users to trade availability to achieve durability with much lower cost.

CHAPTER 6

SUMMARY AND FUTURE WORK

While promising huge improvements in performance and cost, main-memory data-driven applications in the cloud computing needs to address a fundamental problem before becoming ubiquitous: They need to become fault-tolerant.

In this dissertation, we first present a benchmarking framework which we use to evaluate the applicability of existing checkpoint-recovery techniques developed for main-memory DBMS to MMO workloads. This framework is extensible for various other main-memory applications, such as OTLP and behavioral simulations, which have frequent points of consistency. Through this framework, we show how to adapt consistent main-memory checkpointing techniques to frequently consistent (FC) applications. We also provide a comprehensive simulation model and evaluate six different strategies. Through a thorough evaluation, we find that there is no single checkpoint-recovery technique which outperforms all the others. In particular, we recommend application developers to use the Copy-On-Update algorithm for low and median update rates and the Naive-Snapshot algorithm for extremely high update rates. In addition, we also analyze undesirable properties of the existing algorithms; i.e., they either use locks or large synchronous copy operations, which hurt throughput and latency, respectively.

The findings of undesirable properties of existing algorithms motivate us to design and implement new algorithms which completely avoid locking and synchronous copying. We have also identified several important requirements for high-performance checkpointing techniques. First, the algorithm should have low overhead during normal operation; ideally, taking a checkpoint should have little impact on the throughput of the system. Second, the algorithm should distribute its overhead uniformly and not introduce any latency spike. Finally, it should be possible to take checkpoints very fre-

quently so as to reduce the recovery time. Guided by these requirements, we design and implement two novel checkpoint-recovery algorithms, namely Wait-Free Zigzag and Wait-Free Ping-Pong, which trade additional space in main memory for significantly lower overhead and latency. 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 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.

While implementing the Wait-Free algorithms, we notice that 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. We go a step further to leverage modern hardware architecture to explore other optimizations. In fact, we construct a series of optimizations which are not only amenable to our Wait-Free algorithms but can also be applied to existing algorithms. In particular, we use low-level memory-barrier operators to design and implement an optimized version of Application-Level Copy-On-Update. This optimized Copy-On-Update algorithm achieves a comparable performance with the performance of our Wait-Free algorithms; however, it uses less memory.

To hide the complexity of maneuvering memory layouts and low-level optimizations, we provide BRRL, a recovery library which transparently implements the proposed techniques. BRRL is optimized for cloud applications with frequent points of consistency. BRRL differs from existing checkpoint-recovery libraries by providing a simple table abstraction and using schema information to optimize checkpointing.

6.1 Future Work: Comprehensive Fault-Tolerance for Data-Driven Applications

In this section, we discuss future work along two dimensions. First, how to design methods that will work on cloud infrastructures. Second, how to relax the assumptions that we made in the previous chapters about the application architecture to achieve truly comprehensive checkpoint recovery.

6.1.1 Cloud Architecture

Cost versus Performance. One important characteristic of the cloud is that it is a *variable-cost* infrastructure. Typically, scientific computing clusters operate in fixed-cost regime; large hardware investments are made up-front to set up the cluster, and applications are optimized to reap the maximum benefit from the available hardware resources. In the cloud, there is a new tradeoff: In a data-driven batch application we may want to optimize not for minimum expected running time, but rather for minimum expected running *cost*. Adding an extra node to a computation may reduce the total expected running time, but the cost of running the extra node for the whole computation may not be offset by the reduction in cost due to the shorter compute time.

As observed in the literature on checkpoint recovery [30, 72, 126], the checkpoint interval can be tuned to balance the computation time of the application, the checkpoint overhead, the MTTI, and the recovery time. Therefore, it is useful to consider cost as well, and explore new algorithms for checkpoint recovery optimized for this trade-off. These algorithms will comprise new *pricing-aware checkpoint-recovery* methods and will complement our research on *latency-aware checkpoint-recovery* methods pre-

sented in this dissertation. These two families of algorithms will serve the needs of cost-sensitive and jitter-sensitive applications, complementing traditional checkpoint-recovery work optimized for total expected running time over a fixed-cost infrastructure.

Special-Purpose versus Commodity Hardware. The cloud violates several simplifying assumptions made by previous distributed checkpoint-recovery methods [34]. The most restrictive assumption is that checkpoints will be written to a shared disk accessible to other nodes. This assumption was reasonable on scientific computing clusters with expensive data storage solutions and network interconnects. Cloud infrastructures are, however, built out of low-cost commodity hardware.

While standardization initiatives such as Fiber Channel over Ethernet (FCoE) aim to bring sophisticated storage protocols to less expensive Ethernet interconnects [36], wide adoption on commodity networks used by cloud infrastructures is far from a certainty. Instead, diskless checkpointing schemes, such as mirror checkpointing [24], are a natural alternative to expensive shared disks. In mirror checkpointing, nodes are organized in a ring and save checkpoints to themselves and to their neighbors. Note that this assumes coordinated checkpoints for state saved locally, but the technique can also be extended to message-logging protocols. Unfortunately, this work also assumes that nodes have uniform bandwidth and memory capacities, which is not true in cloud environments. In fact, widely varying network bandwidths may be observed between nodes located in different racks or clusters [10]. In addition, cloud providers operate with heterogeneous servers. Main memory and processing power of nodes in the system may differ depending on cost or age. Concurrent client tasks may affect performance by competing for processing power or bandwidth.

The first step to address the problem of heterogeneity is to benchmark the performance of previous checkpoint recovery methods, such as mirror checkpointing, in cloud

environments. This will determine the exact means by which this lack of uniformity degrades their performance. Because of heterogenous bandwidth, nodes using mirror checkpointing need to be arranged so that neighbors have sufficient bandwidth to handle the message traffic passing between them. This may require that the nodes be arranged in a network overlay other than a ring. Furthermore, this overlay may need to change as the number of messages between neighbors changes over time. Hence, a second step in addressing non-uniformity is the development of load-balancing algorithms that rearrange neighbors according to bandwidth demand.

Fixed versus Elastic Distribution. Another unique difficulty in running data-driven applications on the cloud is *elasticity*. Unlike cluster computing, in which a program is run on a specific number of nodes, cloud computing is a service where computing nodes can be added and removed according to computing needs. Adding more nodes to a running distributed computation alters the MTTI for the computation as well as the computation time. Depending on the balance between reduction in MTTI and decrease in computation time, the total expected runtime or cost of a data-driven batch application could actually degrade when new nodes are added. This problem can be addressed in two ways. First, by developing analytical methods for optimizing the number and the configuration of nodes for any given checkpoint recovery technique. Second, by developing algorithms that allow a system to optimally switch between checkpoint recovery methods, adapting to changes in the computing configuration.

6.1.2 Application Architecture

Static versus Dynamic Memory Allocation. This dissertation assumes data structures that are relatively static. We place objects in fixed tables, and use foreign keys instead

of pointers. This makes serialization of our data easy, and simplifies cache-aware placement optimizations.

This assumption is too restrictive for general data-driven applications. While a number of existing applications do in fact represent data as first normal form relations in main memory [5, 122], the runtimes of these systems often build additional data structures for performance reasons. Effective checkpoint recovery techniques will need to balance the savings achievable by not checkpointing these structures against the additional recovery time required to rebuild them after a crash. In addition, data is often dynamic: over its lifetime the application may undergo huge changes in the amount of data being managed. Thus, it is beneficial to make cache-aware placement optimizations for *dynamically allocated* data. There has not been a great deal of work on the locality properties of memory allocators, and most existing work has addressed aggregate cache miss rates, without attempting the kind of detailed placement decisions that we need for our algorithms [124, 45, 100, 37]. The general problem of designing an allocator placement policy to minimize cache misses is provably intractable [84]. An alternative approach is to widen the allocator API to support declarative specification of which objects require shadow copies and metadata for checkpointing. The allocator will then be able to make appropriate placement decisions to minimize checkpointing overhead for those objects.

A related issue is fragmentation: Checkpointing a fragmented heap is more expensive than checkpointing a compact one. At a higher level, data representation also potentially affects the choice of checkpoint scheme. This dissertation assumes a row-store format, but recent work suggests that column stores are sometimes much more efficient [13, 110]. One promising approach involves schemes for periodic reorganization, similar in spirit to differential indexing [103, 56, 79, 78].

Single-threaded versus Parallel Mutator. One strong assumption made in our work is that only two threads interact during execution: a single-threaded Mutator and an asynchronous checkpoint writer. We have taken advantage of this assumption to make the Mutator and checkpoint writer lock-free between points of consistency, resulting in significant improvement in the overhead introduced by checkpointing.

In the future, it is expected that data-driven applications to be heavily optimized for multi-core architectures. Naturally, these systems will employ a mix of data and operator-level parallelism, as in parallel database systems. In particular, there is interaction between the technique used to parallelize the mutator and the amount of overhead introduced by the recovery strategy. If the mutator itself uses locks for synchronization, then the relative benefit of a lock-free recovery thread will be smaller and may not justify the space required for shadow copies.

In addition, it is well-known that cache optimizations are a major contributor to performance in modern CPU architectures, and the database literature has started to investigate the interplay of parallelism and cache awareness [68, 113, 63, 1]. There is a need to investigate how cache optimizations applied to the mutator influence the optimizations that can be used to diminish checkpointing overhead. The appropriate optimization goal here is again to minimize total expected runtime or monetary cost [101]. It may well be worth incurring extra checkpointing overhead if this cost helps to increase the performance in the parallel mutator.

Single-Node versus Distributed Applications. Significant work has been done in the distributed systems area on multi-node checkpointing in message-passing systems [34]. As we have discussed in this dissertation, either coordinated or message-logging checkpoint protocols may be reused for data-driven applications, depending on whether they exhibit global or only local points of consistency. While distributed applications are

clearly important, we currently believe that existing techniques will suffice for the cloud, and that we can layer them on top of our proposed work.

BIBLIOGRAPHY

[1]

[2] Michel Adiba and Bruce Lindsay. Database Snapshots. In *Proc. VLDB*, 1980.

[3] Anastassia Ailamaki, David DeWitt, Mark Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, 2001.

[4] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache tables: paving the way for an adaptive database cache. In *Proc. VLDB*, pages 718–729, 2003.

[5] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell C Sears. BOOM: Data-centric programming in the datacenter. Technical Report UCB/EECS-2009-113, EECS Department, University of California, Berkeley, 2009.

[6] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2.

[7] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3.

[8] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *ICDE*, pages 821–831, 2003.

[9] Mary Baker and Mark Sullivan. The recovery box: Using fast recovery to provide high availability in the unix environment. In *Proc. USENIX Summer Conference*, pages 31–43, 1992.

[10] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[11] Joel Bartlett, Jim Gray, and Bob Horst. Fault tolerance in tandem computer systems. Technical Report 86.2, PN87616, Tandem Computers, 1986.

[12] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogenous networks of workstations. *J. Parallel Distrib. Comput.*, 43(2):147–155, 1997.

[13] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[14] Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Recent Advances in Checkpoint/Recovery Systems. In *Proc. IPDPS*, 2006.

[15] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated Application-level Checkpointing of MPI Programs. In *Proc. PPoPP*, 2003.

[16] Greg Bronevetsky, Martin Schulz, Peter Szwed, Daniel Marques, and Keshav Pingali. Application-level Checkpointing for Shared Memory Programs. In *Proc. ASPLOS*, 2004.

[17] Aaron B. Brown and David A. Patterson. Undo for operators: building an undoable e-mail store. In *Proc. USENIX Annual Technical Conference*, pages 1–14, 2003.

[18] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proc. OSDI*, pages 31–44, 2004.

[19] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. SIGMOD*, 2011.

[20] Simon Carless. The Activision/Blizzard Merger: Five Key Points.

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

[22] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A Prototype Implementation of Archival Intermemory. In *Proc. ICDL*, 1999.

[23] Zizhong Chen and Jack Dongarra. Highly scalable self-healing algorithms for high performance scientific computing. *IEEE Transactions on Computers*, 58(11):1512–1524, 2009.

[24] Tzi-Cker Chiueh and Peitao Deng. Evaluation of Checkpoint Mechanisms for Massively Parallel Machines. In *Proc. FTCS*, 1996.

[25] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, Yuan Yuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *Proc. NIPS*, 2006.

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

[27] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 1(1):1153–1165, 2008.

[28] Ruben Cortez. World Class Networking Infrastructure. In *Proc. Austin GDC*, 2007.

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

[30] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[31] Special issue on database caching strategies for web content, 2004. IEEE Data Engineering Bulleting, Volume 27, Number 2.

[32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, 2004.

[33] David J DeWitt, Randy Katz, Frank Olken, Leonard Shapiro, Michael Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. SIGMOD*, 1984.

[34] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[35] Elwyn R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, 1968.

[36] Fibre channel over ethernet (fcoe). http://www.t11.org/fcoe.

[37] Yi Feng and Emery D. Berger. A locality-improving dynamic memory allocator. In Brad Calder and Benjamin G. Zorn, editors, *Memory System Performance*, pages 68–77. ACM, 2005.

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

[39] Built-in functions for atomic memory access. http://gcc.gnu.org/onlinedocs/gcc-4.2.0/gcc/Atomic-Builtins.html.

[40] Sanjay Ghemawat and Sanjay Ghemawat. The modified object buffer: A storage management technique for object-oriented databases. Technical report, 1995.

[41] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System . In *Proc. SOSP*, 2003.

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

[43] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proc. SIGMOD*, pages 173–182, 1996.

[44] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *Proc. SIGMOD*, 1994.

[45] Dirk Grunwald, Benjamin G. Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *PLDI*, pages 177–186, 1993.

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

[47] Erico Guizzo. The Game-Frame Guild. *IEEE Spectrum*, August, 2008.

[48] Hongfei Guo, Per-ke Larson, and Raghu Ramakrishnan. Caching with 'good enough' currency, consistency, and completeness. In *Proc. VLDB*, pages 457–468, 2005.

[49] Hadoop Distributed File System Architecture. http://hadoop.apache.org/common/docs/current/hdfs_design.html.

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

[51] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proc. SIGMOD*, pages 981–992, 2008.

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

[53] Svein-Olaf Hvasshovd, Oystein Torbjornsen, Svein Bratsberg, and Per Holager. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In *Proc. VLDB*, 1995.

[54] *SQL Replication Guide and Reference*, 2009. IBM DB2 Version 9.7 for Linux, UNIX, and Windows English manuals.

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

[56] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *Proc. VLDB*, pages 16–25, 1997.

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

[58] R. Jiménez-Peris, M. Pati no Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proc. ICDCS*, pages 477–484, 2002.

[59] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *Proc. SIGMOD*, 2010.

[60] Bo Käler and Oddvar Risnes. Extending Logging for Database Snapshot Refresh. In *Proc. VLDB*, 1987.

[61] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 134–143. Morgan Kaufmann, 2000.

[62] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[63] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs.

hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[64] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. ASPLOS*, 2000.

[65] Tirthankar Lahiri, Amit Ganesh, Ron Weiss, and Ashok Joshi. Fast-start: quick fault recovery in oracle. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 593–598, New York, NY, USA, 2001. ACM.

[66] Per-ke Larson, Jonathan Goldstein, and Jingren Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *ICDE*, pages 177–188, 2004.

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

[68] Rubao Lee, Xiaoning Ding, Feng Chen, Qingda Lu, and Xiaodong Zhang. Mcc-db: Minimizing cache conflicts in multi-core processors for databases. *PVLDB*, 2(1):373–384, 2009.

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

[70] Antti-Pekka Liedes and Antoni Wolski. SIREN: a Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for In-Memory Databases. In *Proc. ICDE*, 2006.

[71] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System. Technical Report 1346, University of Winsconsin-Madison, 1997.

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

[73] David Lomet, Roger Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Immortal db: transaction time support for sql server. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 939–941, New York, NY, USA, 2005. ACM.

[74] David Lomet, Zografoula Vagena, and Roger Barga. Recovery from "bad" user transactions. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 337–346, New York, NY, USA, 2006. ACM.

[75] Trina 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 .

[76] Memory Barrier. http://www.mjmwired.net/kernel/Documentation/memory-barriers.txt.

[77] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, 1992.

[78] Peter Muth, Patrick O'Neil, Achim Pick, and Gerhard Weikum. The lham log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, 2000.

[79] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[80] *Oracle Streams Replication Administrators Guide, 11g Release 2 (11.2)*, 2009. Oracle Database Documentation Library.

[81] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, 2010.

[82] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, Computer Science Technical Report, University of California, Berkeley, 2002.

[83] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. SIGMOD*, 1988.

[84] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *POPL*, pages 101–112, 2002.

[85] James S. Plank. Improving the Performance of Coordinated Checkpointers on Networks of Workstations using RAID Techniques. In *Proc. SRDS*, 1996.

[86] James S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[87] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under UNIX. In *Proc. USENIX Winter Technical Conference*, 1995.

[88] James S. Plank and Kai Li. Faster Checkpointing with N+1 Parity. In *Proc. FTCS*, 1994.

[89] Christian Plattner. *Ganymed: A Platform for Database Replication*. PhD thesis, ETH Zurich, 2006. Diss. ETH No. 16945.

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

[91] Shannon Posniewski. SQL Considered Harmful. In *Proc. GDC*, 2008.

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

[93] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogeneous architectures. In *In Symposium on Fault-Tolerant Computing*, pages 58–67. Kluwer Academic Press, 1997.

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

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

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

[97] Daniel Rosenkrantz. Dynamic Database Dumping. In *Proc. SIGMOD*, 1978.

[98] Kenneth Salem and Hector Garcia-Molina. Checkpointing Memory-Resident Databases. In *Proc. ICDE*, 1989.

[99] Marcos Antonio Vaz Salles, Tuan Cao, Benjamin Sowell, Alan J. Demers, Johannes Gehrke, Christoph Koch, and Walker M. White. An evaluation of checkpoint recovery for massively multiplayer online games. *PVLDB*, 2(1):1258–1269, 2009.

[100] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In Erez Petrank and J. Eliot B. Moss, editors, *ISMM*, pages 84–94. ACM, 2006.

[101] Bianca Schroeder and Garth Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conf. Ser.*, 78, 2007.

[102] Bianca Schroeder and Garth A. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proc. DSN*, 2006.

[103] Dennis G. Severance and Guy M. Lohman. Differential files: their application to the maintenance of large databases. *ACM TODS*, 1(3):256–267, 1976.

[104] Ross Shaull, Liuba Shrira, and Hao Xu. Skippy: a New Snapshot Indexing Method for Time Travel in the Storage Manager. In *Proc. SIGMOD*, 2008.

[105] Liuba Shrira and Hao Xu. SNAP: Efficient Snapshots for Back-in-Time Execution. In *Proc. ICDE*, 2005.

[106] Swaminathan Sivasubramanian, Gustavo Alonso, Guillaume Pierre, and Maarten van Steen. Globedb: autonomic data replication for web applications. In *WWW*, pages 33–42, 2005.

[107] Ben Sowell, Alan J. Demers, Johannes Gehrke, Nitin Gupta, Haoyuan Li, and Walker M. White. From declarative languages to declarative processing in computer games. In *CIDR*, 2009.

[108] *SQL Server Replication*, 2009. SQL Server 2008 Product Documentation.

[109] Michael Stonebraker, Samuel Madden, Daniel Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proc. VLDB*, 2007.

[110] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column oriented dbms. In *Proc. VLDB*, pages 553–564, 2005.

[111] Trevor Strohman and W. Bruce Croft. Efficient Document Retrieval in Main Memory. In *Proc. SIGIR*, 2007.

[112] Tim Sweeney. The Next Mainstream Programming Language: a Game Developer's Perspective. In *Proc. POPL*, 2006.

[113] Shirish Tatikonda and Srinivasan Parthasarathy. Mining tree-structured data on multicore systems. *PVLDB*, 2(1):694–705, 2009.

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

[115] Alexander Thomson and Daniel Abadi. The case for determinism in database systems. In *Proc. VLDB*, 2010.

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

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

[118] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

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

[120] Hakim Weatherspoon and John D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. IPDPS*, 2002.

[121] Walker White, Christoph Koch, Nitin Gupta, Johannes Gehrke, and Alan Demers. Database Research Opportunities in Computer Games. *ACM SIGMOD Record*, 36(3), 2007.

[122] Walker M. White, Alan J. Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportion. In *SIGMOD Conference*, pages 31–42, 2007.

[123] Arthur Whitney, Dennis Shasha, and Stevan Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL, or C++. In *Proc. HPTS*, 1997.

[124] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry G. Baker, editor, *IWMM*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer, 1995.

[125] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433, 2005.

[126] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.

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