Checkpointing at System Calls using BDI Compression

Adam K. Hastings†, Hiroshi Sasaki†, Miguel A. Arroyo, Kent Williams-King, Vasileios P. Kemerlis, and Simha Sethumadhavan

Abstract—For many years, checkpoint and recovery schemes have been proposed as a way for systems to take snapshots of their state, and then later revert to them as needed. More recently, a renewed interest in the topic has led to highly-optimized, hardware-assisted checkpoint and recovery systems. While these systems have tolerable overhead under normal operation, they perform poorly during recovery. In this work, we present a new hardware-assisted checkpoint and recovery system which has minimal overheads in both taking and restoring from checkpoints. Our approach leverages data compression to store checkpoint data inline with program data which reduces the amount of data that needs to be logged between checkpoints. Compression also enables us to aggressively checkpoint in caches as opposed to prior approaches which do so only in memory, freeing us from invalidating the content of caches upon checkpoint and recovery. Our compression-based checkpointing system requires ∼50% less checkpoint data to be stored in memory when evaluated with SPEC CPU2006 benchmarks.

Index Terms—Checkpointing, error recovery, fault tolerance

1 Introduction

In computer systems, checkpoint and recovery (or checkpointing) is the act of periodically recording a system's state, and then later recovering from the recorded state if necessary. Such systems have been used extensively in fault tolerant computing, where recorded states (or checkpoints) act as a system backup in case of an unrecoverable error. Checkpointing with deterministic replay has also been proposed for a number of different applications, ranging from improved debuggers [11] to cache side-channel detections [12] to malware defenses [8]. Despite the success of current checkpoint and recovery systems, future applications are stymied by the non-trivial overheads of taking and recovering from checkpoints.

In this work, we introduce a new checkpoint and recovery system which improves performance over the state of the art. Our proposed system is based on the principle that data values exhibit spatial and temporal locality, which can be exploited to reduce the volume of information needed when taking checkpoints via data compression. In particular, we find that subsequent data updates tend to be similar enough to frequently enable base-delta-immediate (BDI) compression [6], which we use to store two values in the same address. Using this approach, we can store both a checkpoint and subsequent updates to the checkpoint inline with each other. By keeping both the checkpoint and current system state in the same location, we avoid additional memory accesses for both taking and recovering from checkpoints. Using this approach, we accrue performance overheads (due to additional memory accesses) only when a checkpoint value and subsequent update cannot be compressed inline.

Using the SPEC CPU2006 benchmark suite [5], we evaluate our design and demonstrate that our system can expect to recover from a checkpoint at least 2x faster than the state of the art for single cores, without compromising the error-free (non-recovery) performance. Our approach adds only minimal hardware: BDI (de)compression hardware, and three bits of metadata per cache line size data chunks.

2 Motivation

As shown in TABLE 1, recovery time is key to the utility of various types of checkpointing. Some techniques, such as speculative execution and hardware transactional memory, require very low overheads for taking state, since recording tend to happen at very fine granularities. However, as the required granularity becomes larger, checkpoints are taken less frequently, and recovery latency becomes less important. This is seen in existing coarse-grained checkpointing systems, where checkpoints are taken millions of cycles apart (if not more), permitting recovery times to be large. In previous systems, this long recovery time has been tolerable, since most checkpoint and recovery systems have been motivated primarily by the need for fault tolerant and resilient computing, where recoveries are assumed to be triggered by rare hardware faults. Consequently, existing systems do not scale well at finer granularities or as the rate of recoveries increases, and are therefore limited in the types of applications they can support. In contrast, our system aims to enable checkpointing applications that require finer granularities or faster recoveries than existing course-grained systems allow.

Faster recovery times are an important factor in determining which types of checkpointing applications are

†*Joint first authors.*

- *A. K. Hastings, H. Sasaki, M. A. Arroyo, and S. Sethumadhavan are with the Department of Computer Science, Columbia University in the City of New York, New York, NY 10027. E-mail: {hastings, sasaki, miguel, simha}@cs.columbia.edu*
- *K. Williams-King and V. P. Kemerlis are with the Department of Computer Science, Brown University, Providence, RI 02912. E-mail: {kwk, vpk}@cs.brown.edu*

TABLE 1
Comparison of different checkpoint and replay schemes.

Purpose	*Mechanism*	*Granularity*	*Checkpoint Latency*	*Recovery Latency*
Speculative Execution	On-chip buffers	∼50 stores	Negligible	∼10s of cycles
Hardware Transactional Memory	On-chip buffers (few Kbs)	∼100s of stores	Negligible	∝ Aborted transaction length (typically < 1M cycles)
Thread-Level Speculation [10]	Caches	∼1,000s of stores	∝ # of successful speculations × Bus request latency	∝ # of failed speculations × Bus request latency
Kilo-Window Processors [4]	Multi-level instruction queues	∼1,000s of insts	∝ # of physical registers	∝ Branch misprediction rate
Compression-Based Checkpointing (**This Work**)	Main memory	System calls	(Delayed) L1 writeback	∝ (1 - compression ratio) × Log size
Reliability (SafetyNet) [9]	On-chip buffers (100+ KB) + Main memory	Millions of insts	Negligible	∝ Log size
Reliability (ReVive/Rebound) [2], [7]	Main memory	Millions+ of insts	Delayed cache writeback (done in background)	∝ Log size + Cache invalidation overhead

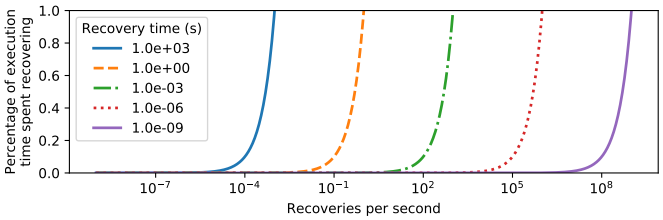

Fig. 1. In order to achieve tolerable overheads, the recovery time must shrink as the rate of recoveries increases.

possible (see Figure 1). If the recovery time is large, the overhead is tolerable only when recoveries are rare events. But as the rate of recoveries increases, systems with long recovery times perform worse, and eventually spend more time recovering state than actually computing meaningful work. This means that existing checkpointing systems preclude other, non-fault-tolerance-oriented applications of checkpointing, such as thread level speculation [10] and kilo-window processors [4], which may require many recoveries per second. In this case, a much faster recovery time (e.g., 10^{-3} to 10^{-6} seconds) is needed to ensure that the processor is not inundated with recovery operations. In cases where recoveries are extremely frequent (perhaps every hundreds or even tens of cycles), it is necessary to have a correspondingly fast recovery time (e.g., 10^{-9} seconds); such might be the case where a system is under an active attack, which may trigger a defense mechanism to repeatedly recover the system to a pre-attack state. In all cases, it is clear that as the rate of recovery increases, checkpointing applications are limited by how long it takes them to recover. By reducing checkpointing overheads, our design opens the door to new applications of course-grained checkpointing.

3 Design

Our checkpointing system follows the same basic principles as many others: periods of execution, known as *epochs*, begin when a checkpoint is taken. In a naïve checkpointing system, this may mean taking a snapshot of the entire system's state and storing it somewhere safe (e.g., DRAM) in case of a recovery. More sophisticated systems like SafetyNet [9], ReVive [7], Rebound [2], and our own design instead declare the system state at a checkpoint event to be the checkpoint data, and then only record (or "log") the data elsewhere upon the first change of the data block (typically at the cache line granularity) — this ensures that the checkpointing system doesn't needlessly store (and restore) data that doesn't change during the epoch. After some time, the epoch ends, a new checkpoint is taken, and a new epoch begins (we checkpoint whenever a userland program makes a system call). If at any point during the epoch some rollback triggering event is detected, the system can revert back to the checkpoint and resume execution.

3.1 Inline Data Compression

In our review of existing course-grained checkpointing systems, we found that restoring from checkpoints typically involves transferring large amounts of data from off-chip DRAM memory back into a processor's memory system, and also involves invalidating the processors caches [2], [7]. However, this large transfer of data and cache invalidations are what makes existing solutions perform poorly when recoveries are frequent. Our proposed checkpointing system obviates both of these issues by utilizing data compression.

In our scheme, we propose storing checkpoint data *inline* with regular program data by means of compressing both values together. This improves previous solutions for two reasons: first, by implementing our scheme in a processor's L2/L3 caches and memory, successful compression allows us to restore from on-chip checkpoint data thus avoiding additional accesses to slow off-chip DRAM; second, aggressively checkpointing in caches means that we include caches' state in the checkpoint (as opposed to those which writeback the dirty content in caches and checkpoint only memory state [2], [7]) thus avoiding cache invalidations upon recovery.

3.2 System Modifications

In our design, the system needs the following modifications:
• Additional hardware to allow for compression and decompression. BDI compression is a natural fit for this problem, as it can be achieved with relatively little hardware with minimal latency while offering reasonable compressibility [6], [13].

Transition	Abbreviation
Checkpoint	Ch
Recovery	R
Update	U
Update (compressible)	U && C
Update (uncompressible)	U && !C

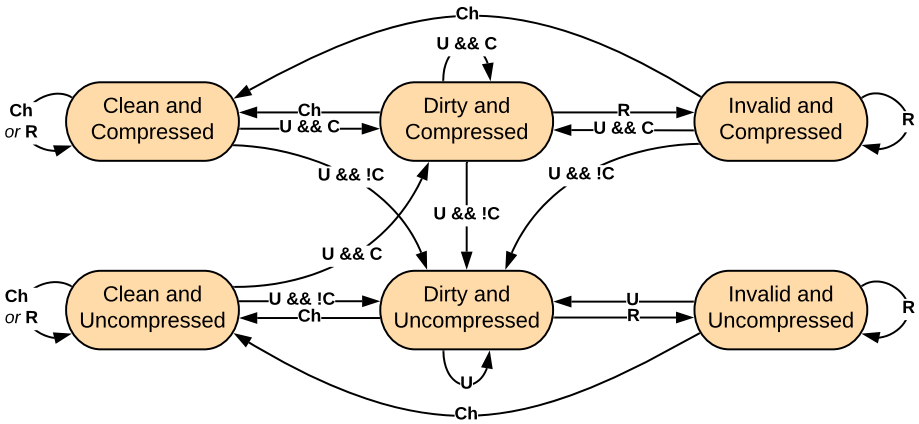

Fig. 2. A state diagram describing a cache line's transition between states, given various system events.

• On-demand and address reserved, virtual memory to log the memory state. These pages are needed in case checkpoint and update data cannot be sufficiently compressed to fit inline. In this case, our design executes microcode instructions which are responsible for logging the checkpoint data in these on-demand pages. Register state is saved in shadow registers as proposed in previous studies [2], [7], [9].
• Three additional bits of metadata per cache line granular data blocks. L2 and L3 caches where we take checkpoints require three bits of additional storage for the metadata, and for DRAM we store them in spare ECC bits similar with SPARC ADI to avoid modifications to the DIMM architecture [1]. These bits encode the six possible states of the cache line: whether the line is compressed or uncompressed, and whether it is clean, dirty or invalid.

3.3 Checkpoint and Recovery State Machine

Figure 2 presents the transition between states. The state of a cache line determines how the line must be read in order to return the correct value. If an uncompressed line is valid (clean or dirty), a read returns the line itself. If it is invalid, then the cache line contains strictly invalid data, and the read operation must find and return the correct value from the log instead (with the help of microcode execution). If a compressed line is valid, a read from the line must decompress the line and serve the more recent of the two lines. If a cache line is compressed and invalid, then the more recent value is invalid, and the read returns the older of the two decompressed values.

Our design also stipulates that following actions must be taken for the cache line updates. If an uncompressible update to a clean block occurs, the line is replaced entirely and the checkpoint data must be written to the log (with the help of microcode execution). If the cache line is in the clean and compressed state when a compressible update occurs, then the update overwrites the older of the two compressed values; otherwise if the line is dirty and compressed, the update overwrites the newer of the two.

When a checkpoint is taken, the system's current state becomes the new checkpoint. The log, which contains checkpoint data from a previous epoch, becomes obsolete and is cleared. Also the cache lines transfer to a clean state. Finally for recovery, unlike other checkpoint and recover systems, our scheme requires that no explicit actions be taken if the data is compressed except transferring the dirty states to invalid; if not, the data is restored as with previous solutions.

4 Methodology

Checkpoint and recovery systems introduce two kinds of overhead: (1) the overhead of normal execution, where checkpoints are taken but not recovered, and (2) the overhead of recovering to checkpoints. In the case of normal execution, our design behaves like existing solutions, with the exception being the compression and decompression hardware (which can be small and have marginal overheads [6]). Therefore, we expect that our design behaves similarly to existing solutions with regards to the first source of overhead, and we instead focus on demonstrating our design's advantage with regards to the second.

There are two main sources of overhead during a recovery in existing checkpoint and recovery systems. The first source of overhead is the latency of restoring the clean (checkpointed) data into the memory hierarchy. This latency is proportional to the amount of data that must be transferred back. The second source of recovery overhead comes from invalidating the caches (if necessary). In contrast to ReVive [7] and Rebound [2] which need to invalidate all the level of caches, our design only requires the L1 cache to be invalidated (since we take checkpoints in L2 and L3 caches). In this paper, we focus on the former and evaluate the amount of data our design must restore from DRAM (i.e., dirty and uncompressed) compared to a baseline scheme which must restore all the dirty data. While we don't quantify how much of an effect the latter cache invalidation has on performance, it is obvious that, all other things held equal, a system with fewer cache invalidations will perform better than a system with more frequent cache invalidations.

We use the Sniper simulator [3] to measure the average amount of compressed/uncompressed dirty data of 18 C/C++ workloads (we omit `h264ref` since the execution does not finish in a reasonable amount of time) from the SPEC CPU2006 benchmark suite [5]. We use the `test` inputs and run to completion. TABLE 2 shows the parameters of the evaluated processor.

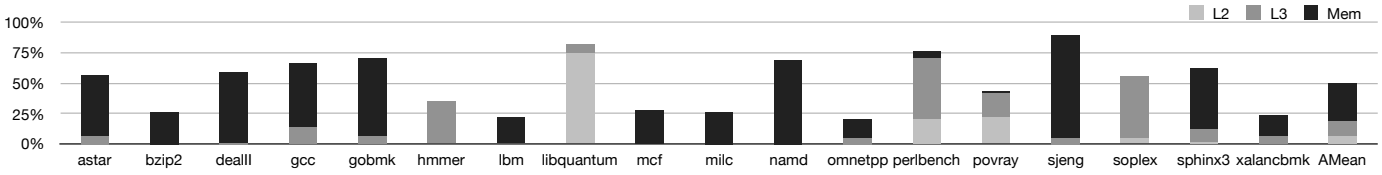

Fig. 3. Average compression ratios of dirty data at the L2 cache, L3 cache and main memory for the SPEC CPU2006 benchmark suite.

TABLE 2
Hardware configuration of the simulated system.

Core	x86-64 Intel Nehalem-like OoO core at 2.66GHz
L1 data cache	32KB, 8-way, 4-cycle latency
L2 cache	256KB, 16-way, 8-cycle latency
L3 cache	2MB, 16-way, 30-cycle latency
DRAM	DDR3-1066 like 45ns access latency, 7.6GB/s per memory controller

5 Results

Figure 3 displays the average compression ratios (sampled every 1M instructions) of dirty data residing in L2 cache, L3 cache and main memory. On average we can see that 50.5% of dirty data can be kept compressed thus avoiding half the extra memory accesses upon recovery. Using these values as a proxy for the latency of restoring data, we can conclude that our design has less overhead than existing checkpoint and recovery system. If we consider that our system also avoids the L2 and L3 cache invalidation penalties, we can conclude that our design in general recovers faster than existing designs, while retaining the performance of non-recovery operation without requiring expensive hardware.

Another interesting fact is that SPEC CPU consists of compute intensive applications with infrequent system calls: executed every 50K (`perlbench`) to 3G (`bzip2`) instructions. In general when taking checkpoints more frequently we can expect higher compression ratio because the data will be "cleaned" more often and thus our technique to be more effective. Evaluating such system call intensive applications (e.g., server workloads) is left for future work.

6 Related Work

Previous work has explored architectural support for checkpointing. SafetyNet proposes logging at main memory and in the caches [9], while ReVive proposes logging in main memory only [7]. Rebound improves on Revive by introducing a protocol for coordinated local checkpoints in multiprocessor systems [2], rather than the less-efficient global checkpoints used in Revive and SafetyNet. Such checkpoint systems are constrained in their recovery times by the size of the log (see TABLE 1). Our design is the only system which, to the best of our knowledge, aims to reduce this log size by employing inline data compression.

7 Conclusion

Better checkpointing systems with rapid recovery are needed to enable new checkpointing applications. In this paper, we presented a new checkpoint and recovery algorithm that improves the state of the art by leveraging program locality with inline data compression. Our checkpointing system outperforms existing solutions by keeping checkpoint data compressed inline with program memory, which keeps checkpoint data nearby in case of a recovery. By keeping checkpoint data nearby, our compression-based checkpointing system allows for implicit, and therefore much faster, recovery. Experimental results reveal that our approach reduces about half the memory accesses required for both checkpoint and recovery of SPEC CPU2006 benchmarks.

References

[1] "Hardware-assisted checking using Silicon Secured Memory (SSM)," https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html, 2015.
[2] R. Agarwal, P. Garg, and J. Torrellas, "Rebound: scalable checkpointing for coherent shared memory," in *Proc. of ISCA*, 2011, pp. 153–164.
[3] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.
[4] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez, "Toward kilo-instruction processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 1, no. 4, pp. 389–417, 2004.
[5] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.
[6] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proc. of PACT*, 2012, pp. 377–388.
[7] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proc. of ISCA*, 2002, pp. 111–122.
[8] Y. Shalabi, M. Yan, N. Honarmand, R. B. Lee, and J. Torrellas, "Record-replay architecture as a general security framework," in *Proc. of HPCA*, 2018, pp. 180–193.
[9] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proc. of ISCA*, 2002, pp. 123–134.
[10] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proc. of ISCA*, 2000, pp. 1–12.
[11] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proc. of ISCA*, 2003, pp. 122–135.
[12] M. Yan, Y. Shalabi, and J. Torrellas, "ReplayConfusion: detecting cache-based covert channel attacks using record and replay," in *Proc. of MICRO*, 2016, pp. 1–14.
[13] V. Young, S. Kariyappa, and M. Qureshi, "Enabling transparent memory-compression for commodity memory systems," in *Proc. of HPCA*, 2019, pp. 570–581.