ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers[*]

Jun Nakano, **Pablo Montesinos**, **Kourosh Gharachorloo**[†], and **Josep Torrellas**
University of Illinois at Urbana-Champaign [†]Google
{nakano, pmontesi, torrellas}@cs.uiuc.edu, kourosh@google.com
http://iacoma.cs.uiuc.edu

Abstract

The increasing demand for reliable computers has led to proposals for hardware-assisted rollback of memory state. Such approach promises major reductions in Mean Time To Repair (MTTR). The benefits are especially compelling for database servers, where existing recovery software typically leads to downtimes of tens of minutes. Unfortunately, adoption of such proposals is hindered by the lack of efficient mechanisms for I/O recovery.

This paper presents and evaluates ReViveI/O, a scheme for I/O undo and redo that is compatible with mechanisms for hardware-assisted rollback of memory state. We have fully implemented a Linux-based prototype that shows that low-overhead, low-MTTR recovery of I/O is feasible. For 20–120 ms between checkpoints, a throughput-oriented workload such as TPC-C has negligible overhead. Moreover, for 50 ms or less between checkpoints, the response time of a latency-bound workload such as WebStone remains tolerable. In all cases, the recovery time of ReViveI/O is practically negligible. The result is a cost-effective highly-available server.

1. Introduction

Highly-available shared-memory servers have to be able to cope with system-level faults. Faults are often transient, such as hardware glitches caused by high-energy particles, or OS panic due to unusual interleavings of software events. There are also permanent hardware faults, which can bring down part of the machine. Fault frequencies are projected to remain high in the future. This is worrisome, given the growing number of businesses with database applications that crucially depend on their servers being up practically all the time.

One approach to attain fault tolerance is to employ extensive self-checking and correcting hardware, often through redundancy and even lock-step execution. This is the approach used by HP's Nonstop Architecture [11] and IBM's S/390 mainframes [33]. Unfortunately, this approach is too expensive for many users.

An alternative approach is to use plain server hardware and support software-based checkpoint and rollback recovery. In such systems, the operating system [18, 19, 32], virtual machine monitor [5], or application (e.g., the database [8]) periodically checkpoints the state of the machine, virtual machine, or processes, respectively, to safe storage. If a fault is detected, the system rolls back to a state preceding the fault. However, since software checkpointing has significant overhead, checkpoints are typically only taken every few minutes or less frequently. As a result, when a fault occurs, the Mean Time To Repair (MTTR) is significant, and the machine becomes unavailable for a sizable period. For example, the recovery time of Oracle 9.2 on a Solaris server is typically tens of minutes [22].

A second shortcoming of software-based checkpointing appears in workloads where server and clients frequently exchange messages. To correctly support recovery, the server must delay sending messages until after they are checkpointed. If checkpoints are infrequent to minimize overheads, messages suffer long delays.

One way to significantly reduce server MTTR and avoid long message delays is to support *high-frequency* checkpointing (e.g., one every few tens of milliseconds). Several architectures with such support have been proposed [21, 24, 29, 34]. These architectures rely on hardware assistance for checkpointing or for data buffering, logging or replication. For example, ReVive induces about 6% overhead and recovers from the types of faults supported in less than 1 second [29]. Such tiny MTTR boosts machine availability. Moreover, as suggested by the ROC project, it opens up opportunities to lower cost of ownership [27].

Unfortunately, past work on these high-frequency checkpointing architectures has focused on recovering the *memory state* of the machine. It has not fully addressed the problem of rollback recovery in the presence of I/O. When workloads perform I/O, rollback is tricky: how can the server "undo" a disk write or a message send? Can it "redo" it? Unless these issues are addressed, the proposed high-frequency checkpointing solutions are unusable. These issues are also particularly relevant to architectures for transactional memory [10], which rely on the ability to roll back a section of code and then re-execute it.

A known approach to handle I/O in checkpointing systems is to delay the commit of output until the next checkpoint (output commit problem). To accomplish this, Masubuchi *et al.* [21] proposed adding a "virtual" or "Pseudo" Device Driver (PDD) layer between the kernel and the Device Drivers (DD). Disk output requests are redirected to the PDD rather than the DD. The PDD blocks any output-requesting process until the next check-

[*]This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel.

point [20], after which the output is performed. Masubuchi *et al.*'s design has limitations, such as (i) blocking processes until a checkpoint and (ii) only supporting disk I/O. However, their general method is attractive because it requires no kernel or application modification. It can be built upon to provide efficient I/O undo/redo for high-frequency checkpointing architectures.

1.1. Contribution of This Paper

Our main contribution is the full implementation, testing, and experimental evaluation of *ReViveI/O*, an efficient I/O undo/redo scheme that is compatible with high-frequency checkpointing architectures such as ReVive [29] and SafetyNet [34]. Our work completes the viability assessment of such novel memory-recovery architectures. It is only through a complete implementation that we identify true overheads, relevant ordering constraints, and corner cases. Moreover, we perform a sensitivity analysis of what checkpoint frequencies are required to maintain acceptable throughput and tolerable response times.

We also enhance Masubuchi *et al.*'s approach in two ways. First, the PDD now also supports network I/O. Secondly, the disk PDD, rather than blocking the output-requesting process, quickly buffers the output and returns. After the next checkpoint, the I/O operation is committed in the background. This provides efficient I/O undo/redo.

We installed our ReViveI/O prototype on a Linux 2.4-based multiprocessor server running TPC-C on Oracle, and WebStone on Apache. Our prototype shows that low-overhead, tiny-MTTR recovery of I/O is feasible. Specifically, for 20–120 ms between checkpoints, a throughput-oriented workload such as TPC-C has negligible overhead. In addition, for 50 ms or less between checkpoints, the response time of a latency-bound workload such as WebStone on Apache remains tolerable. In all cases, the recovery time of ReViveI/O is practically negligible. Finally, combining ReVive and ReViveI/O is likely to reduce the throughput of TPC-C-class applications by 7% or less for 60–120 ms checkpoint intervals, while incurring a tiny MTTR of less than 1 second.

Our work is significant in that, with ReVive and ReViveI/O, a shared-memory server can quickly recover from: (i) any hardware (and some software) transient faults in the machine, and (ii) permanent faults that at most take out one node in the machine. Indeed, both the processor/memory state (thanks to ReVive) and the I/O state (thanks to ReViveI/O) are restored to the preceding checkpoint *within 1 second and transparently* to the database. No ongoing database transactions are lost.

There are rare faults for which ReVive cannot restore the processor/memory state, such as the simultaneous permanent loss of multiple nodes. In this case, the fault is not transparent to the database. A few seconds after the machine is rebooted, ReViveI/O brings the I/O state to its correct state at the preceding checkpoint. Then, we simply depend on the normal recovery mechanisms of the database to reconstruct the state from the logs saved on disk.

The overall result is much higher server availability: the majority of faults are recovered from with sub-second MTTR and transparently, while only infrequent faults require the much slower recovery mechanism of the database.

The paper is organized as follows: Section 2 gives background; Sections 3 and 4 present ReViveI/O's architecture and implementation; Section 5 describes our evaluation methodology; Section 6 evaluates ReViveI/O; and Section 7 discusses related work. Note that fault detection is beyond the scope of this paper.

2. Background

2.1. Context of Our Work

The context of our work is shared-memory multiprocessors such as IBM's eServer pSeries p5 595 [14] or HP's Integrity Superdome [12] used as back-end database servers. These servers store the database in local disk subsystems and communicate over networks with many clients. They execute transaction-processing applications similar to TPC-C.

A major issue in these systems is server uptime. Unfortunately, a high-energy particle impact may cause a processor reset, an unusual data race may crash the OS, or a link failure may disconnect a node. In these cases, transactions are typically aborted and the database attempts to recover. Such recovery often renders the server unavailable for tens of minutes [22].

To understand the recovery requirements of these systems, note that I/O is practically limited to disk and network. Moreover, these workloads are typically not latency bound. For example, in TPC-C, 88% of transactions are NewOrder or Payment, which involve the exchange of a single request and response between client and server. IBM's p5 595 reports an average response time of 340 ms for these transactions [37]. Consequently, adding a few tens of ms to each transaction to support a recovery scheme is tolerable.

2.2. Fault Model

We leverage proposed rollback-recovery architectures [21, 24, 29, 34] that support high-frequency checkpointing (ten times or more per second) and, for the fault types supported, are able to recover the *memory state* of the machine before the fault. These schemes typically have low overhead and a tiny MTTR.

As an example, we use ReVive [29] in this paper. Appendix A outlines ReVive. With 100 ms between checkpoints, ReVive has an average execution overhead of 6.3%. Moreover, it recovers from the supported faults in under 1 second. This results in 99.999% availability even with one fault per day.

Specifically, ReVive recovers the memory state of the machine for: (i) transient faults and (ii) permanent faults that at most take out one node in the machine. Although fault detection is beyond the scope of this paper, the implicit assumption is that some mechanism detects these faults within a checkpoint interval. Such short detection latency is more feasible for hardware faults than for software ones. However, there are some software transient faults that are fail fast. For example, Gu *et al.* [9] show that a sizable portion of kernel errors can be detected within 100,000 cycles. Overall, we refer to all these faults, from which ReVive can recover the memory state, as *Memory-Recoverable* (MR) faults.

The other faults, from which ReVive cannot recover the machine's memory state, we call *Non-Memory-Recoverable* (NMR). An example is the simultaneous permanent loss of multiple nodes [29].

In this paper, we also assume that non-volatile storage, namely disks and any closely-attached non-volatile memories (NVRAMs), can only suffer transient faults. They have the appropriate support (e.g., RAID 5) to avoid permanent faults.

With these assumptions, we will show that, for MR faults, we restore both the processor/memory state (thanks to ReVive) and the I/O state (thanks to ReViveI/O) to the preceding checkpoint. The restoration is *transparent* to the database. No ongoing transactions are lost (Figure 1).

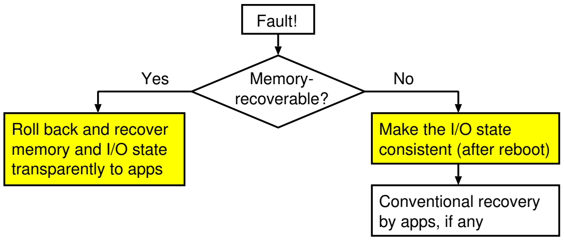

Fault!
Memory-recoverable?
Yes
No
Roll back and recover memory and I/O state transparently to apps
Make the I/O state consistent (after reboot)
Conventional recovery by apps, if any

Figure 1. Faults handled by the combination of ReVive and ReViveI/O.

For NMR faults, recovery is not transparent to the database. The machine has to be fixed and rebooted. ReViveI/O then restores the I/O state to its consistent state at the preceding checkpoint. Finally, the conventional recovery mechanisms of the database reconstruct the database state.

2.3. Integrating I/O with Checkpoints

Work on checkpointed message-based distributed systems [7, 17] shows how to support I/O undo/redo under checkpointing. The commit of outputs is delayed until the next checkpoint (output commit problem); only then can the system guarantee that it will not have to roll back to a state prior to issuing the outputs.

To address the output commit problem without kernel modifications, Masubuchi *et al.* proposed the Pseudo Device Driver (PDD) [21] (Figure 2). Disk output requests are redirected to the PDD rather than the Device Driver (DD). The PDD blocks any output-requesting process until the next checkpoint [20], after which the output is performed. The PDD can be considered an extremely thin virtual machine layer for I/O checkpointing.

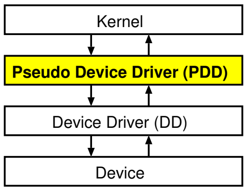


Figure 2. The Pseudo Device Driver software layer.

We enhance Masubuchi *et al.*'s scheme in two ways. First, processes requesting disk writes are not blocked until the next checkpoint. Secondly, we also support network I/O. Kernel, DDs, and server/client applications remain unmodified.

3. Architecture of ReViveI/O

This section describes the organization ReViveI/O, with key ordering issues, overheads, and limitations.

3.1. Description of Operation

We start by examining three properties that we leverage. Then, for readability, we describe ReViveI/O in two steps: first, an initial incomplete solution, and then the complete one.

3.1.1. Properties Leveraged

We leverage three properties to build a low-overhead I/O undo/redo prototype. First, ReVive's ability to roll back the memory state is leveraged to restore PDD consistency after a fault. Specifically, we assign to the PDD a portion of main memory called the *Memory Buffer*. In there, the PDD buffers all the output requests until the next checkpoint; after the checkpoint, the outputs are performed in the background and removed from the buffer. If a fault occurs, ReVive returns the memory state to the previous checkpoint. This automatically makes the PDD consistent: all the output requests in the current checkpoint interval disappear from the Memory Buffer, and all those from the previous checkpoint interval re-appear in the Memory Buffer and are ready to be performed again.

Second, the fact that the output operations under consideration are idempotent (i.e., replayable) is leveraged to allow the recovery scheme to re-perform output operations without hurting correctness. Indeed, disk output operations are trivially idempotent. Network output is idempotent due to the high-level support provided by TCP [36]. With TCP, each packet has a sequence number. If the client receives the same packet twice, TCP sees the same sequence number and discards one of them[1]. Consequently, correctness is not compromised when, after a rollback, the requests in the Memory Buffer force our scheme to re-write the same disk blocks and re-send the same messages.

Finally, properties of the I/O considered are leveraged to not have to buffer any inputs for later "re-consumption" should rollback be needed. Specifically, disk inputs need no buffering because the application will automatically re-issue them if it needs to. For network input, we avoid buffering by again relying on TCP properties. With TCP, packets are acknowledged by the receiver; if the client does not receive an acknowledgment (ACK) from the server within a timeout period, it resends the packet. In our design, ACKs, like all outgoing messages, are delayed by the server until after the next checkpoint.

Consequently, suppose that the server receives an input message, issues an ACK that gets stored in the Memory Buffer, and a fault occurs. Two cases are possible. First, if the fault occurs before the end of the next checkpoint, the rollback removes the effect of the input message from the server, as well as the ACK from the Memory Buffer. In this case, the ACK is not sent and the client will resend the input message. If, instead, the fault occurs after the next checkpoint, the rollback removes neither the effect of the input message from the server, nor the ACK from the Memory Buffer. This case is also correct because the ACK will eventually be sent. In either case, the server does not need to buffer network input.

3.1.2. Initial Incomplete Solution: BufferVolatile

All network and disk output requests issued by the application (*OutReq1* and *OutReq2* in Figure 3-(a)) are transparently intercepted by the PDD and buffered in the Memory Buffer. The buffered information includes the output data and metadata such as the destination block number in the device. After the next

[1] The User Datagram Protocol (UDP) does not provide TCP's support to eliminate duplicates. UDP is unreliable by definition, and the application (e.g., NFS over UDP) is responsible for dealing with duplicate and lost packets. Consequently, we only focus on TCP.

checkpoint (*C2* in Figure 3-(a)), the PDD passes the information to the DDs, which perform the output operations (e.g., DMA writes to disk or to the network card) in the background (*OutOp1* and *OutOp2* in Figure 3-(a)).

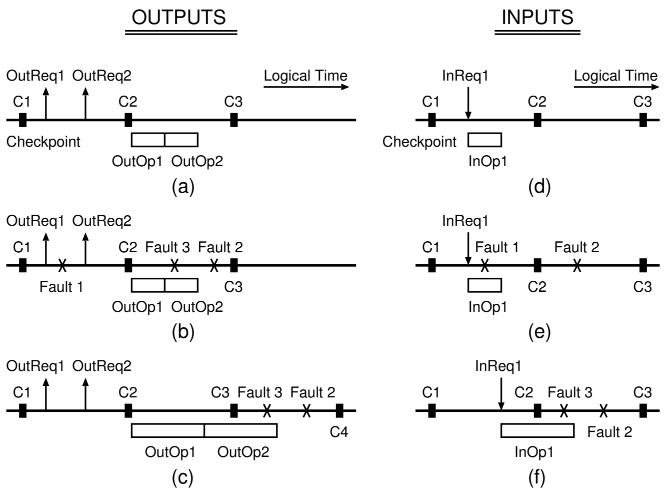


Figure 3. I/O operations and faults in different scenarios. In the figure, *InReq*, *OutReq*, *InOp*, and *OutOp* mean input request, output request, input operation, and output operation, respectively.

Consider now input requests, such as reads from the disk or the network card. On receiving the request (*InReq1* in Figure 3-(d)), the PDD checks if the requested data is in the Memory Buffer. If so, the data is provided. Otherwise, the PDD passes the request to the DD, which performs the operation in the background (*InOp1* in Figure 3-(d)). As indicated in Section 3.1.1, no buffering is needed.

We call this initial solution *BufferVolatile*. With it, if an MR fault (Section 2.2) occurs, the server recovers both memory and I/O states *transparently* to the running application. Consider the four possible timeframes wherein a fault can occur.

1. Fault before the end of the checkpoint that immediately follows the I/O request (*Fault 1* in Figures 3-(b) and (e)). In this case, ReVive rolls back the memory state to the previous checkpoint *C1*. As a result, the Memory Buffer loses any record of output request *OutReq1*. This automatically "undoes" *OutReq1*, as desired. Thus, *OutOp1* is not performed. As for input I/O, since the rollback operation involves resetting the devices, any ongoing input operation such as *InOp1* is aborted.

2. Fault after the end of the checkpoint that immediately follows the I/O request; the I/O is already performed (*Fault 2* in Figures 3-(b) and (e)). ReVive rolls back the memory state to the previous checkpoint *C2*. The only interesting case is for outputs. The Memory Buffer gets restored to the state it had at *C2*, where it contained a record of the output operations to perform. Consequently, the PDD will eventually automatically re-issue *OutOp1* and *OutOp2* to the DDs. This is correct because of the idempotent nature of the I/O in consideration.

3. Like Case 2 but the background I/O is not yet completely performed when the fault occurs (*Fault 3* in Figure 3-(b)). As the system rolls back, the devices are reset and the ongoing I/O is aborted. Then, all I/O operations (*OutOp1* and *OutOp2*) will eventually be performed again.

4. Special case: Fault in an interval preceded by a checkpoint overlapped with an I/O operation. Sometimes, an I/O operation initiated before a checkpoint extends past it. This is seen for *OutOp2* in Figure 3-(c) and *InOp1* in Figure 3-(f). If a fault such as *Fault 2* or *Fault 3* in these figures occurs, the memory state rolls back to the checkpoint that overlapped with the I/O operation. During the recovery process, the I/O operation (*OutOp2* or *InOp1*) gets killed, since all I/O devices (disk controller and network adapter) get reset. This is discussed in Section 3.3.2. Unfortunately, the rollback would leave the memory state in inconsistent state: while the I/O operation is killed, it is incorrectly marked "in progress", and it is only partially performed.

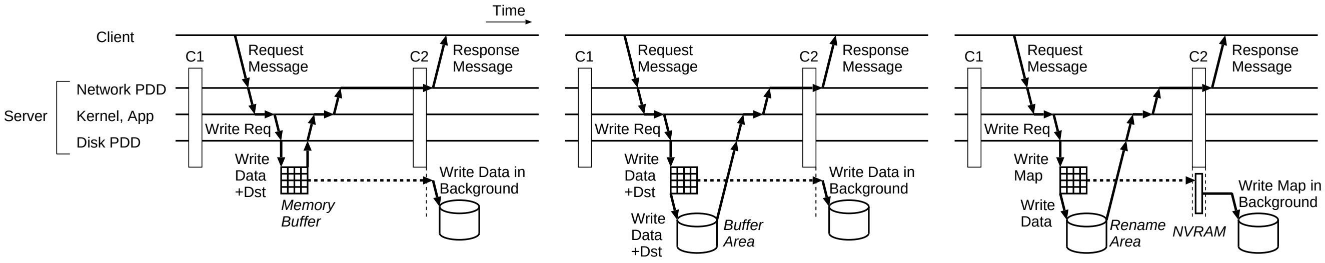
To solve this problem, after the recovery process rolls back the memory state to the previous checkpoint, the PDD re-issues to the DDs any checkpoint-overlapping I/O operation *from the beginning*. Note that the PDD can find out what are the I/O operations that are (incorrectly) marked "in progress" (*OutOp2* and *InOp1*). Optionally, the PDD can skip re-issuing the network input operations that overlapped with the checkpoint: there is no need to re-initiate the transfer of data from the network adapter to memory because TCP will ensure that incoming packets are retransmitted.

Overall, our *BufferVolatile* scheme ensures database consistency in an environment with MR faults (and/or transient faults in non-volatile storage as per Section 2.2). For example, assume that a client starts a transaction that involves writes to disk (Figure 4-(a)). After the disk PDD has buffered the data and destination block number in the Memory Buffer, the database sends a response message to the client. The message is buffered by the network PDD. After the next checkpoint, the message is sent and the write is issued to the disk. If an MR fault occurs before this checkpoint, *BufferVolatile* will roll back and eventually receive the automatic retransmission of the original request message after the timeout. Moreover, if the PDD observes a transient disk error when it issues the write, it will simply retry until it succeeds. In any case, when the client receives the response message, it can assume that the disk write in the transaction has been committed. Some ordering issues are examined in Section 3.2.

3.1.3. Complete Solutions: Buffer and Rename

BufferVolatile is inadequate if an NMR fault (Section 2.2) occurs, such as the simultaneous permanent loss of multiple nodes. As an example, consider Figure 4-(a) after the second checkpoint. Suppose that an NMR fault occurs after the response to the client is sent but before the disk is updated in the background. ReVive cannot restore the contents of the Memory Buffer and, therefore, reissue the buffered write to disk. The disk is left in a state that is inconsistent with the information passed to the client. Note that conventional database recovery mechanisms cannot help: the missing write can be a log write, without which the database cannot redo the operation.

To solve this problem, we enhance *BufferVolatile* to ensure that the PDD also saves the output request information in non-volatile "temporary" storage before the next checkpoint. If an NMR fault occurs as in the example just described, we can copy the information from the non-volatile temporary storage to the server disk, and thus make the disk consistent. Then, we can rely on the con-



(a) BufferVolatile (b) Buffer (c) Rename

Figure 4. Operation of ReViveI/O. The scenario depicts a transaction with a disk write. NVRAM and *Dst* stand for Non-Volatile RAM and the destination block number, respectively.

ventional mechanisms of the database for recovery, although the downtime will be longer (Figure 1).

We propose two alternative schemes, called *Buffer* and *Rename*, as shown in Figures 4-(b) and (c), respectively. *Buffer* is based on temporarily buffering output request data, and is conceptually simpler. *Rename* is based on renaming the data, and can be more efficient because it requires fewer disk writes. To describe the schemes, we focus on disk I/O because network I/O does not distinguish between the schemes.

In *Buffer*, a disk write request updates the Memory Buffer and a disk buffer area before returning (Figure 4-(b)). The update includes both the data and some metadata such as the destination block number. To speed up this operation, the updates to the disk buffer are done on sequential blocks. Moreover, the disk buffer can be a dedicated small, fast disk, similar to the Disk Caching Disk [13]. After the next checkpoint, all data in the Memory Buffer are copied to their true locations on the main disk. In fault-free conditions, the disk buffer is never read.

In *Rename*, a write request writes the output data to a new disk block in a rename area, and saves the new logical-to-physical block number mapping in the Memory Buffer before returning (Figure 4-(c)). During the checkpoint, the mappings in the Memory Buffer are copied to a small (e.g., 32 MB) Non-Volatile RAM (NVRAM) associated with the disk. Later, the mappings in the NVRAM are committed to disk in the background.

The NVRAM is not a single point of failure. Specifically, there are commodity disk adapters with internal NVRAM where the NVRAM is transparently backed up by an additional copy of the data. An example is IBM's Fast Write Cache [15]. Moreover, transient errors in the NVRAM or associated disk are handled by the PDD retrying the request. Recall that we assume there are no permanent faults in non-volatile storage.

Note that these NVRAMs usually work asynchronously — the data is destaged to the disk only when the NVRAM is getting full or the data has stayed in the NVRAM for a certain time. Therefore, if a fault occurs during a checkpoint, an asynchronous NVRAM could incorrectly write its mappings to disk while we are rolling back to the previous checkpoint. This problem is solved by also storing the original mapping information in the NVRAM, so that if the problem occurs, we can undo the changes of mapping.

An important design issue in *Rename* is the policy for allocating the renamed blocks. One option is to write the new blocks sequentially on a free disk area, just like the log-structured file system (LFS) [31]. With this design, occasional defragmentation of the disk may be needed. Another option is to map each block to either one of two physically consecutive blocks in the disk as in TWIST [30]. This design requires double disk space for blocks. However, the mapping information per block is only one bit, compared to (typically) 8 bytes for the first design.

With *Buffer* or *Rename*, the disk can always be brought to the state corresponding to the checkpoint immediately preceding the fault, even for an NMR fault. Indeed, consider a disk write request. If the fault happens between the request and the end of the next checkpoint, the contents of the disk buffer are discarded (*Buffer*) or the new mapping information is not written to NVRAM (*Rename*); the main disk remains unmodified and the client is never notified. If, instead, the fault happens after the next checkpoint, a disk update is guaranteed to occur: the data and metadata in the disk buffer (*Buffer*) or the mapping in NVRAM (*Rename*) are used to update the disk.

3.2. Ordering Issues

Our schemes (*Buffer* and *Rename*) change the timing of I/O operations. However, they always satisfy the ordering properties of I/O in databases, which require that logs are fully committed before data is. This is accomplished by updating the Memory Buffer and the disk buffer or rename area in the order that the disk write requests are received. Moreover, after the subsequent checkpoint, all updates are guaranteed to commit to their final storage locations.

As the updates commit after the checkpoint, our schemes do not force disk write serialization across different block addresses. Instead, the data blocks buffered in the Memory Buffer (or mappings in the NVRAM) are written to the final disk location in an overlapped manner. They can even proceed with some re-ordering. Overlapping and re-ordering enables higher performance without affecting correctness. In theory, the performance could be even higher than a system without recovery.

We have seen that in all cases, if a client receives a transaction-completion message (Figure 4), then disk updates are guaranteed to commit. Still, it is possible that the client receives the completion message and sends another request before the server has physically finished the disk writes. Even if this request needs to read the data that is still being written to disk, no race occurs. The reason is that the PDD will automatically redirect the request to the Memory Buffer, which only deallocates an entry when the final disk update is completed.

3.3. Overheads and Recovery Latency

3.3.1. Overheads

ReViveI/O increases the latency of network messages because the server PDD does not send packets until after the next checkpoint. In practice, back-end database servers running TPC-C class applications are not particularly latency bound. Adding tens of ms to each transaction to support recovery is tolerable.

However, we tune TCP in two ways. First, since packets now take longer to be acknowledged, we increase the sliding window [36] that buffers yet-to-be-acknowledged packets. There is no danger of buffer overflow because TCP throttles packet sending as a buffer becomes full. Second, since the server PDD sends packets after checkpoints, the packet round trip time becomes more variable. This additional variability disrupts TCP's flow control mechanism. To solve this problem, the server PDD does not send all the packets as fast as it can after a checkpoint; instead, it smooths out the traffic.

ReViveI/O's impact on computation, memory, and disk accesses may also affect application throughput. Consider computation and memory accesses. For each output request, ReViveI/O requires an initial write and a later read to the Memory Buffer (Figure 4). For network I/O, the data written/read is only a pointer to the socket buffer; the actual packet payload is retained in the sliding window elsewhere in memory. For disk I/O, the data written/read is the block data and metadata (*Buffer*), or the mapping only (*Rename*). For *Rename*, the mapping is also written to the NVRAM at checkpoints. In addition, the PDD executes bookkeeping code to manage the Memory Buffer.

For disk accesses, *Buffer* performs two disk writes per write request, although only one is in the critical path (Figure 4-(b)). The one in the critical path is fast because the disk buffer is written sequentially, and it can be a small, fast disk. *Rename* performs two writes on the same disk per request: one for the data and one for the new mapping (Figure 4-(c)). The first one is in the critical path; the second one can be merged with other updates of mappings from the same checkpoint interval. In addition, *Rename* may require periodic disk block compaction.

3.3.2. Recovery Latency

The latency of a recovery depends on the type of fault. Consider first an MR fault. Figure 5 shows a ReVive recovery time-line from [29], for the worst MR fault: the permanent loss of one node at a checkpoint. The latency numbers assume a 100 ms checkpoint interval. On top of the ReVive recovery, the thick up-arrows show the three actions performed by ReViveI/O.

Immediately after fault detection, ReViveI/O resets the I/O devices, namely network card and disk. This operation kills any ongoing DMA, which could overwrite the data being restored in the rollback. This operation is quick — 1 ms or less. It is also device dependent: it involves writing to a special I/O port to reset the network card, and sending a signal to reset the disk controller. It does not require any slow disk access.

After the memory state has been rolled back to the checkpoint, ReViveI/O re-initializes the device drivers (Figure 5). They have been left in an inconsistent state relative to the reset devices. This operation involves updating data structures in memory such as buffers and pointers, and bringing back the device driver's configuration parameters. It typically takes 10 ms or less.

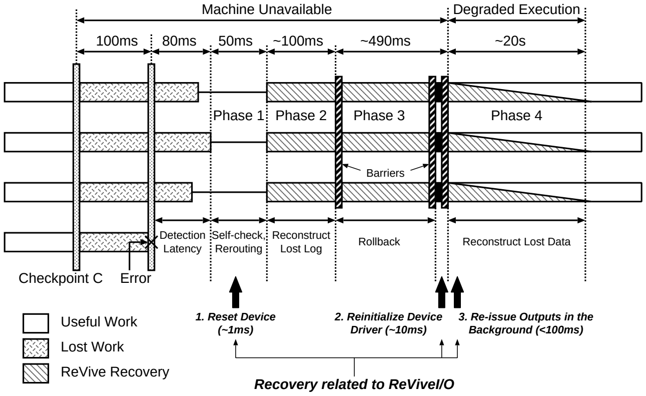


Figure 5. Recovery time-line for the permanent loss of one node at a checkpoint. ReViveI/O actions are shown with thick arrows.

Finally, after application execution has resumed, ReViveI/O performs in the background all the output operations needed to bring the I/O state to the checkpoint immediately preceding the fault. Performing all these operations is easy. Indeed, for *Buffer*, the rolled-back Memory Buffer has an accurate record of all such operations. For *Rename*, the Memory Buffer has the record for the network operations, while the NVRAM has the record of the disk mappings to save. Performing these actions degrades the machine's performance for tens of ms, but does not make it unavailable.

Overall, the three ReViveI/O-related recovery actions negligibly add to the unavailable time and keep the fault transparent to the database.

Consider now an NMR fault. All currently-executing transactions abort and the system is typically rebooted. Before the database can use its own recovery mechanisms, ReViveI/O brings the disk to the state at the checkpoint immediately preceding the fault. This is done as follows: for *Buffer*, the disk output information is recovered from the disk buffer area; for *Rename*, the disk mappings are recovered from the NVRAM. The latency of performing these actions is much smaller than the time required for rebooting or for the database to recover. Therefore, ReViveI/O again adds negligibly to the unavailable time.

Finally, a second fault may occur while recovering. If ReVive can recover the memory state, ReViveI/O re-executes the three actions shown in Figure 5. Otherwise, ReViveI/O brings the disk to a consistent state after reboot as just described.

3.4. Limitations

ReViveI/O has several applicability limitations. First, it is not applicable to latency-critical workloads, such as those with user interaction through graphics, keyboards, or other devices.

ReViveI/O relies on system-level code to intercept I/O requests, buffer them, and perform the operations later. This approach rules out, as they are currently implemented, user-level I/O and I/O coprocessors such as TCP Offload Engines (TOEs). User-level I/O relies on user libraries to perform I/O, eliminating kernel involvement. For example, uncached accesses from user mode to the network interface send messages without involving the kernel. To be able to support schemes similar to ReViveI/O, we would have to add a PDD component to the user libraries

TOEs implement TCP operations in hardware. The kernel is not involved in performing the low-level operations in packet handling. Again, to support schemes similar to ReViveI/O, we would have to modify the TOE hardware to perform the PDD operation, namely buffer the packet for later issue. We would also have to synchronize the processor and the TOE at checkpoints.

We feel that, while user-level I/O and TOEs are interesting alternatives, they are still new technologies with poor standardization and, as a result, are hard to maintain in large server installations. The standard software TCP solution that ReViveI/O supports is overwhelmingly the most popular one.

4. Implementation Aspects

We have implemented a ReViveI/O prototype on a multiprocessor server with two 1.5 GHz AMD Athlon processors, 1.2 Gbytes of memory, two 80-Gbyte IDE disks, and a 1 Gbit ethernet card. One of the two disks is used as a disk buffer. The server runs Linux 2.4. The PDD is about 2,000 lines of C code for the disk and 2,000 for the network. The DDs, the Linux kernel, and the applications remain unmodified.

Note that our server does not have ReVive hardware. Consequently, when needed, we simulate its effect. Specifically, to test recovery, we pretend that a fault occurs immediately after a checkpoint and, therefore, the memory state rolls back instantly. ReViveI/O can then proceed with resetting the devices, re-initializing the DDs and issuing all the buffered outputs. Since the memory state recovery and the I/O state recovery are conveniently decoupled (Figure 5), we can test the correctness of the I/O-related recovery without memory-checkpointing hardware. Under fault-free conditions, we do not model ReVive. However, in Section 6.3, we estimate the combined overhead and availability of ReVive and ReViveI/O.

Except for the ReVive support, we have thoroughly tested the prototype under many workload conditions (e.g., heavy disk writes, frequent small messages, or bulk data transmission) and restart scenarios (e.g., DMA in progress or many pending I/O requests). We also injected different faults that allowed us to test most software paths. In the following, we outline some implementation aspects.

4.1. Support for Disk I/O

ReViveI/O can be designed for disks accessed through a file system or as raw devices. In our prototype, we use a file system. Figure 6 expands Figure 2 showing the interface between kernel, PDD, and disk DD [4] for *Buffer* and *Rename*. The modules in shaded pattern are those added for ReViveI/O: PDD, Memory Buffer, disk buffer or rename area, and NVRAM.

In a conventional system, a read request causes a buffer cache access. If a miss occurs, the low-level DD satisfies the request. In a write, a block is allocated in the buffer cache if it is not already there. The block is updated and marked dirty. Sometime later, the kernel writes it to disk.

With ReViveI/O, such dirty block writes are directed to the PDD. As indicated before, the PDD buffers the information and commits it after the next checkpoint. Although individual DMA operations (e.g., setup, execution, postprocessing) performed by the low-level DD may not be idempotent, disk writes at the PDD level are idempotent because the PDD triggers these low-level operations as an indivisible operation.

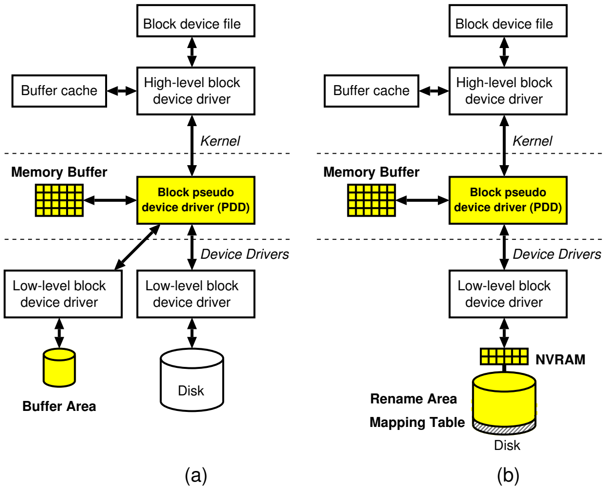

Block device file
Buffer cache
High-level block device driver
Kernel
Memory Buffer
Block pseudo device driver (PDD)
Device Drivers
Low-level block device driver
Low-level block device driver
Buffer Area
Disk
(a)
Block device file
Buffer cache
High-level block device driver
Kernel
Memory Buffer
Block pseudo device driver (PDD)
Device Drivers
Low-level block device driver
NVRAM
Rename Area
Mapping Table
Disk
(b)

Figure 6. Interface between kernel, PDD, and disk DD for *Buffer* (a) and *Rename* (b).

The Memory Buffer is sized based on the machine's total disk bandwidth and the checkpoint interval. The same applies to the NVRAM except that the bandwidth is per disk, since ReViveI/O has one NVRAM per disk. For example, a 100 MB/s disk array in a 100 ms checkpoint interval can consume 10 MB. Consequently, for *Buffer*, a 20 MB Memory Buffer and a 20 MB disk buffer area suffice. For this update rate, *Rename* generates about 20 KB of mappings per checkpoint interval. Consequently, for *Rename*, a 20 MB disk rename area, 40 KB Memory Buffer, and 40 KB NVRAM suffice. Since the checkpoint operation takes about 1 ms [29], an NVRAM built out of battery-backed SRAM has sufficient bandwidth (∼100 MB/s [35]) to load these 20 KB mappings during a checkpoint.

4.2. Support for Network I/O

The kernel does not use the usual interface (e.g., `eth0`). Instead, it uses the virtual interface provided by the network PDD (say, `veth0`). The data structure passed between the kernel and the network DD is the socket buffer, which contains the length of the packet, a pointer to the packet, and other fields. When the kernel passes a socket buffer to the PDD, a pointer to it is copied to the Memory Buffer. We copy only the pointer to reduce overhead. After the checkpoint, the socket buffer is passed to the network DD.

When an input packet arrives at the network card, the appropriate handler is triggered in the network DD, which in turn calls the `netif_rx` function in the kernel to process the packet. TCP would get confused if the kernel sent a packet through `veth0` and received the reply from `eth0`. Consequently, the DD call is routed through a `netif_rx` function in a special library that changes the device field of the socket buffer to `veth0`. Neither DD nor kernel are modified.

Workload		Description	I/O
Micro-Benchmarks	RandomWrite	Repeatedly write blocks of a given size to disk. The writes are synchronous and directed to random locations. Size can be set.	Disk
	SequentialWrite	Like RandomWrite but writes are directed to sequential locations.	Disk
	Iperf [16]	Repeatedly send messages of a given size. Size can be set.	Network
Throughput oriented	TPC-C-like on Oracle 9.0.2	32 warehouses, 30 remote clients, no think time, 400-Mbyte database buffer, and 4-Kbyte blocks	Disk and network
Latency bound	WebStone 2.5 with Apache 2.0	Memory resident, variable number of remote clients, no think time, 85 HTML documents of 100 KB on average	Network

Table 1. Workloads used in the evaluation. We use the term "TPC-C-like" because compliance with the specification is not fully checked.

5. Evaluation Methodology

We evaluate the three schemes of Table 2. Of the ReViveI/O approaches, we select *Buffer* for evaluation. *NoRollback* is the unmodified server, which has no provision for I/O undo/redo. *Stall* is a scheme for disk I/O similar to Masubuchi *et al.* [20, 21]. In *Stall*, there is no data buffering; requesting processes are not notified of output I/O completion until the next checkpoint.

Scheme	Description
Buffer	ReViveI/O approach. Supports I/O undo/redo for disk & network I/O.
NoRollback	Unmodified server. No provision for I/O undo/redo.
Stall	Output I/O blocks until next checkpoint. Scheme for disk I/O only. Similar to Masubuchi *et al.* [20, 21].

Table 2. Schemes evaluated.

We run the workloads of Table 1: a throughput-oriented one (TPC-C on Oracle 9.0.2), a latency-bound one (WebStone [38] on the Apache server [1]), and several microbenchmarks.

We experiment with 20-240 ms checkpoint intervals. For each checkpoint interval, we set the TCP sliding window size to buffer all unacknowledged packets at the 1 Gbit ethernet bandwidth, and the Memory Buffer size to hold all the data that can be written to disk. For 80 ms intervals, this is 12 MB for the sliding window and 8 MB for the buffer. The ratio is the same for the other intervals.

6. Evaluation

To evaluate our ReViveI/O prototype, we measure its overhead in fault-free execution (Section 6.1) and the latency of fault recovery (Section 6.2). Then, we project the impact of combining ReViveI/O and ReVive (Section 6.3).

6.1. Execution Overhead

6.1.1. Disk I/O Microbenchmarks

The RandomWrite and SequentialWrite microbenchmarks test worst-case disk I/O conditions. They consist of a loop that synchronously writes blocks of a given size to disk. Consequently, the disk is constantly busy. In our server, one disk can support up to 32 Mbytes/s of write throughput, while the other (used as disk buffer) up to 36 Mbytes/s. With such hardware, Figure 7 shows the resulting system throughput as a function of the size of the blocks written. We consider 20 and 160 ms checkpoint intervals.

We see that the various overheads of *Buffer* (Section 3.3.1) do not reduce throughput relative to *NoRollback* under heavy disk write traffic. In fact, *Buffer*'s throughput is slightly higher than *NoRollback*'s. The reason is that a write request in *Buffer* returns as soon as the data is written to the faster disk *sequentially*, rather than to the other disk randomly (RandomWrite) or sequentially (SequentialWrite).

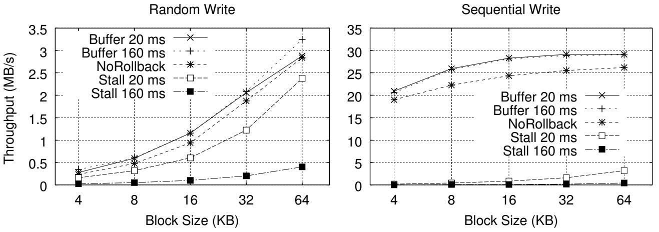


Figure 7. Disk I/O throughput for random and sequential synchronous writes.

The checkpoint interval size has little effect on *Buffer*. The reason is that PDD operations have only tiny overhead. Finally, *Stall* delivers a very low throughput; effectively, *Stall* manages only a single synchronous write per thread per checkpoint.

6.1.2. Network I/O Microbenchmark

The Iperf microbenchmark measures the maximum TCP bandwidth. Figure 8 shows the sustained throughput as a function of the message size for two cases: our server sends messages to one client (Unidirectional) and both client and server send messages to each other (Bidirectional). The case where the client sends messages to the server is similar to Unidirectional — messages have the same Round Trip Time (RTT) because the server delays all packets, including ACKs. In this experiment, *Stall* does not apply.

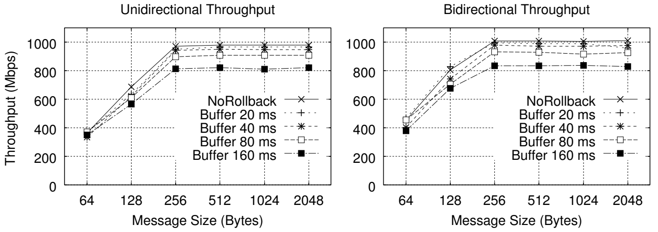


Figure 8. Uni- and bi-directional throughput between one client and the server over 1 Gbit ethernet.

We see that, under these extreme conditions, *Buffer* lowers the throughput relative to *NoRollback*. This is due to PDD overheads (Section 3.3.1) and suboptimal TCP operation in a high-bandwidth, high-latency network. However, the throughput reduction is modest: it ranges from 5% with a 20 ms interval to less than 20% with 160 ms.

Finally, *Buffer* also increases packet RTT. The resulting impact on the response time depends on the application. The impact is tolerable in applications where the server performs substantial work or where the communication pattern involves bulk data transmission. We consider this issue next.

6.1.3. Throughput-Oriented Workload: TPC-C + Oracle

This workload is typical of back-end database servers. The major concern is not response time, but maintaining high throughput. Individual transactions can take significant time, as they typically perform disk I/O in the server. In *Buffer*, this workload exercises both disk and network PDDs.

Figure 9 shows the average TPC-C throughput with different schemes and checkpoint intervals normalized to that in *NoRollback*[2]. The throughput of our moderately tuned *NoRollback* setup is 1561 transactions per minute and its average response time is 612 ms. The figure shows data for *Stall*, and for *Buffer* with disk PDD only and with disk plus network PDDs. To minimize errors, we report conditions of the first 10-minute interval after the database is warmed up and the throughput becomes steady. During that period, we take a measurement every 30 seconds. The figure shows the mean and standard deviation of such measurements.

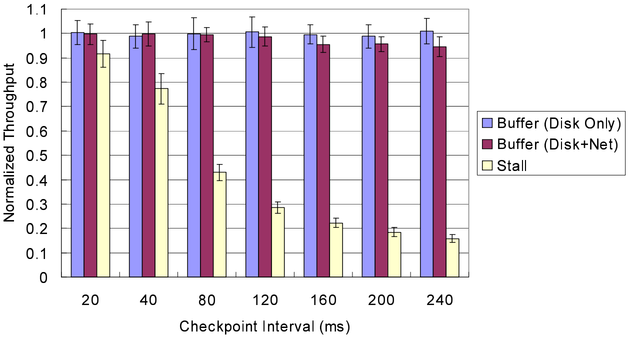


Figure 9. Transaction throughput with different schemes and checkpoint intervals normalized to *NoRollback*. The experiments run with 30 remote clients.

Consider *Stall* first. If we can only tolerate a 5% reduction in throughput, none of the checkpoint intervals shown is acceptable. In contrast, *Buffer* keeps the throughput reduction within 1% up to 120 ms checkpoint intervals. Note that for checkpoint intervals above 120 ms, the throughput reduction comes mainly from network PDD overhead. Interestingly, the average response time does not degrade as we increase the checkpoint interval. In fact, it goes down slightly, decreasing to 590 ms by the time we use 240 ms checkpoint interval. The reason is that long intervals reduce the transaction rate, which in turn diminishes disk contention.

We have repeated the *Buffer* experiments for different numbers of clients and obtained similar results. Figure 10 shows the normalized throughput for a range of checkpoint intervals for 60 and 90 clients. Each bar is normalized to *NoRollback* for the same number of clients. As we increase the number of clients (i.e., more transactions overlap in time), the delay incurred by the network PDD is less visible and, therefore, long checkpoint intervals become more tolerable.

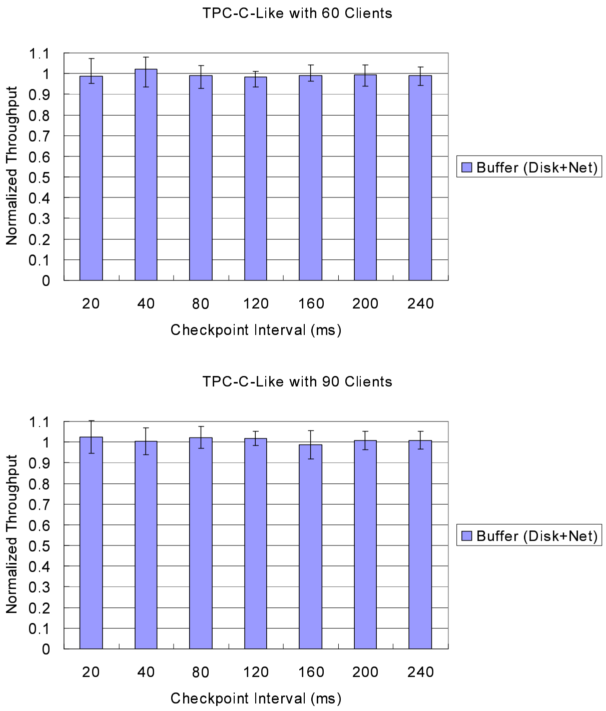


Figure 10. Normalized transaction throughput as a function of the checkpoint interval for different numbers of clients.

Overall, we conclude that in throughput-oriented workloads like TPC-C, and for 20–120 ms checkpoint intervals, our proposed *Buffer* scheme induces very small throughput reductions of up to 1%. In such workloads, the progress of transaction processing typically depends on the rate of synchronous writes issued by the database log-writer process. Our *Buffer* scheme affects such progress minimally.

6.1.4. Latency-Bound Workload: WebStone + Apache

In this workload, multiple remote clients read HTML documents that are memory-resident in the server. Each transaction involves establishing the connection with a three-way handshake, reading a file, and closing the connection. Transactions are short because there is no disk I/O — only the network PDD is exercised. This workload simulates an interactive environment. Consequently, we are interested in response time, measured as the time between requesting the connection until the whole file is received.

Figure 11 shows the response time for different numbers of clients and checkpoint intervals. The figure is organized in numbers of clients. In each group, there are bars for *NoRollback*, and for *Buffer* with different checkpoint intervals. Since there is no disk I/O, the *Stall* scheme is irrelevant. To obtain the data, we run each experiment for 10 minutes, with the server at 100% CPU usage.

We see that the response time quickly increases with the checkpoint interval. With a checkpoint interval T, the response time should increase by $2 \times T$, since we add T to establish the connection and T to get the data (Recall from Section 3.3.1 that the PDD smooths out outgoing messages). This is what we observe with

[2] Throughput is given in New Order transactions per minute; response time is for New Order transactions as well.

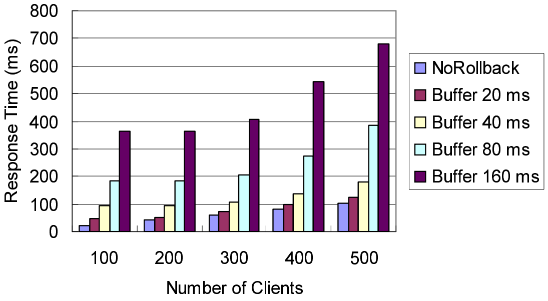


Figure 11. Response time for different numbers of clients and checkpoint intervals.

100 clients. For more than 300 clients and $T \geq 80$ms, contention causes larger increases in the response time.

According to [25], it is acceptable to add up to 100 ms to the response time of a transaction. Consequently, our *Buffer* scheme can be used in the web server measured, as long as the checkpoint interval is $\sim$50 ms or shorter.

6.2. Latency of Fault Recovery

To recover from an MR fault, our schemes perform three actions (Section 3.3.2): reset the devices, re-initialize the DDs and, in the background, perform all the buffered output operations to bring the I/O state to the checkpoint immediately preceding the fault. While our prototype server lacks ReVive hardware, we can measure the recovery latency of ReViveI/O as discussed in Section 4.

We have measured the latency of each part of the recovery for *Buffer*, and listed average values in Table 3. From the table, we see that device reset and DD re-initialization are quick. Note that, to re-initialize the disk DD, we do not need to access the disk to get information such as the number of cylinders and the sector size; these parameters are obtained from the recovered memory. Finally, the third operation takes tens of ms, but it is executed in the background. Overall, compared to the ReVive recovery latency (Figure 5), ReViveI/O adds negligible recovery overhead.

Operation	Duration	
	Disk	Network
Reset device	1 ms	15 μs
Re-initialize device driver	10 ms	60 μs
Re-issue operations in background	$\sim T$	$\sim T$

Table 3. Latencies of the operations needed to recover from an MR fault. T is the checkpoint interval.

If the fault is NMR, *Buffer* only needs to re-issue the buffered output operations. This activity takes several seconds, as *Buffer* has to read data and mappings from the disk buffer. Such latency is negligible compared to the tens of minutes needed to reboot the server and run a database recovery routine after this type of fault.

6.3. Combining ReVive and ReViveI/O: Performance Overheads and Availability

We would like to estimate the impact of combining ReViveI/O and ReVive. In [29], ReVive was evaluated for a checkpoint interval $T = 100$ ms, where each checkpoint took 1 ms. ReVive induced a 6% execution overhead.

We model ReVive as inducing a $c = 1$ ms overhead every checkpoint, and a fixed $r = 5\%$ overhead for the period between checkpoints, independently of T. Therefore, the throughput reduction factor induced by ReVive is $f = \frac{c+(T-c)\times r}{T}$. The response time increase due to ReVive is $t_s \times \frac{f}{1-f}$, where t_s is the time that the transaction spends executing in the server.

Figure 12 takes the impact of *Buffer* on TPC-C throughput (Section 6.1.3) and WebStone response time (Section 6.1.4), and adds the estimated effect of ReVive. We can see that a throughput-oriented workload such as TPC-C keeps the throughput reduction at 7% or below for checkpoint intervals between 60 and 120 ms. Most of the overhead is due to ReVive. On the other hand, the response time increase in a latency-bound workload such as WebStone is practically all due to ReViveI/O. The increase is $2 \times T$, where T is the checkpoint interval (Section 6.1.4).

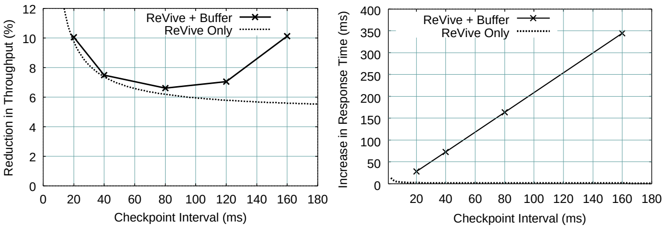


Figure 12. Estimated combined effect of ReVive and ReViveI/O on the throughput of TPC-C (left) and the response time of WebStone (right).

To complete the picture, we compare the availability of *ReVive+Buffer* and *NoRollback*. For the former, we assume that MR and NMR faults are independent and distributed exponentially. As a result, the availability of *ReVive+Buffer* is $1 - \frac{MTTR_{MR}}{MTBF_{MR}} - \frac{MTTR_{NMR}}{MTBF_{NMR}}$. We estimate $MTTR_{NMR}$ as 5 minutes for machine reboot plus 5 minutes for database recovery [22]. We set $MTTR_{MR}$ to 1 second [29]. For *NoRollback*, all faults have the same MTTR, namely $MTTR_{NMR}$.

Figure 13 shows the unavailability as a function of $MTBF_{MR}$. Note that both axes in the figure are logarithmic. For *ReVive+Buffer*, we show two curves: 1:100 assumes that $MTBF_{NMR} = 100 \times MTBF_{MR}$, while 1:1000 assumes that $MTBF_{NMR} = 1000 \times MTBF_{MR}$. For *NoRollback*, both curves are practically the same, and we show only one.

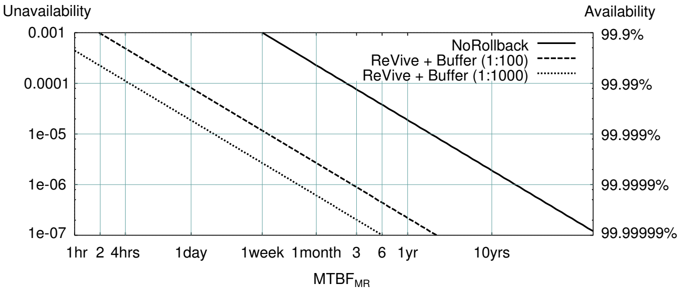


Figure 13. Unavailability as a function of the MTBF of MR faults. The MTBF of NMR faults is set to 100 or 1000 times the MTBF of MR faults (curves 1:100 and 1:1000, respectively).

The figure shows that *ReVive+Buffer* has much lower unavailability than *NoRollback* thanks to its tiny recovery latency for the more common MR faults. For example, for 1-week $MTBF_{MR}$, *ReVive+Buffer* (1:100) has an unavailability of $\sim 10^{-5}$, which corresponds to 99.999% availability, while *NoRollback* has an unavailability of 0.001, which corresponds to 99.9% availability. Overall, *ReVive+Buffer*'s unavailability is 86 and 375 times lower than *NoRollback*'s for 1:100 and 1:1000, respectively.

In summary, *ReVive+Buffer* provides higher availability than conventional systems while delivering slightly lower throughput. We believe that, at least in the applications considered, reducing downtime is much more important than achieving peak throughput while the machine is up.

7. Related Work

Masubuchi *et al.* [21] proposed adding a PDD to the kernel to support disk I/O recovery. Their scheme corresponds to *Stall* in Section 5, which blocks disk writes until the next checkpoint [20]. ReViveI/O differs as follows: (i) instead of blocking, it buffers the data and commits them later in the background, and (ii) it supports network I/O. More importantly, this paper contributed with the full implementation, testing, and evaluation of an efficient I/O undo/redo prototype compatible with solutions such as ReVive [29] or SafetyNet [34].

High-availability machines such as HP's Nonstop Architecture [11] and IBM's S/390 mainframes [33] attain fault tolerance through expensive hardware support, often involving extensive component replication. We seek a less expensive design point.

Sequoia [3] is an SMP where mirrored main memory is considered to be the checkpointed state. When a dirty line is evicted from a cache, a checkpoint is triggered. A checkpoint is also forced every time a processor requests I/O, which is an inefficient approach.

OS- and library-based checkpointing includes UNICOS [18], KeyKOS [19], diskless checkpointing [28], and fault-tolerant Mach [32]. Their checkpoint interval is typically minutes or longer. This results in high MTTR and would induce intolerably long message delays. Some schemes provide disk I/O recovery by journaling, but none addresses network I/O recovery.

Database management systems (DBMS) have their own well-known I/O recovery mechanisms [8]. Checkpoint intervals are in the order of minutes to achieve low enough overheads, and the recovery process takes tens of minutes [22] or more. For example, for System B of [22], targeting a 5-minute database recovery time, incurs 8.2% overhead in throughput. Note that such database recovery time does not include the system recovery time (e.g., repair and reboot), which at least adds minutes. Similarly, targeting a 20-minute database recovery time incurs 6.6% overhead. Also, the fault results in the loss of ongoing transactions. In contrast, for the frequent faults (MR faults), ReVive plus ReViveI/O has three advantages: (i) ongoing transactions are not lost, (ii) the system recovers in less than 1 second, and (iii) applications and kernel need no modification. When a DBMS runs on ReVive with our I/O undo/redo layer, only the infrequent faults (NMR faults) trigger database recovery. Consequently, the DBMS can use lower-overhead checkpointing, at the expense of longer recovery time.

There are cluster options for databases such as Oracle RAC (Real Application Clusters) [26]. These solutions use multiple machines to provide higher system availability. However, to achieve good performance, they require major changes to the database management system and, possibly, to the applications that run on top of it.

Our mechanisms overlap with ideas from other works. Hu and Yang [13] proposed the Disk Caching Disk (DCD), mainly to boost the performance of random disk writes; our disk buffer area is like a DCD for reliability. Several systems, such as the Legato Prestoserve [23] and Baker *et al.*'s scheme [2] have used NVRAM to speed up disk writes; we use NVRAM to speed up writes within checkpoints. Our PDD can be thought of as a thin Virtual Machine (VM) for I/O; VMs have been used for forward error recovery [5] and for post-intrusion replay [6], rather than for rollback recovery of faults.

ReViveI/O is also related to transactional memory systems (TMS) [10]. TMS share similar I/O issues with hardware-based memory checkpointing systems: when a transaction is aborted, TMS need to undo any work done by that transaction, including I/O. On the other hand, there are some differences: (i) TMS do not need to redo I/O after a transaction is aborted, (ii) TMS have less emphasis on durability than memory checkpointing systems because their primary purpose is to support atomicity, not reliability, and (iii) TMS may additionally need some mechanism to detect and track the dependences and conflicts between threads that are induced by I/O operations.

8. Conclusion

The main contribution of this paper is the full implementation, testing, and experimental evaluation of *ReViveI/O*, an efficient scheme for I/O undo/redo that is compatible with high-frequency checkpointing architectures such as ReVive and SafetyNet. In addition, we perform a sensitivity analysis of what checkpoint frequencies are required to maintain acceptable throughput and response times. Overall, this work completes the viability assessment of such novel memory-recovery architectures.

Our ReViveI/O prototype shows that low-overhead, tiny-MTTR recovery of I/O is feasible. For 20–120 ms between checkpoints, the throughput of a throughput-oriented workload such as TPC-C on Oracle decreases by no more than 1%. Moreover, for 50 ms between checkpoints or less, the response time of a latency-bound workload such as WebStone on Apache remains tolerable. In all cases, the recovery time is practically negligible. Moreover, kernel, device drivers, and applications remain unmodified. Finally, the combination of ReVive and ReViveI/O is likely to reduce the throughput of TPC-C-class applications by 7% or less for 60–120 ms checkpoint intervals, while incurring a tiny MTTR of less than 1 second.

Our analysis suggests that support for ReVive plus ReViveI/O makes for a very cost-effective high-availability server. In general, compared to software-based recovery solutions (OS, library, and database), our approach has a higher error coverage, lower overhead, and much smaller MTTR. We estimate that it delivers a 2–3 orders of magnitude reduction in unavailability over a database server that we evaluated. Finally, compared to hardware-intensive solutions such as HP NonStop systems, it is much cheaper while maintaining high availability.

An avenue for future work is to apply our techniques to transactional memory systems. We expect that ReViveI/O can be applied to transactional memory systems by adding support for per-thread output commit and for some dependence/conflict tracking mechanism through the Memory Buffer.

Acknowledgments

The authors would like to thank the members of HP's former Western Research Laboratory and of the I-ACOMA group at the University of Illinois for their helpful comments.

References

[1] Apache. http://www.apache.org/.

[2] M. Baker et al. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System*, pages 10–22, 1992.

[3] P. Bernstein. Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing. *IEEE Computer*, 21(2):37–45, 1988.

[4] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2002.

[5] T. Bressoud and F. Schneider. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 1–11, 1995.

[6] G. Dunlap et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, 2002.

[7] E. Elnozahy et al. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[9] W. Gu et al. Characterization of Linux kernel behavior under errors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 459–468, 2003.

[10] L. Hammond et al. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, 2004.

[11] HP Integrity NonStop computing. http://www.hp.com/go/integritynonstop.

[12] HP Integrity servers. http://www.hp.com/products1/servers/integrity/.

[13] Y. Hu and Q. Yang. DCD – Disk Caching Disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 169–178, 1996.

[14] IBM System p5 servers. http://www.ibm.com/systems/p/.

[15] IBM Advanced SerialRAID Adapters Technical Reference (SA33-3286-02), September 2000.

[16] Iperf Benchmark. http://dast.nlanr.net/Projects/Iperf/.

[17] D. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 86–95, 1993.

[18] B. Kingsbury and J. Kline. Job and process recovery in a UNIX-based operating system. In *Proceedings of the Usenix Winter 1989 Technical Conference*, pages 355–364, 1989.

[19] C. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91, 1992.

[20] Y. Masubuchi. Personal communication. Oct. 2003.

[21] Y. Masubuchi et al. Fault recovery mechanism for multiprocessor servers. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 184–193, 1997.

[22] J. Mauro. Tuning Oracle to minimize recovery time. Technical Report 817-4445-10, Sun Microsystems, November 2003.

[23] J. Moran et al. Breaking through the NFS performance barrier. In *Proceedings of the European Unix Users Group Spring*, pages 199–206, 1990.

[24] C. Morin et al. COMA: an opportunity for building fault-tolerant scalable shared memory multiprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 56–65, 1996.

[25] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[26] Oracle Real Application Clusters 10g. Oracle Technical White Paper, May 2005.

[27] D. Patterson et al. Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, UC Berkeley, March 2002.

[28] J. Plank, K. Li, and M. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[29] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, 2002.

[30] A. Reuter. A fast transaction-oriented logging scheme for UNDO recovery. *IEEE Transactions on Software Engineering*, SE-6(4):348–356, 1980.

[31] M. Rosenblum and K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[32] M. Russinovich and Z. Segall. Fault-tolerance for off-the-shelf applications and hardware. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 67–71, 1995.

[33] T. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[34] D. Sorin et al. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, 2002.

[35] Sun Microsystems. *Sun StorEdge Fast Write Cache 2.0 Configuration Guide*, February 2000.

[36] A. Tanenbaum. *Computer Networks*. Prentice Hall, 2003.

[37] Transaction Processing Performance Council. Top ten non-clustered TPC-C by performance (as of November 2005). http://www.tpc.org/.

[38] WebStone Benchmark. http://www.mindcraft.com/webstone/.

Appendix A: Summarized Operation of ReVive

During fault-free execution, all processors are periodically interrupted to establish a global checkpoint. Establishing a checkpoint involves writing back register and modified cache state to memory. As a result, main memory *is* the checkpointed state at that point. Between checkpoints, the program may modify the memory contents. However, when a line of checkpoint data in main memory is about to be overwritten by a cache eviction, the memory controller saves the line in a log. Later, after the next checkpoint is established, the logs are discarded. At any time, if a fault is detected, the logs are used to restore the memory state to that of the last checkpoint.

To enable recovery from faults that result in lost memory content, ReVive organizes pages from different nodes into parity groups. Each main memory write is intercepted by the memory controller in the node, triggering an update of the corresponding parity bits located in a page on another node. The parity information is used when the system detects a fault in the memory of one node (e.g., permanent loss of a node's memory). Then, the parity bits and data from the remaining nodes' memories are used to reconstruct the lost memory content (both logs and program state). More details are found in [29].