RemusDB: Transparent High Availability for Database Systems

Umar Farooq Minhas[†] Shriram Rajagopalan[‡] Brendan Cully[‡]
Ashraf Aboulnaga[†] Kenneth Salem[†] Andrew Warfield[‡]
[†]University of Waterloo, Canada
{ufminhas, ashraf, kmsalem}@cs.uwaterloo.ca
[‡]University of British Columbia, Canada
{rshriram, brendan, andy}@cs.ubc.ca

ABSTRACT

In this paper we present a technique for building a high-availability (HA) database management system (DBMS). The proposed technique can be applied to any DBMS with little or no customization, and with reasonable performance overhead. Our approach is based on Remus, a commodity HA solution implemented in the virtualization layer, that uses asynchronous virtual machine (VM) state replication to provide transparent HA and failover capabilities. We show that while Remus and similar systems can protect a DBMS, database workloads incur a performance overhead of up to 32% as compared to an unprotected DBMS. We identify the sources of this overhead and develop optimizations that mitigate the problems. We present an experimental evaluation using two popular database systems and industry standard benchmarks showing that for certain workloads, our optimized approach provides very fast failover ($\leq$ 3 seconds of downtime) with low performance overhead when compared to an unprotected DBMS. Our approach provides a practical means for existing, deployed database systems to be made more reliable with a minimum of risk, cost, and effort. Furthermore, this paper invites new discussion about whether the complexity of HA is best implemented within the DBMS, or as a service by the infrastructure below it.

1. INTRODUCTION

Maintaining availability in the face of hardware failures is an important goal for any database management system (DBMS). Users have come to expect 24×7 availability even for simple non-critical applications, and businesses can suffer costly and embarrassing disruptions when hardware fails. Many database systems are designed to continue serving user requests with little or no disruption even when hardware fails. However, this *high availability (HA)* comes at a high cost in terms of complex code in the DBMS, complex setup for the database administrator, and sometimes extra specialized hardware. In this paper, we present a reliable, cost-effective HA solution that is transparent to the DBMS, runs on commodity hardware, and incurs a low performance overhead. A key feature of our solution is that it is based on *virtual machine (VM) replication* and leverages the capabilities of the underlying virtualization layer.

Providing HA guarantees as part of the DBMS can add a substantial amount of complexity to the DBMS implementation. For example, to integrate a simple *active-standby* approach, the DBMS has to support propagating database updates from the active to the standby (e.g., by shipping log records), coordinating transaction commits and aborts between the active and standby, and ensuring consistent atomic handover from active to standby after a failure.

In this paper, we present an active-standby HA solution that is based on running the DBMS in a virtual machine and pushing much of the complexity associated with HA out of the DBMS, relying instead on the capabilities of the *virtualization layer*. The virtualization layer captures changes in the state of the whole VM at the active host (including the DBMS) and propagates them to the standby host, where they are applied to a backup VM. The virtualization layer also detects failure and manages the *failover* from the active host to the standby, transparent to the DBMS. During failover, all transactional (ACID) properties are maintained and client connections are preserved, making the failure transparent to the DBMS clients.

Database systems are increasingly being run in virtual machines for easy deployment (e.g., in cloud computing environments [1]), flexible resource provisioning [27], better utilization of server resources, and simpler administration. A DBMS running in a VM can take advantage of different services and capabilities provided by the virtualization infrastructure such as live migration, elastic scaleout, and better sharing of physical resources. These services and capabilities expand the set of features that a DBMS can offer to its users while at the same time simplifying the implementation of these features. Our view in this paper is that adding HA to the set of services provided by the virtualization infrastructure continues down this road: any DBMS running on a virtualized infrastructure can use our solution to offer HA to its users with little or no changes to the DBMS code for either the client or the server. Our design decisions ensure that the setup effort and performance overhead for this HA is minimal.

The idea of providing HA by replicating machine state at the virtualization layer is not new [5], and our system is based on *Remus* [8], a VM checkpointing system that is already part of the Xen hypervisor [4]. Remus targets commodity HA installations and transparently provides strong availability guarantees and seamless failure recovery. However, the general VM replication used by systems such as Remus imposes a significant performance overhead on database systems. In this paper, we develop ways to reduce this overhead and implement them in a *DBMS-aware VM checkpointing* system that we call *RemusDB*.

We identify two causes for the performance overhead experi-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.
Proceedings of the VLDB Endowment, Vol. 4, No. 11
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

enced by a database system under Remus and similar VM checkpointing systems. First, database systems use memory intensively, so the amount of state that needs to be transferred from the primary VM to the backup VM during a checkpoint is large. Second, database workloads can be sensitive to network latency, and the mechanisms used to ensure that client-server communication can survive a failure add latency to the communication path. RemusDB implements techniques that are completely transparent to the DBMS to reduce the amount of state transferred during a checkpoint (Section 3). To reduce the latency added to the client-server communication path, RemusDB provides facilities that are not transparent to the DBMS, but rather require minor modifications to the DBMS code (Section 4). We use RemusDB to add high availability to PostgreSQL and MySQL, and we experimentally demonstrate that it effectively recovers from failures and imposes low overhead on normal operation (Section 5). For example, as compared to Remus, RemusDB achieves a performance improvement of 29% and 30% for TPC-C workload running under PostgreSQL and MySQL, respectively. It is also able to recover from a failure in ≤ 3 seconds while incurring 3% performance overhead with respect to an unprotected VM.

2. BACKGROUND & SYSTEM OVERVIEW

In our setup, shown in Figure 1, two servers are used to provide HA for a DBMS. One server hosts the *active* VM, which handles all client requests during normal operation. As the active VM runs, its entire state including memory, disk, and active network connections are continuously checkpointed to a *standby* VM on a second physical server. Our objective is to tolerate a failure of the server hosting the active VM by *failing over* to the DBMS in the standby VM, while preserving full ACID transactional guarantees. In particular, the effects of transactions that commit (at the active VM) before the failure should persist (at the standby VM) after the failover, and failover should not compromise transaction atomicity.

During normal operation, Remus takes frequent, incremental checkpoints of the complete state of the virtual machine on the active server. These checkpoints are shipped to the standby server and "installed" in the virtual machine there. The checkpoints also act as heartbeat messages from the active server (Server 1) to the standby server (Server 2). If the standby times out while waiting for a checkpoint, it assumes that the active server has failed. This causes a failover, and the standby VM begins execution from the most recent checkpoint that was completed prior to the failure. This failover is completely transparent to clients. The standby VM has the same IP address as the active VM, and the standby server's hypervisor ensures that network packets going to the (dead) active VM are automatically routed to the (live) standby VM after the failure, as in live VM migration [7]. In checkpoint-based whole-machine protection systems like Remus, the virtual machine on the standby server does *not* mirror the execution at the active server during normal operation. Rather, the activity at the standby server is limited to installation of incremental checkpoints from the active server, which reduces the resource consumption at the standby.

Remus's checkpoints capture the entire state of the active VM, which includes disk, memory, CPU, and network device state. Thus, this captures both the state of the database and the internal execution state of the DBMS, e.g., the contents of the buffer pool, lock tables, and client connection state. After failover, the DBMS in the standby VM begins execution with a completely warmed up buffer pool, picking up exactly where the active VM was as of the most recent checkpoint, with all session state, TCP state, and transaction state intact. This fast failover to a warm backup and no loss of client connections is an important advantage of our approach.

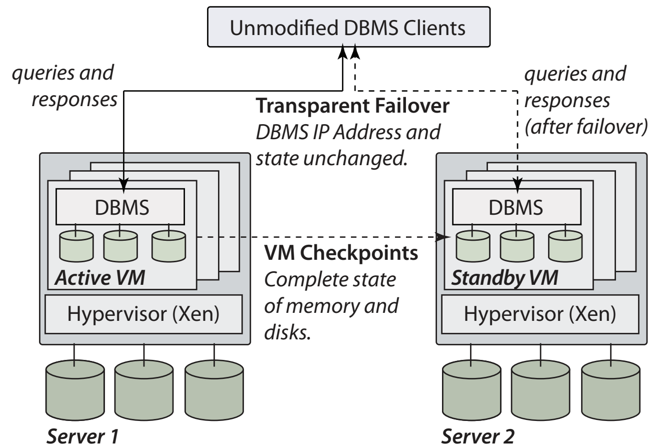

Figure 1: RemusDB System Architecture.

Some DBMS-level HA solutions provide similar features, but these features add more code and complexity to the already complex systems. With our approach, these features are essentially free.

Figure 2 shows a simplified timeline illustrating checkpoints and failover. In reality, checkpoint transmission and acknowledgement is carefully overlapped with execution to increase performance while maintaining consistency [8]. However, the simplified timeline shown in Figure 2 is sufficient to illustrate the important features of this approach to DBMS high availability. When the failure occurs in Figure 2, all of the work accomplished by the active server during epoch C is lost. If, for example, the active server had committed a database transaction T during epoch C, any trace of that commit decision will be destroyed by the failure. Effectively, the execution of the active server during each interval is *speculative* until the interval has been checkpointed, since it will be lost if a failure occurs. Remus controls *output commit* [28] to ensure that the external world (e.g., the DBMS clients) sees a consistent view of the server's execution, despite failovers. Specifically, Remus queues and holds any outgoing network packets generated by the active server until the completion of the next checkpoint. For example, outgoing packets generated by the active server during epoch B in Figure 2 will be held by Remus until the completion of the checkpoint at the end of B, at which point they will be released. Similarly, a commit acknowledgement for transaction T, generated during epoch C, will be held by Remus and will be lost when the failure occurs. This *network buffering* ensures that no client will have been able to observe the speculative commit of T and conclude (prematurely or incorrectly) that T is durably committed. The output commit principle is also applied to the disk writes generated at the active server during an epoch. At the standby server, Remus buffers the writes received from active server during epoch B and releases them to its disk only at the end of the epoch. In the case of failure during epoch C, Remus discards the buffered writes of this epoch, thus maintaining the overall consistency of the system.

For a DBMS, the size of a Remus checkpoint may be large, which increases checkpointing overhead. Additionally, network buffering introduces message latency which may have a significant effect on the performance of some database workloads. RemusDB extends Remus with optimizations for reducing checkpoint size and for reducing the latency added by network buffering. We present these optimizations in the next two sections.

3. MEMORY OPTIMIZATIONS

Remus takes a deliberately simple approach to memory checkpointing: at every checkpoint, it copies all the pages of memory

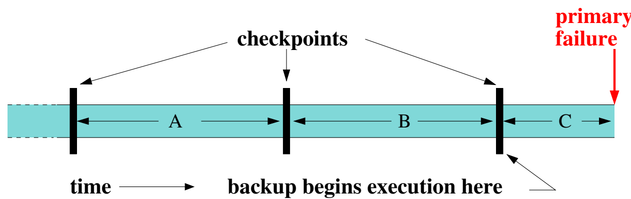

Figure 2: A Primary Server Execution Timeline.

that change from the active host and transmits them over the network to the backup host. The authors of Remus argue that this simplicity is desirable: it provides high availability with an acceptable degree of overhead, with an implementation that is simple enough that one can have confidence in its correctness, regardless of the target application or hardware architecture. This is in stark contrast to the complexity of previous systems, even those implemented in the hypervisor [5]. And while this argument for simplicity holds for database systems, the overhead penalty is higher: database workloads tend to modify more memory in each checkpoint epoch than other workloads. This section describes a set of optimizations designed to reduce this overhead.

3.1 Sending Less Data

Compressing checkpoints is beneficial when the amount of data to be replicated is large, and the data contains redundancy. Our analysis found that both of these conditions apply to database workloads: (a) they involve a large set of frequently changing pages of memory (most notably buffer pool pages), and (b) the memory writes often change only a small part of the pages on which they occur. This presents an opportunity to achieve a considerable reduction in replication traffic by only sending the actual changes to these pages.

To achieve this, we implemented an LRU-based cache of frequently changing pages from previous checkpoints. This cache is maintained in *domain 0*, the privileged VM used for control by the Xen hypervisor. Our experimentation showed that a cache size of 10% of VM memory offers the desired performance improvement while maintaining an acceptable memory footprint in domain 0. When sending pages to the backup, we first check to see if the previous version of the page exists in this cache. If it does, the contents of the two pages are XORed, usually resulting in a page that contains mostly zeros, reflecting the large amount of identical data. The result is then run-length encoded for transmission. If the page is not found in the cache, it is sent uncompressed, and is added to the cache using the standard LRU eviction policy.

The original Remus work maintained that asynchronous, pipelined checkpoint processing while the active VM continues to execute is critical to minimizing its performance impact. The benefits of this approach were evident in implementing checkpoint compression: moving the implementation into an asynchronous stage and allowing the VM to resume execution in parallel with compression and replication in domain 0 halved the overhead of RemusDB.

3.2 Protecting Less Memory

Compressed checkpoints help considerably, but the work involved in taking and sending checkpoints is still proportional to the amount of memory changed between checkpoints. In this section, we discuss ways to reduce checkpoint size by selectively *ignoring* changes to certain parts of memory. Specifically, a significant fraction of the memory used by a DBMS goes into the buffer pool. Clean pages in the buffer pool do not need to be sent in Remus checkpoints if they can be regenerated by reading them from the disk. Even dirty buffer pool pages can be omitted from Remus checkpoints if the DBMS can recover changes to these pages from the transaction log.

In addition to the buffer pool, a DBMS uses memory for other purposes such as lock tables, query plan cache, working memory for query operators, and connection state. In general, memory pages whose contents can be regenerated, or alternatively can be safely thrown away may be ignored during checkpointing. Based on these observations, we developed two checkpointing optimizations: *disk read tracking* and *memory deprotection*. Disk read tracking is presented below. Details of memory deprotection can be found in Appendix A.

Disk Read Tracking

Remus, like the live VM migration system on which it is based [7], uses hardware page protection to track changes to memory. As in a copy-on-write process fork, all of the page table entries of a protected virtual machine are set to read only, producing a trap when any page is modified. The trap handler verifies that the write is allowed, then updates a bitmap of "dirty" pages, which determines the set of pages to transmit to the backup server at each checkpoint. This bitmap is cleared after the checkpoint is taken.

Because Remus keeps a synchronized copy of the disk on the backup, any pages that have been read from disk into memory may be safely excluded from the set of dirty pages, as long as the memory has not been modified after the page was read from disk. Our implementation interposes on disk read requests from the virtual machine and tracks the set of memory pages into which the reads will be placed, and the associated disk addresses from which those pages were read. Normally, the act of reading data from disk into a memory page would result in that page being marked as dirty and included in the data to be copied for the checkpoint. Our implementation does *not* mark that page dirty, and instead adds an annotation to the replication stream indicating the sectors on disk that may be read to reconstruct the page remotely.

Normally, writes to a disk pass through the operating system's (or DBMS's) buffer cache, and this will inform Remus to invalidate the read-tracked version of the page and add it to the set of pages to transmit in the next checkpoint. However, it is possible that the contents of the sectors on disk that a read-tracked page refers to may be changed without touching the in-memory read-tracked page. For example, a process different from the DBMS process can perform a direct (unbuffered) write to the file from which the read-tracked page is to be read after failure. In this case, read tracking would incorrectly recover the newer version of the page on failover. Although none of the database systems that we studied exhibited this problem, protecting against it is a matter of correctness, so RemusDB maintains a set of backpointers from read-tracked pages to the associated sectors on disk. If the VM writes to any of these sectors, we remove the page from the read tracking list and send its contents normally.

4. COMMIT PROTECTION

Irrespective of memory optimizations, the single largest source of overhead for many database workloads on the unmodified Remus implementation was the delay introduced by buffering network packets for controlling output commit. Client/server interactions in DBMS environments typically involve long-lived sessions with frequent interactions over low-latency local area networks. For example, a TPC-C transaction on PostgreSQL in our experiments has an average of 32 packet exchanges between client and server, and a maximum of 77 packet exchanges. Remus's network buffering delays all these packets; packets that might otherwise have round trip

times on the order of hundreds of microseconds are held until the next checkpoint is complete, potentially introducing two to three orders of magnitude in latency per round trip.

Output commit for all network packets is unnecessarily conservative for database systems since they have their own notion of transactions with clear consistency and durability semantics. This makes Remus's TCP-level per-checkpoint transactions redundant. To let the DBMS supersede the Remus network transaction protocol, we relax output commit by allowing the DBMS to decide which packets sent to the client should be protected (i.e., buffered until the next checkpoint) and which packets can be unprotected (i.e., sent unbuffered). The DBMS should be able to protect transaction control messages, i.e., acknowledgements to `COMMIT` and `ABORT` requests, and server-generated `ABORT` messages. The messages comprising the contents of the transaction can remain unprotected. We call this approach *commit protection*.

To implement commit protection, the communication channel between the database server and the client (e.g., stream sockets) needs to provide abstractions to dynamically enable or disable buffering of network messages, i.e., abstractions to *protect* or *deprotect* the channel. Given these abstractions, the following protocol can be used to implement commit protection inside a DBMS:

1. If the message received from the client is either `COMMIT` or `ABORT`, or if the server decides to abort an active transaction, switch the client/server communication channel to protected mode.
2. Perform all actions necessary to commit or abort the transaction, up to writing the `COMMIT` or `ABORT` log record to disk.
3. After successfully committing or aborting the transaction, send the corresponding acknowledgement to the client through the (currently protected) channel.
4. Switch the channel back to deprotected mode.

This protocol ensures that committed (aborted) transactions exposed by the primary VM are guaranteed to survive a failure: the client only receives the acknowledgement of the commit or abort after state changes made by this transaction at the primary VM have propagated to the backup VM as part of a Remus checkpoint.

To recover from a failure of the active host, we rely on a post-failover recovery handler that runs inside the DBMS at the backup VM after it takes over from the primary VM after a failure. The recovery handler cleanly aborts all in-flight transactions that are active on any client/server communication channel that was in deprotected mode before the failure. This is necessary because, due to deprotection, the client may have been exposed to speculative state that is lost after a failover, i.e., rolled back to the last Remus checkpoint. While some active transactions may be aborted during failover, no transactions that have been acknowledged as committed will be lost.

To provide a DBMS with the ability to dynamically switch a client connection between protected and deprotected modes, we added a new `setsockopt()` option to Linux. Our implementation guarantees that protected packets will be buffered until the next Remus checkpoint is reached. One outstanding issue with commit protection is that while it preserves complete application semantics, it exposes TCP connection state that can be lost on failover: unbuffered packets advance TCP sequence counters that cannot be reversed, which can result in the connection stalling until it times out. In the current implementation of RemusDB we have not addressed this problem: only a small subset of connections are affected, and the transactions occurring over them will be recovered when the connection times out just like any other timed out client

	Virtualization Layer	Guest VM Kernel	DBMS
Commit Protection	13	396	103(PostgreSQL), 85(MySQL)
Disk Read Tracking	1903	0	0
Compression	593	0	0

Table 1: RemusDB Source Code Modifications (Lines of Code).

connection. In the future, we plan to explore techniques by which we can track sufficient state to explicitly close TCP connections that have become inconsistent at failover time, in order to speed up transaction recovery time for those sessions.

We have implemented commit protection in PostgreSQL and MySQL, with minor modifications to the client connection layer. Because the changes required are for a small and well-defined part of the client/server protocol, we expect them to be easily applied to any DBMS. Table 1 provides a summary of the source code changes made to different subsystems to implement the different optimizations that make up RemusDB.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Environment

Our experimental setup consists of two servers each equipped with two quad-core Intel Xeon processors, 16GB RAM, and two 500GB SATA disks. We use the Xen 4.0 hypervisor (64-bit), Debian 5.0 (32-bit) as the host operating system, and Ubuntu 8.04 (32-bit) as the guest operating system. XenLinux Kernel 2.6.18.8 is used for both host and guest operating systems, with disks formatted using the *ext3* filesystem.

We evaluate RemusDB with PostgreSQL 8.4.0 (referred to as Postgres) and MySQL 5.0, against representative workloads, namely TPC-C [31] (OLTP) and TPC-H [30] (DSS). For interested readers, Appendix B contains additional results with a TPC-W [32] (e-commerce) workload. To run TPC-C with Postgres and MySQL we use the TPCC-UVa [19] and Percona [24] benchmark kits, respectively. We modified the TPCC-UVa benchmark kit so that it uses one connection per client; the original benchmark kit uses one shared connection for all clients. Our TPC-H experiments on Postgres consist of a *warmup* run with all 22 read-only TPC-H queries executed serially, followed by one *power stream* query sequence. We do not perform TPC-H throughput tests or use the refresh streams. The benchmark clients are always run from a separate physical machine.

Table 2 provides a summary of our experimental settings including the Remus checkpointing interval (CPI). We use the following abbreviations to refer to different RemusDB optimizations in our experiments: RT – Disk Read Tracking, ASC – Asynchronous Checkpoint Compression, and CP – Commit Protection. We present the key results of our experimental study in this section. Further experimental results can be found in Appendix B.

5.2 Behaviour of RemusDB During Failover

In the first experiment, we show RemusDB's performance in the presence of failures of the primary host. We run the TPC-C benchmark against Postgres and MySQL and plot throughput in transactions per minute (TpmC). We only plot transactions that complete within 5 seconds as required by TPC-C specifications. We run the test for 1 hour, a failure of the primary host is simulated at 30 minutes by cutting power to it. We compare the performance of a database system protected by unoptimized Remus and by Re-

DBMS	Benchmark	Performance Metric	Default Scale	Test Duration (mins)	DB Size (GB)	BP Size (MB)	VM Memory (GB)	vCPUs	Remus CPI (ms)
Postgres	TPC-C	TpmC	20W, 200C	30	1.9	190	2	2	50
	TPC-H	Execution Time	1	–	2.3	750	1.5	2	250
	TPC-W	WIPSb	10K Items	20	1.0	256	2	2	100
MySQL	TPC-C	TpmC	30W, 300C	30	3.0	300	2	2	50

Table 2: Experimental Settings.

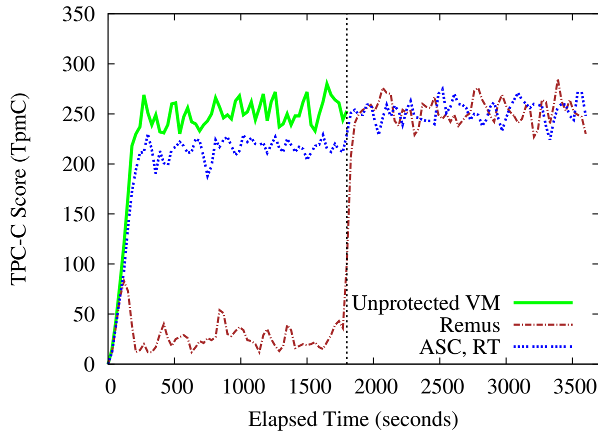

Figure 3: TPC-C Failover (Postgres).

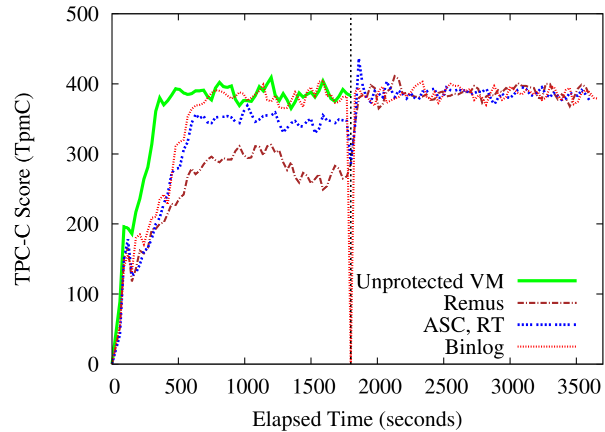

Figure 4: TPC-C Failover (MySQL).

musDB with its two transparent optimizations (ASC, RT) in Figures 3 and 4. The performance of an unprotected database system (without HA) is also shown for reference. The throughput shown in the figure is the average throughput for a sliding window of 60 seconds. Note that MySQL is run with a higher scale (Table 2) than Postgres because of its ability to handle larger workloads when provided with the same resources.

Under both versions of Remus, when the failure happens at the primary physical server, the VM at the backup physical server recovers with $\leq$ 3 seconds of downtime and continues execution. The database is running with a *warmed up buffer pool, no client connections are lost, and in-flight transactions continue to execute normally* from the last checkpoint. We only lose the speculative execution state generated at the primary server since the last checkpoint. In the worst case, Remus loses one checkpoint interval's worth of work. But this loss of work is completely transparent to the client since Remus only releases external state at checkpoint boundaries. After the failure, throughput rises sharply and reaches a steady state comparable to that of the unprotected VM before the failure. This is because the VM after the failure is not protected, so we do not incur the replication overhead of Remus.

Figure 4 also shows results with MySQL's integrated replication solution, Binlog [22, Ch. 5.2.3]. The current stable release of Postgres, used in our experiments, does not provide integrated HA support, although such a facility is in development for Postgres 9. MySQL Binlog replication, in combination with monitoring systems like Heartbeat [18], provides performance very close to that of an unprotected VM and can recover from a failure with $\leq$ 5 seconds of server downtime. However, we note that RemusDB has certain advantages when compared to Binlog replication:

- *Completeness*. On failover, Binlog replication can lose up to one transaction even under the most conservative settings [22, Ch. 16.1.1.1]. In contrast, even with aggressive optimizations such as commit protection, RemusDB never loses transactions.
- *Transparency*. Client-side recovery is more complex with Binlog, which loses all existing client sessions at failure. To show Binlog performance after recovery in Figure 4, we had to modify the TPC-C client to reconnect after the failure event. This violates the TPC specification, which requires that clients not reconnect if their server context has been lost [31, 6.6.2]. Because we are comparing server overhead, we minimized the client recovery time by manually triggering reconnection immediately upon failover. In practice, DBMS clients would be likely to take much longer to recover, since they would have to time-out their connections.
- *Implementation complexity*. Binlog accounts for approximately 18K lines of code in MySQL, and is intricately tied to the rest of the DBMS implementation. Not only does this increase the effort required to develop the DBMS (as developers must be cautious of these dependencies), but it also results in constant churn for the Binlog implementation, ultimately making it more fragile. Binlog has experienced bugs proportionate to this complexity: more than 700 bugs were reported over the last 3 years.

5.3 Overhead During Normal Operation

Having established the effectiveness of RemusDB at protecting from failure and its fast failover time, we now turn our attention to the overhead of RemusDB during normal operation. This section serves two goals: (a) it quantifies the overhead imposed by unoptimized Remus on normal operation for different database benchmarks, and (b) it measures how the RemusDB optimizations affect this overhead, when applied individually or in combination. In this section, we report results for the TPC-C and TPC-H benchmarks on Postgres. Results for TPC-W and for MySQL can be found in Appendix B.

Figure 5 presents TPC-C benchmark results for Postgres. The benchmark was run for 30 minutes using the settings presented

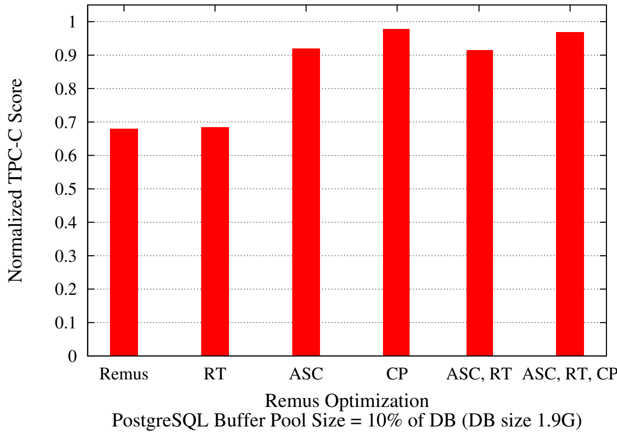

Figure 5: TPC-C Overhead(Postgres) [Base Score = 243 tpmC].

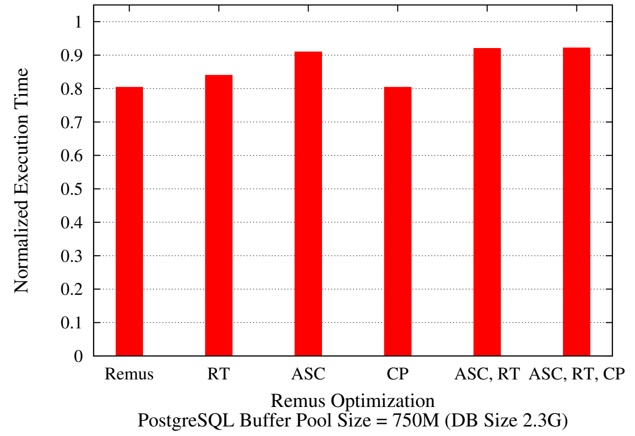

Figure 6: TPC-H Overhead (Postgres) [Base Runtime = 921s].

in Table 2. The TpmC score reported in this graph takes into account all transactions during the measurement interval irrespective of their response time (even if it violates the TPC-C specification). It is clear from the graph that without optimizations, Remus protection for database systems comes at a very high cost. The RT optimization provides very little performance benefit because TPC-C has a small working set and dirties many of the pages that it reads. However, ASC and CP provide significant performance gains (performance of 0.9-0.97 w.r.t. base). TPC-C is particularly sensitive to network latency and both of these optimizations help reduce this latency, either by reducing the time it takes to checkpoint (ASC) or by getting rid of the extra latency incurred due to Remus's network buffering for all but commit packets (CP). The combination of all three optimizations (ASC, RT, CP) yields the best performance at the risk of a few transaction aborts (not losses) and connection failures. Other experiments (not presented here) show that on average about 10% of the clients lose connectivity after failover when CP is enabled. In most cases, this is an acceptable trade-off given the high performance under (ASC, RT, CP) during normal execution. This is also better than many existing solutions where there is a possibility of losing not only connections but also committed transactions, which never happens in RemusDB.

Figure 6 presents the results for TPC-H with Postgres. Since TPC-H is a decision support benchmark that consists of long-running compute and I/O intensive queries typical of a data warehousing environment, it shows very different performance gains with different RemusDB optimizations as compared to TPC-C. In particular, as opposed to TPC-C, we see some performance gains with RT because TPC-H is a read intensive workload, and absolutely no gain with CP because it is insensitive to network latency. A combination of optimizations still provides the best performance, but in case of TPC-H most of the benefits come from memory optimizations (ASC and RT). These transparent memory optimizations bring performance to within 10% of the base case, which is a reasonable performance overhead. Using the non-transparent CP adds no benefit and is therefore not necessary. Moreover, the opportunity for further performance improvement by using the non-transparent memory deprotection interface (presented in Appendix A) is limited to 10%. Therefore, we conclude that it is not worth the additional complexity to pursue it.

RemusDB has a lot to offer for a wide variety of workloads that we study in this experiment. This experiment shows that a combination of memory and network optimizations (ASC and CP) work well for OLTP style workloads, while DSS style workloads gain

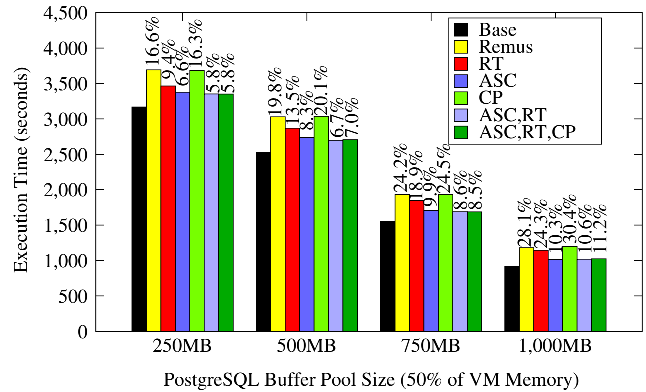

Figure 7: Effect of DB Buffer Pool Size on RemusDB (TPC-H).

the most benefit from memory optimizations alone (ASC and RT). It also shows that by using the set of optimizations that we have implemented in RemusDB, we gain back almost all of the performance lost when going from an unprotected VM to a VM protected by unoptimized Remus.

5.4 Effects of DB Buffer Pool Size

In this experiment, we study the effect of varying the size of the database buffer pool on different memory optimizations (ASC and RT). We use the TPC-H workload for this experiment since the previous experiment showed that memory optimizations offer significant performance gains for this workload.

We run a scale factor 1 TPC-H workload, varying the database buffer pool size from 250MB to 1000MB. We measure the total execution time for the warmup run and the power test run in each case, and repeat this for different RemusDB optimizations. To have reasonably realistic settings, we always configure the buffer pool to be 50% of the physical memory available to the VM. For example, for a 250MB buffer pool, we run the experiment in a 500MB VM and so on. Results are presented in Figure 7. The numbers on top of each bar show the relative overhead with respect to an unprotected VM for each buffer pool setting.

Focusing on the results with a 250MB buffer pool in Figure 7, we see a 16.6% performance loss with unoptimized Remus. Optimized RemusDB with RT and ASC alone incurs only 9.4% and 6.6% overhead, respectively. The RemusDB memory optimizations (ASC, RT) when applied together result in an overhead of

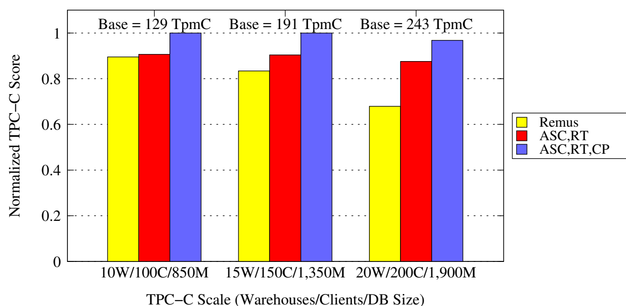

Figure 8: Effect of Database Size on RemusDB (TPC-C).

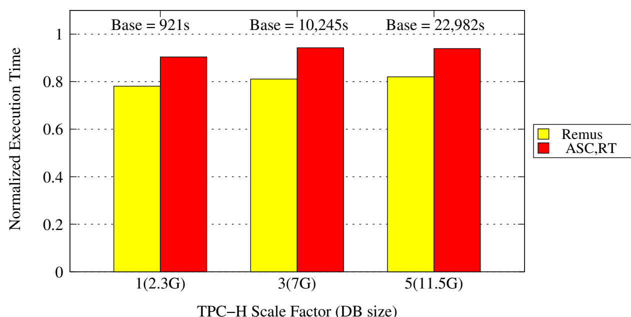

Figure 9: Effect of Database Size on RemusDB (TPC-H).

only 5.8%. As noted in the previous experiment, CP does not offer any performance benefit for TPC-H. We see the same trends across all buffer pool sizes. It can also be seen from the graph that the overhead of RemusDB increases with larger buffer pool (and VM memory) sizes. This is because the amount of work done by RemusDB to checkpoint and replicate changes to the backup VM is proportional to the amount of memory dirtied, and there is potential for dirtying more memory with larger buffer pool sizes. However, this overhead is within a reasonable 10% for all cases.

Another insight from Figure 7 is that the benefit of RT decreases with increasing buffer pool size. Since the database size is 2.3GB on disk (Table 2), with a smaller buffer pool size (250 and 500MB) only a small portion of the database fits in main memory, resulting in a lot of "paging" in the buffer pool. This high rate of paging (frequent disk reads) makes RT more useful. With larger buffer pool sizes, the paging rate decreases drastically and so does the benefit of RT, since the contents of the buffer pool become relatively static. In practice, database sizes are much larger than the buffer pool sizes, and hence a moderate paging rate is common.

For this experiment, we also measure the savings in replication network bandwidth usage achieved with different optimizations for different buffer pool sizes. Results are presented in Appendix B.

5.5 Effect of Database Size on RemusDB

In the last experiment, we want to show how RemusDB scales with different database sizes. Results for the TPC-C benchmark on Postgres with varying scales are presented in Figure 8. We use three different scales: (a) 10 warehouses, 100 clients, 850MB database; (b) 15 warehouses, 150 clients, 1350MB database; and (c) 20 warehouses, 200 clients, 1900MB database. The Postgres buffer pool size is always 10% of the database size. As the size of the database grows, the relative overhead of unoptimized Remus increases considerably, going from 10% for 10 warehouses to 32% for 20 warehouses. RemusDB with memory optimizations (ASC, RT) incurs an overhead of 9%, 10%, and 12% for 10, 15, and 20 warehouses, respectively. RemusDB with memory and network optimizations (ASC, RT, CP) provides the best performance at all scales, with almost no overhead at the lower scales and only a 3% overhead in the worst case at 20 warehouses.

Results for TPC-H with scale factors 1, 3, and 5 are presented in Figure 9. Network optimization (CP) is not included in this figure since it does not benefit TPC-H. Unoptimized Remus incurs an overhead of 22%, 19%, and 18% for scale factor 1, 3, and 5, respectively. On the other hand, RemusDB with memory optimizations has an overhead of 10% for scale factor 1 and an overhead of 6% for both scale factors 3 and 5 – showing much better scalability.

In addition to the experiments shown here, Appendix B presents an experiment that studies the sensitivity of RemusDB to varying the length of the checkpointing interval.

6. RELATED WORK

Several types of HA techniques are used in database systems, sometimes in combination. Many database systems [6, 16, 20, 21] implement some form of active-standby HA, which is also the basis of RemusDB. In active-standby systems, update propagation may be synchronous or asynchronous. With synchronous propagation, transaction commits are not acknowledged to the database client until both the primary and backup systems have durably recorded the update, resulting in what is known as a *2-safe* system [13, 25]. A 2-safe system ensures that a single server failure will not result in lost updates, but synchronous update propagation may introduce substantial performance overhead. In contrast, asynchronous propagation allows transactions to be acknowledged as soon they are committed at the primary. Such *1-safe* systems impose much less overhead during normal operation, but some recently-committed (and acknowledged) transactions may be lost if the primary fails. RemusDB, which is itself an active-standby system, uses asynchronous checkpointing to propagate updates to the backup. However, by controlling the release of output from the primary server, RemusDB ensures that committed transactions are not acknowledged to the client until they are recorded at the backup. Thus, RemusDB is 2-safe. RemusDB also differs from other database active-standby systems in that it protects the entire database server state, and not just the database.

Alternatives to active-standby HA include *shared access* and *multi-master* HA. Under the former approach, multiple database systems share access to a single, reliable database image. The latter approach uses multiple database replicas, each handling queries and updates. More discussion of these alternatives can be found in Appendix C.

Virtualization has been used to provide high availability for arbitrary applications running in virtual machines, by replicating the entire virtual machine as it runs. Replication can be achieved either through event logging and execution replay or whole-machine checkpointing. Event logging requires much less bandwidth than whole-machine checkpointing, but it is not guaranteed to be able to reproduce machine state unless execution can be made *deterministic*. Enforcing determinism on commodity hardware requires careful management of sources of non-determinism [5, 9], and becomes infeasibly expensive to enforce on shared-memory multiprocessor systems [2, 10, 33]. Respec [17] does provide deterministic execution recording and replay of multithreaded applications with good performance by lazily increasing the level of synchronization it enforces depending on whether it observes divergence during replay, but it requires re-execution to be performed on a different core of the same physical system, making it unsuitable for HA applications. For these reasons, the replay-based HA systems of which we are aware support only uniprocessor VMs [26]. Since RemusDB uses whole-machine checkpointing, it supports multiprocessor VMs.

7. CONCLUSION

We presented RemusDB, a system for providing simple transparent DBMS high availability at the virtual machine layer. RemusDB provides active-standby HA and relies on VM checkpointing to propagate state changes from the primary server to the backup server. It can make any DBMS highly available with little or no code changes and it imposes little performance overhead.

We see several directions for future work. One direction is making RemusDB completely transparent to the DBMS. Currently, the commit protection of RemusDB requires code changes to the client communication layer of a DBMS. Implementing these changes in an ODBC or JDBC driver that can be used by any DBMS or in a proxy can eliminate the need for these code changes. Another direction for future work is exploring the possibility of using DBMS-visible deprotection techniques (e.g., memory deprotection) to improve performance and availability. At a higher level, RemusDB's approach to high availability opens up new opportunities for DBMS deployments on commodity hardware in the cloud. High availability could be decoupled from the DBMS and offered as a transparent service by the cloud infrastructure on which the DBMS runs. Exploring these opportunities is another interesting direction for future work.

8. REFERENCES

[1] A. Aboulnaga, K. Salem, A. A. Soror, U. F. Minhas, P. Kokosielis, and S. Kamath. Deploying database appliances in the cloud. *IEEE Data Eng. Bull.*, 32(1):13–20, 2009.

[2] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proc. Symp. on Operating Systems Principles (SOSP)*, pages 193–206, 2009.

[3] M. Baker and M. Sullivan. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. In *Proc. USENIX Summer Conf.*, pages 31–43, 1992.

[4] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. Symp. on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[5] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. In *Proc. Symp. on Operating System Principles (SOSP)*, pages 1–11, 1995.

[6] W.-J. Chen, M. Otsuki, P. Descovich, S. Arumuggharaj, T. Kubo, and Y. J. Bi. High availability and disaster recovery options for DB2 on Linux, Unix, and Windows. Technical Report IBM Redbook SG24-7363-01, IBM, 2009.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. Symp. Network Systems Design and Implementation (NSDI)*, pages 273–286, 2005.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proc. Symp. Network Systems Design and Implementation (NSDI)*, pages 161–174, 2008.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. Symp. on Operating Systems Design & Implementation (OSDI)*, pages 211–224, 2002.

[10] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proc. Int. Conf. on Virtual Execution Environments (VEE)*, pages 121–130, 2008.

[11] D. K. Gifford. Weighted voting for replicated data. In *Proc. Symp. on Operating System Principles (SOSP)*, pages 150–162, 1979.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 173–182, 1996.

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

[14] Java TPC-W implementation, PHARM group, University of Wisconsin. [online] http://www.ece.wisc.edu/pharm/tpcw/.

[15] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 134–143, 2000.

[16] D. Komo. *Microsoft SQL Server 2008 R2 High Availability Technologies White Paper*. Microsoft, 2010.

[17] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *Proc. Int. Conf. on Arch. Support for Prog. Lang. and Operating Systems (ASPLOS)*, pages 77–90, 2010.

[18] Linux-HA Project. [online] http://www.linux-ha.org/doc/.

[19] D. R. Llanos. TPCC-UVa: An open-source TPC-C implementation for global performance measurement of computer systems. *SIGMOD Record*, 35(4):6–15, 2006.

[20] MySQL Cluster 7.0 and 7.1: Architecture and new features. A MySQL Technical White Paper by Oracle, 2010.

[21] Oracle. *Oracle Data Guard Concepts and Administration*, 11g release 1 edition, 2008.

[22] Oracle. *MySQL 5.0 Reference Manual*, 2010. Revision 23486, http://dev.mysql.com/doc/refman/5.0/en/.

[23] *Oracle Real Application Clusters 11g Release 2*, 2009.

[24] Percona Tools TPC-C MySQL Benchmark. [online] https://code.launchpad.net/ percona-dev/perconatools/tpcc-mysql.

[25] C. A. Polyzois and H. Garcia-Molina. Evaluation of remote backup algorithms for transaction processing systems. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 423–449, 1992.

[26] D. J. Scales, M. Nelson, and G. Venkitachalam. The design and evaluation of a practical system for fault-tolerant virtual machines. Technical Report VMWare-RT-2010-001, VMWare, 2010.

[27] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *Transactions on Database Systems (TODS)*, 35(1):1–47, 2010.

[28] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[29] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[30] The TPC-H Benchmark. [online] http://www.tpc.org/tpch/.

[31] The TPC-C Benchmark. [online] http://www.tpc.org/tpcc/.

[32] The TPC-W Benchmark. [online] http://www.tpc.org/tpcw/.

[33] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comp. Arch. News*, 31(2):122–135, 2003.

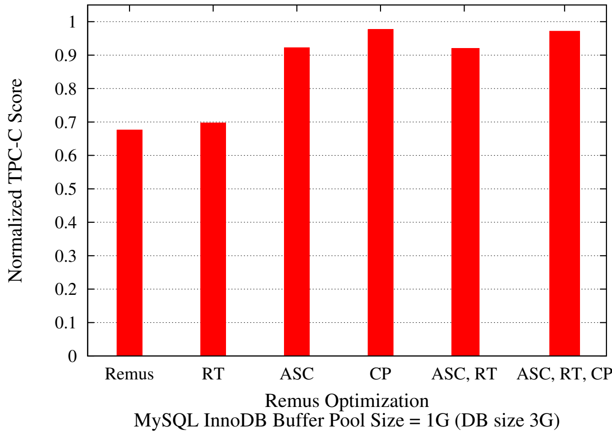

Figure 10: TPC-C Overhead (MySQL).

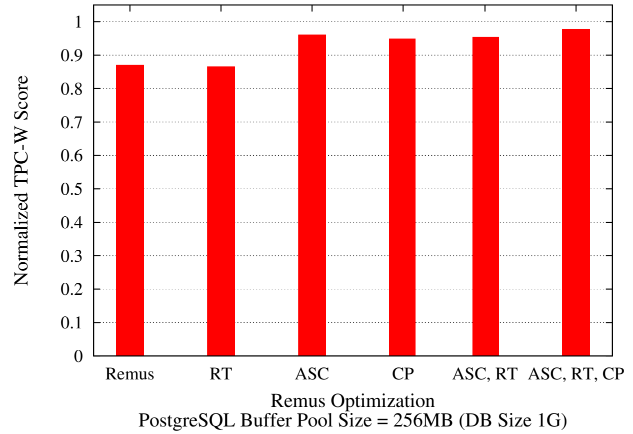

Figure 11: TPC-W Overhead (Postgres).

APPENDIX

A. MEMORY DEPROTECTION

In addition to disk read tracking, described in Section 3.2, we also explored the use of a second memory optimization to reduce the amount of memory protected by RemusDB. This memory optimization aims to provide the DBMS with a more explicit interface to control which portions of its memory should be deprotected (i.e., not replicated during checkpoints). We were surprised to find that we could not produce performance benefits over simple read tracking using this interface.

The idea for memory deprotection stemmed from the Recovery Box [3], a facility for the Sprite OS that replicated a small region of memory that would provide important recent data structures to speed up recovery after a crash (Postgres session state is one of their examples). Our intuition was that RemusDB could do the opposite, allowing the majority of memory to be replicated, but also enabling the DBMS to flag high-churn regions of working memory, such as buffer pool descriptor tables, to be explicitly deprotected and a recovery mechanism to be run after failover.

The resulting implementation was an interesting, but ultimately useless interface: The DBMS is allowed to deprotect specific regions of virtual memory, and these addresses are resolved to physical pages and excluded from replication traffic. On failover, the system would continue to run but deprotected memory would suddenly be in an unknown state. To address this, the DBMS registers a *failover callback handler* that is responsible for handling the deprotected memory, typically by regenerating it or dropping active references to it. The failure handler is implemented as an idle thread that becomes active and gets scheduled only after failover, and that runs with all other threads paused. This provides a safe environment to recover the system.

While we were able to provide what we felt was both a natural and efficient implementation to allow the deprotection of arbitrary memory, it is certainly more difficult for an application writer to use than our other optimizations. More importantly, we were unable to identify any easily recoverable data structures for which this mechanism provided a performance benefit over read tracking. One of the reasons for this is that memory deprotection adds CPU overhead for tracking deprotected pages during checkpointing, and the savings from protecting less memory need to outweigh this CPU overhead to result in a net benefit. We still believe that the interface

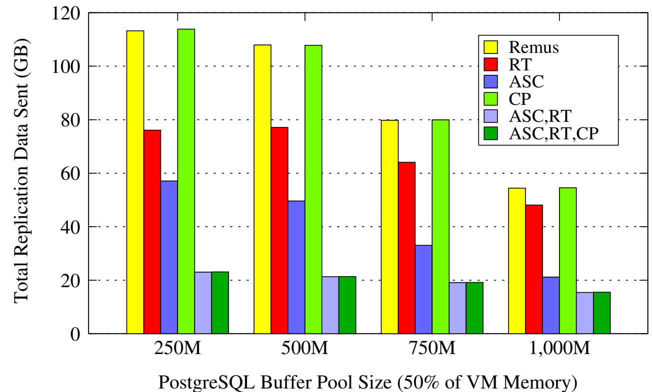

Figure 12: Effect of DB Buffer Pool Size on Amount of Data Transferred During RemusDB Checkpointing (TPC-H).

may be useful for other applications and workloads, but we have decided not to use it in RemusDB.

To illustrate our reasoning, we ran a TPC-H benchmark on Postgres with support for memory deprotection in our experimental setting. Remus introduced 80% overhead relative to an unprotected VM. The first data structure we deprotected was the shared memory segment, which is used largely for the DBMS buffer pool. Unsurprisingly, deprotecting this segment resulted in roughly the same overhead reduction we achieved through read tracking (bringing the overhead down from 80% to 14%), but at the cost of a much more complicated interface. We also deprotected the dynamically allocated memory regions used for query operator scratch space, but that yielded only an additional 1% reduction in overhead. We conclude that for the database workloads we have examined, the *transparency vs. performance* tradeoff offered by memory deprotection is not substantial enough to justify investing effort in complicated recovery logic.

B. ADDITIONAL EXPERIMENTAL RESULTS

B.1 Overhead During Normal Operation

In this experiment, we measure the overhead during normal operation for the TPC-C benchmark running on MySQL and the TPC-W benchmark running on Postgres.

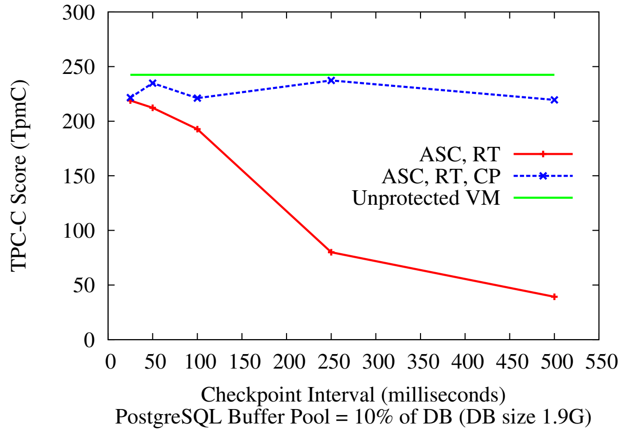

Figure 13: Effect of Checkpoint Interval (TPC-C).

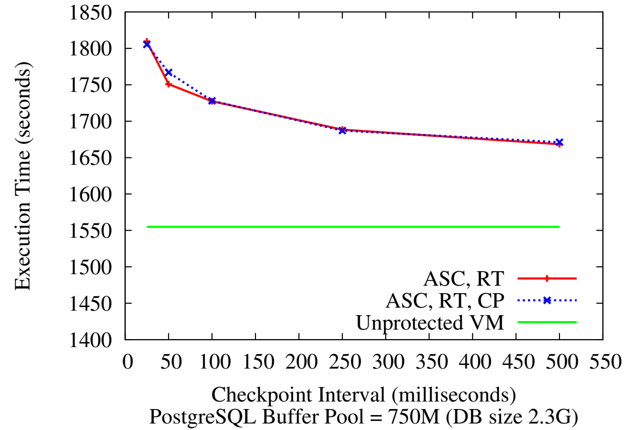

Figure 14: Effect of Checkpoint Interval (TPC-H).

For TPC-C with MySQL we use the settings presented in Table 2 and present the results in Figure 10. On the x-axis, we have different RemusDB optimizations and on the y-axis we present TpmC scores normalized with respect to an unprotected (base) VM. The base VM score for MySQL is 365 TpmC. The conclusions are the same as the ones presented for TPC-C with Postgres in Section 5.3 (Figure 5): network optimizations provide more benefit than memory optimizations because of the sensitivity of the TPC-C benchmark to network latency, while combining various optimizations brings the performance very close to that of an unprotected VM.

For TPC-W experiments with Postgres we use the TPC-W implementation described in [14]. We use a two-tier architecture with Postgres in one tier and three instances of Apache Tomcat v6.0.26 in the second tier, each running in a separate VM. Postgres runs on a virtual machine with 2GB memory and 2 virtual CPUs. We use a TPC-W database with 10,000 items (1GB on disk). Postgres's buffer pool is set to 256MB. Each instance of Apache Tomcat runs in a virtual machine with 1GB memory, and 1 virtual CPU. In these experiments, when running with Remus, only the Postgres VM is protected. In order to avoid the effects of virtual machine scheduling while measuring overhead, we place the Tomcat VMs on a separate, well-provisioned physical machine.

The results for TPC-W with Postgres are presented in Figure 11. Each test was run with the settings presented in Table 2 for a duration of 20 minutes. We drive the load on the database server using 252 Emulated Browsers (EBs) that are equally divided among the three instances of Apache Tomcat. The Apache Tomcat instances access the database to create dynamic web pages and return them to the EBs as specified by the TPC-W benchmark standard [32]. We use the TPC-W *browsing mix* at the clients with image serving turned off. The y-axis on Figure 11 presents TPC-W scores, Web Interactions Per Second (WIPS), normalized to the base VM score (36 WIPS). TPC-W behaves very similar to TPC-C: ASC and CP provide the most benefit while RT provides little or no benefit.

B.2 Network Bandwidth Savings by RemusDB

For this experiment we use the TPC-H benchmark running on Postgres that was presented earlier in Section 5.4 (Figure 7). Figure 12 presents the total amount of data transferred from the primary server to the backup server during checkpointing for the entire duration of the different TPC-H runs. The different bars in Figure 12 correspond to the bars in Figure 7. With a 250MB buffer pool size, unoptimized Remus sends 113GB of data to the backup host while RemusDB with ASC and RT together sends 23GB, a saving of 90GB (or 80%). As we increase the buffer pool size, the network bandwidth savings for RemusDB decrease for the same reasons explained in Section 5.4: with increasing buffer pool size the rate of memory dirtying decreases, and so do the benefits of memory optimizations, both in terms of total execution time and network savings. Recall that CP is not concerned with checkpoint size, and hence it has no effect on the amount of data transferred.

B.3 Effects of RemusDB Checkpoint Interval

This experiment aims to explore the relationship between RemusDB's checkpoint interval (CPI) and the corresponding performance overhead. We conducted this experiment with TPC-C and TPC-H, which are representatives of two very different classes of workloads. We run each benchmark on Postgres, varying the CPI from 25ms to 500ms. Results are presented in Figures 13 and 14 for TPC-C and TPC-H, respectively. We vary CPI on the x-axis, and we show on the y-axis TpmC for TPC-C (higher is better) and total execution time for TPC-H (lower is better). The figures show how different CPI values affect RemusDB's performance when running with (ASC, RT) and with (ASC, RT, CP) combined, compared to an unprotected VM.

From the TPC-C results presented in Figure 13, we see that for (ASC, RT) TpmC drops significantly with increasing CPI, going from a relative overhead of 10% for 25ms to 84% for 500ms. This is to be expected because, as noted earlier, TPC-C is highly sensitive to network latency. Without RemusDB's network optimization (CP), every packet incurs a delay of $\frac{CPI}{2}$ milliseconds on average. With a benchmark like TPC-C where a lot of packet exchanges happen between clients and the DBMS during a typical benchmark run, this delay per packet results in low throughput and high transaction response times. When run with memory (ASC, RT) and network (CP) optimizations combined, RemusDB's performance is very close to that of unprotected VM, with a relative overhead $\leq 9\%$ for all CPIs.

On the other hand, the results of this experiment for TPC-H (Figure 14) present a very different story. In contrast to TPC-C, increasing CPI actually leads to reduced execution time for TPC-H. This is because TPC-H is not sensitive to network latency but is sensitive to the overhead of checkpointing, and a longer CPI means fewer checkpoints. The relative overhead goes from 14% for 25ms CPI to 7% for 500ms. We see a similar trend for both (ASC, RT) and (ASC, RT, CP) since CP does not help TPC-H (recall Figure 7).

There is an inherent trade-off between RemusDB's CPI, work lost on failure, and performance. Choosing a high CPI results in more lost state after a failover since all state generated during an epoch (between two consecutive checkpoints) will be lost, while choosing a low CPI results in a high runtime overhead during normal execution for certain types of workloads. This experiment shows how RemusDB's optimizations, and in particular the network optimization (CP), helps relax this trade-off for network sensitive workloads. For compute intensive workloads that are also insensitive to latency (e.g., TPC-H), choosing a higher CPI actually helps performance.

C. ADDITIONAL RELATED WORK

As noted in Section 6, alternatives to active-standby HA include shared access and multi-master approaches. In shared access approaches, two or more database server instances share a common storage infrastructure, which holds the database. The storage infrastructure stores data redundantly, e.g., by mirroring it on multiple devices, so that it is reliable. In addition, the storage interconnect, through which the servers access the stored data (e.g., a SAN), is made reliable through the use of redundant access pathways. In case of a database server failure, other servers with access to the same database can take over the failed server's workload. Examples of this approach include Oracle RAC [23], which implements a virtual shared buffer pool across server instances, failover clustering in Microsoft SQL Server [16], and synchronized data nodes accessed through the NDB backend API in MySQL Cluster [20]. RemusDB differs from these techniques in that it does not rely on a shared storage infrastructure.

Like active-standby systems, multi-master systems (also known as update anywhere or group systems [12]) achieve high availability through replication. Multi-master systems relax the restriction that all updates must be performed at a single site. Instead, all replicas handle user requests, including updates. Replicas then propagate changes to other replicas, which must order and apply the changes locally. Various techniques, such as those based on quorum consensus [11, 29] or on the availability of an underlying atomic broadcast mechanism [15], can be used to synchronize updates so that global one-copy serializability is achieved across all of the replicas. However, these techniques introduce both performance overhead and complexity. Alternatively, it is possible to give up on serializablility and expose inconsistencies to applications. However, these inconsistencies must then somehow be resolved, often by applications or by human administrators. RemusDB is based on the simpler active-standby model, so it does not need to address the update synchronization problems faced by multi-master systems.