The VLDB Journal (2013) 22:29–45
DOI 10.1007/s00778-012-0294-6

SPECIAL ISSUE PAPER

RemusDB: transparent high availability for database systems

Umar Farooq Minhas · Shriram Rajagopalan · Brendan Cully · Ashraf Aboulnaga · Kenneth Salem · Andrew Warfield

Received: 27 January 2012 / Revised: 30 July 2012 / Accepted: 9 September 2012 / Published online: 18 October 2012

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 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 fast failover ($\leq$3 s 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.

Keywords High availability · Fault tolerance · Virtualization · Checkpointing · Performance modeling

U. F. Minhas (✉) · A. Aboulnaga · K. Salem
University of Waterloo, Waterloo, Canada
e-mail: ufminhas@cs.uwaterloo.ca

A. Aboulnaga
e-mail: ashraf@cs.uwaterloo.ca

K. Salem
e-mail: kmsalem@cs.uwaterloo.ca

S. Rajagopalan · B. Cully · A. Warfield
University of British Columbia, Vancouver, Canada
e-mail: rshriram@cs.ubc.ca

B. Cully
e-mail: brendan@cs.ubc.ca

A. Warfield
e-mail: andy@cs.ubc.ca

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 [31], 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 scale-out, 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 experienced 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 (Sect. 4). 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 (Sect. 5). We also describe how RemusDB reprotects a VM after failure by synchronizing the primary VM with the backup VM after the primary VM comes back online (Sect. 6). We use RemusDB to add high availability to Postgres and MySQL, and we experimentally demonstrate that it effectively recovers from failures and imposes low overhead on normal operation (Sect. 7). For example, as compared to Remus, RemusDB achieves a performance improvement of 29 and 30 % for TPC-C workload running under Postgres and MySQL, respectively. It is also able to recover from a failure in $\leq$3 s while incurring only 3 % performance overhead with respect to an unprotected VM.

An earlier version of this paper appeared in [21]. In this extended version, we add more details about the background and motivation for our work (Sect. 3), a proof of correctness of our approch for reducing communication latency (Sect. 5.1), and a description of reprotection after failure (Sect. 6).

2 Background and system overview

In our setup, shown in Fig. 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

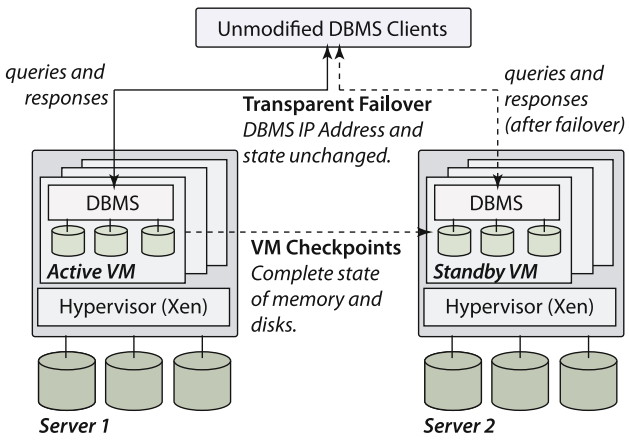

Fig. 1 RemusDB system architecture

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. The time between two checkpoints is referred to as an *epoch*. 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. When the standby VM takes over after a failure, it 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.

A detailed description of Remus's checkpointing mechanism can be found in [8]. Here we present a brief overview. Remus's checkpoints capture the entire state of the active VM, which includes disk, memory, CPU, and network device state. For disk checkpointing, Remus uses an asynchronous disk replication mechanism with writes being applied to the active VM's disk and at the same time asynchronously replicated and buffered in memory at the standby VM, until the end of the current epoch. When the next checkpoint command is received at the standby VM, it flushes the buffered writes to its local disk. If failure happens in the middle of an epoch, the in-memory state is discarded and the standby VM is resumed with a consistent state from the last committed checkpoint on its local disk. Memory and CPU checkpoints are implemented very similar to live VM migration [7] with several optimizations discussed in [8].

Remus's checkpoints capture both the state of the database and the internal execution state of the DBMS, for example, 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 with no loss of client connections is an important advantage of our approach. Some DBMS-level HA solutions provide similar features, but these features add more code and complexity

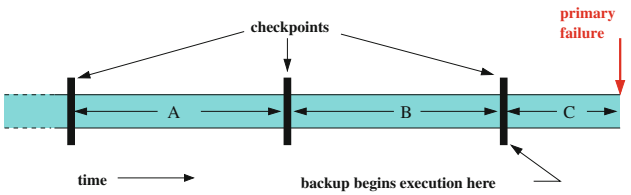

Fig. 2 A primary server execution timeline

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 acknowledgment is carefully overlapped with execution to increase performance while maintaining consistency [8]. However, the simplified timeline shown in Fig. 2 is sufficient to illustrate the important features of this approach to DBMS high availability. When the failure occurs in Fig. 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* [32] 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 Fig. 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 acknowledgment 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 an overview of RemusDB in Sect. 3 and discuss the details of these optimizations in Sects. 4 and 5.

3 Adapting Remus to database workloads

While the simplicity and transparency with which Remus provides high availability is desirable, applying Remus to database workloads is not an ideal fit for a number of reasons. First, as described above, Remus continuously transmits checkpoints of the running virtual machine to the backup host, resulting in a steady flow of replication traffic that is proportional to the amount of memory that has changed between checkpoints; the large amount of memory churn in database workloads results in a high degree of replication traffic. The large amount of replication data makes checkpoints slower and results in a significant performance overhead for database workloads.

Second, the fact that Remus controls output commit by buffering every transmitted packet is over-conservative for database systems, which already provide higher-level transactional semantics. Client–server interactions with a database system typically involve several round trips on the fast, local area network. Within a transaction, the delay introduced by network buffering on messages from the server in each round trip results in an amplification of Remus's existing latency overheads. Moreover, the high memory churn rate of database workloads compounds this problem by requiring longer checkpoint epochs, resulting in longer delays for network buffering. For example, in a run of the TPC-C benchmark on Postgres in our experimental setting (described in Sect. 7), we observed that Remus introduced an overhead of 32 % compared to the unprotected case. Turning off network buffering for this benchmark run reduced the overhead to 7 %.

In designing RemusDB, we aimed to adapt Remus to address these two issues. In both cases, we observed that Remus's goal of providing HA that is completely transparent to the DBMS was excessively conservative and could be relaxed, resulting in a large reduction in overhead. More precisely, we made the following two observations:

1. *Not all changes to memory need to be sent to the backup.* In attempting to maintain an exact replica of the protected VM on the backup system, Remus was transmitting every page of memory whose contents changed between epochs. However, many page updates can either be reconstructed, as with clean pages in the buffer pool that can be reloaded from disk, or thrown away altogether, in the case of working memory that can be recomputed or safely lost.
2. *Not all transmitted messages need output commit.* Buffering transmitted messages until the checkpoint that generated them has been protected prevents the system from exposing execution state that is rolled back (and so lost) in the event of failure. In a DBMS environment, this intermediate state is already protected by transaction boundaries. In light of this, we may relax output commit to the point that it preserves transactional semantics.

In addition to relaxing the comprehensiveness of protection in Remus to reduce overhead, our analysis of database workloads revealed one additional insight about these workloads that allowed further optimization:

3. *While changes to memory are frequent, they are often small.* Remus uses hardware page protection to identify the pages that have changed in a given checkpoint and then transfers those pages to the backup at page granularity. Our analysis revealed that memory updates in database workloads were often considerably smaller than page size and could consequently be compressed fairly effectively.

Remus was adapted in light of each of these observations, in order to provide more efficient high availability for database workloads. Section 4 discusses optimizations related to how memory is tracked on the primary VM and replicated over the network to the backup. Section 5 describes how latency overheads have been reduced by relaxing network buffering in some situations.

4 Memory optimizations

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

4.1 Sending less data

Compressing checkpoints is beneficial when the amount of data to be replicated is large, and the data contain redundancy. Our analysis found that both of these conditions apply to database workloads: (1) they involve a large set of frequently changing pages of memory (most notably buffer

pool pages), and (2) 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.

When the Xen hypervisor runs on a physical machine, it creates a privileged VM called *domain 0* for controlling other VMs running on that physical machine. Domain 0 serves as an administrative front-end for the Xen hypervisor and manages the creation, configuration, and control of other VMs on the physical machine. In RemusDB, we implement checkpoint compression by maintaining in domain 0 an LRU-based cache of frequently changing pages in the protected VM obtained from previous checkpoints in that VM. A per-VM cache is maintained in domain 0 if there are multiple protected VMs. 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 whether 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 the performance impact of checkpointing. 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. Figure 3 illustrates the workflow for checkpoint compression.

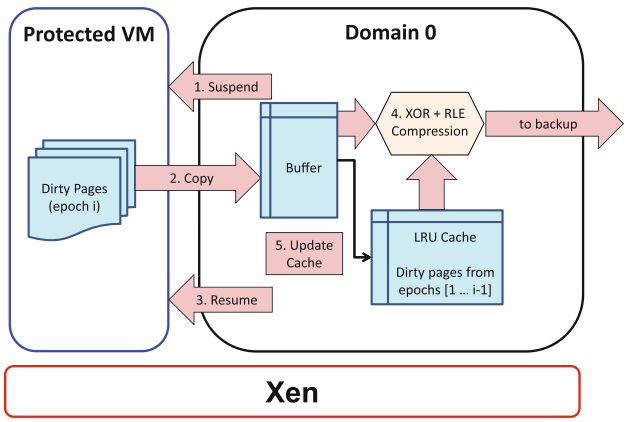

Fig. 3 Checkpoint compression workflow

4.2 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*.

4.2.1 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 and 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 VM, 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. 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 back-pointers 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.

4.2.2 Memory deprotection

Our second 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 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* versus *performance* tradeoff offered by memory deprotection is not substantial enough to justify investing effort in complicated recovery logic. Hence, we do not use memory deprotection in RemusDB.

5 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 Postgres 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.

In RemusDB, we exploit database transaction semantics to avoid much of Remus's network buffering, and hence to eliminate much of the performance overhead that network buffering introduces. The purpose of network buffering in Remus is to avoid exposing the client to the results of speculative server processing until it has been checkpointed. In RemusDB, we relax this behavior by allowing server communications resulting from speculative processing to be released immediately to the client, *but only within the scope of an active database transaction*. If the client attempts to commit a transaction, RemusDB will buffer and delay the commit acknowledgment until that transaction is safe, that is, until

the processing of that transaction has been checkpointed. Conversely, if a failure occurs during the execution of such a transaction, RemusDB will ensure that the transaction is aborted on failover. Relaxing Remus's network buffering in this way allows RemusDB to release most outgoing network packets without delay. However, failover is no longer completely transparent to the client, as it would be in Remus, as a failover may necessitate the abort of some in-progress transactions. As long as failures are infrequent, we expect this to be a desirable tradeoff.

To implement this approach in RemusDB, we modified the hosted DBMS to implement a protocol we call *commit protection*. The commit protection protocol requires fine (message level) control over which outgoing messages experience network buffering and which do not. To support this, RemusDB generalizes Remus's communication abstraction. Stream sockets, which are implemented on top of TCP, guarantee in-order message delivery, and database systems normally use stream sockets for communication with clients. In RemusDB, each stream socket can be in one of two states: *protected* or *unprotected*. RemusDB provides the hosted DBMS with *protect* and *deprotect* operations to allow it to change the socket state. Outgoing messages sent through a protected socket experience normal Remus network buffering, that is, they are delayed until the completion of the next Remus commit. Messages sent through an unprotected socket are not subjected to network buffering and are released immediately. RemusDB preserves in-order delivery of all messages delivered through a socket, regardless of the state of the socket when the message is sent. Thus, an unprotected message sent shortly after a protected message may be delayed to ensure that it is delivered in the correct order.

The hosted DBMS implements the commit protection protocol using the socket protection mechanism. The commit protocol has two parts, as shown in Fig. 4. The first part of the protocol runs when a client requests that a transaction commit. The server protects the transaction's socket before starting commit processing, at it remains protected until after sending the commit acknowledgment to the client. All transaction output up until the arrival of the commit request is sent unprotected. The second part of the protocol runs at the standby server after a failover, and causes all active transactions that are not committing to abort. Remus is designed to run a recovery thread in the standby VM as soon as it takes over after a failure. In RemusDB, the recovery thread runs inside the standby DBMS and implements the failover part of the commit protection protocol. Once the recovery thread finishes this work, the DBMS resumes execution from state captured by the most recent pre-failover checkpoint. Note that, on failover, the recovery handler will see transaction states as they were at the time of the last pre-failure checkpoint.

At COMMIT WORK:

```
protect the client's socket
perform normal DBMS commit processing
send the COMMIT acknowledgement
deprotect the socket
```

On failover (at the standby host):

```
for each active transaction t do
    if t is not committing then ABORT t
end for
```

Fig. 4 The commit protection protocol

5.1 Correctness of commit protection

In this section, we state more precisely what it means for the commit protection protocol to behave correctly. Essentially, if the client is told that a transaction has committed, and then that transaction should remain committed after a failure. Furthermore, if an active transaction has shown speculative results to the client and a failure occurs, then that transaction must ultimately abort. These guarantees are formalized in the following lemmas.

Lemma 1 (Fail-safe commit) *For each transaction T that is created at the active server prior to the point of failure, if a client receives a commit acknowledgment for T, then T will be committed at the standby site after failover.*

Proof COMMIT WORK acknowledgments are always sent using a protected socket, which does not release messages to the client until a checkpoint has occurred. If the client has received a commit acknowledgment for T, then a server checkpoint must have occurred after T's commit message was sent and thus after the active server made the commit decision for T. Thus, the active server's commit decision for T (and T's effects) will be captured by the checkpoint and reflected at the standby site after the failure. Furthermore, at the time of the checkpoint, T will either have been committing at the active site or it will have finished. Since the recovery thread at the standby site only aborts active transactions that are not committing, it will not attempt to abort T. □

Lemma 2 (Speculation) *For each transaction T that is created at the active server prior to the point of failure, if T's client does not submit a COMMIT WORK request for T prior to the failure, then either T will be aborted at the standby server after the failure, or it will not exist there at all.*

Proof Let C represent the last checkpoint at the active server prior to its failure. There are three cases to consider. First, T may have started after C. In this case, T will not exist at the time of the checkpoint C, and therefore, it will not exist at the standby server after failover. Second, T may have started

Table 1 RemusDB source code modifications (lines of code)

	Virtualization layer	Guest VM Kernel	DBMS
Commit protection	13	396	103 (Postgres), 85 (MySQL)
Disk read tracking	1, 903	0	0
Compression	593	0	0

before C and remained active at C. In this case, some of T's effects may be present at the standby site because they are captured by C. Since the client has not submitted a COMMIT WORK request, T cannot have been committing at the time of C. Therefore, the recovery thread will see T as an active, non-committing transaction at failover and will abort T. The third case is that T may have started, aborted, and finished prior to C. In this case, the checkpoint will ensure that T is also aborted at the standby site after failover. □

5.2 Implementation of protection and deprotection

To provide a DBMS with the ability to dynamically switch a client connection between protected and unprotected modes, we added a new `setsockopt()` option to Linux. A DBMS has to be modified to make use of protection and deprotection via the commit protection protocol shown in Fig. 4. We have implemented commit protection in Postgres 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.

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 [33] 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 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.

6 Reprotection after failure

When the primary host crashes, the backup host takes over and becomes the new primary. When the original, now-crashed primary host comes back online, it needs to assume the role of backup host. For that to happen, the storage (i.e., disks) of the VMs on the two hosts must be resynchronized. The storage of the VM on the host that was failed and is now back online must catch up with the storage of the VM on the other, now-primary host. After this storage synchronization step, checkpointing traffic can resume between the primary and backup host.

For the protected DBMS to remain available during storage synchronization, this synchronization must happen online, while the DBMS in the primary VM is running. The storage replication driver used by Remus is based on Xen's Blktap2 driver [38] and does not provide a means for online resynchronization of storage. One way to perform the required online resynchronization is to use a brute-force approach and copy all the disk blocks from the primary to the backup. This is sufficient to ensure correctness, but it would impose unnecessary load on the disk subsystem and increase the time to restart HA. A better approach, which we adopt in this paper, is used by the SecondSite system [29]. In this approach, only the disk blocks changed by the new primary / old backup VM after failure are copied over. The system also overwrites disk blocks written by the old primary VM during the last unfinished checkpoint with data from the backup.

7 Experimental evaluation

In this section, we present an evaluation of RemusDB. The objectives of this evaluation are as follows:

- First, we wish to demonstrate that RemusDB is able to survive a failure of the primary server and to illustrate the performance of RemusDB during and after a failover.
- Second, we wish to characterize the performance overhead associated with RemusDB during normal operation. We compare the performance of unoptimized Remus and optimized RemusDB against that of an unprotected DBMS to measure this overhead. We also consider the impact of specific RemusDB optimizations on different types of database workloads.
- Third, we consider how key system parameters and characteristics, such as the size of the DBMS buffer pool and the length of the Remus checkpoint interval, affect the overhead introduced by Remus.

7.1 Experimental environment

Our experimental setup consists of two servers each equipped with two quad-core Intel Xeon processors, 16 GB RAM, and two 500 GB 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, using three widely accepted benchmarks namely: TPC-C [35], TPC-H [36], and TPC-W [37]. We run TPC-C experiments on both Postgres and MySQL while TPC-H and TPC-W experiments are run on Postgres only. We use a Remus checkpointing interval (CPI) of 50, 100, and 250 ms for TPC-C, TPC-W, and TPC-H experiments, respectively. These different CPIs for each type of benchmark are chosen because they offer the best trade-off between overhead during normal execution and availability requirements of that particular workload. We evaluate the effect of varying CPI on the TPC-C and TPC-H benchmarks in Sect. 7.6.

Our default settings for TPC-C experiments are as follows: The virtual machine is configured with 2 GB memory and 2 virtual CPUs. For MySQL, we use the Percona benchmark kit [27] with a database of 30 warehouses and 300 concurrent clients (10 clients per warehouse). The total size of the database on disk is 3 GB. For Postgres, we use the TPCC-UVa benchmark kit [20] with a database of 20 warehouses (1.9 GB database on disk) and 200 concurrent clients. We modified the TPCC-UVa benchmark kit so that it uses one TCP connection per client; the original benchmark kit uses one shared connection for all clients. We choose different scales for MySQL and Postgres due to differences in how they scale to larger workloads when provided with fixed (equal) resources. The database buffer pool is configured to be 10 % of the database size on disk. We do not use connection pooling or a transaction monitor; each client directly connects to the DBMS.

Our default settings for TPC-H experiments are a virtual machine with 1.5 GB memory, 2 virtual CPUs, and a database with TPC-H scale factor 1. The total size of the database on disk is 2.3 GB. We configure Postgres with a buffer pool size of 750 MB. Our TPC-H experiments consist of one *warmup* run where we execute the 22 read-only TPC-H queries sequentially, followed by one *power stream* run [36] where we execute the queries sequentially and measure the total execution time. We do not perform TPC-H throughput tests or use the refresh streams.

Lastly, for TPC-W experiments, we use the TPC-W implementation described in [15]. 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 2 GB memory, and 2 virtual CPUs. We use a TPC-W database with 10,000 items (1 GB on disk). Postgres's buffer pool is configured to be 256 MB. Each instance of Apache Tomcat runs in a virtual machine with 1 GB 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.

Table 2 provides a summary of our experimental settings. 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.

7.2 Behavior 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 run the test for 1 h; a failure of the primary host is simulated at 30 min by cutting power to it. We compare the performance of a database system protected by unoptimized Remus and by RemusDB with its two transparent optimizations (ASC, RT) in Figs. 5 and 6. 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 s. 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.

Without any mechanism for high availability in place, the unprotected VM cannot serve clients beyond the fail-

Table 2 Experimental settings

DBMS	Benchmark	Performance metric	Default scale	Test duration (min)	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	10 K Items	20	1.0	256	2	2	100
MySQL	TPC-C	TpmC	30W, 300C	30	3.0	300	2	2	50

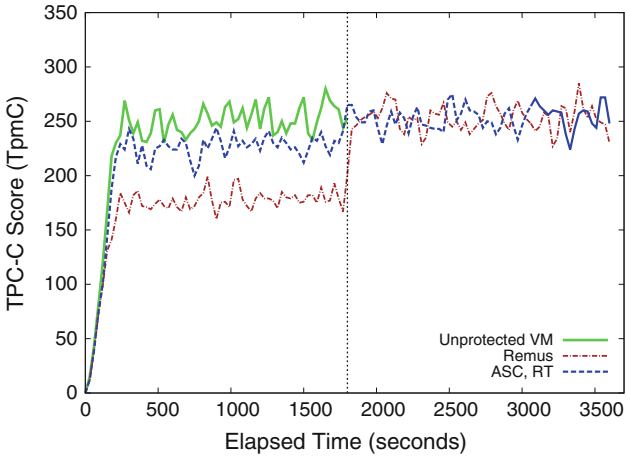

Fig. 5 TPC-C failover (Postgres)

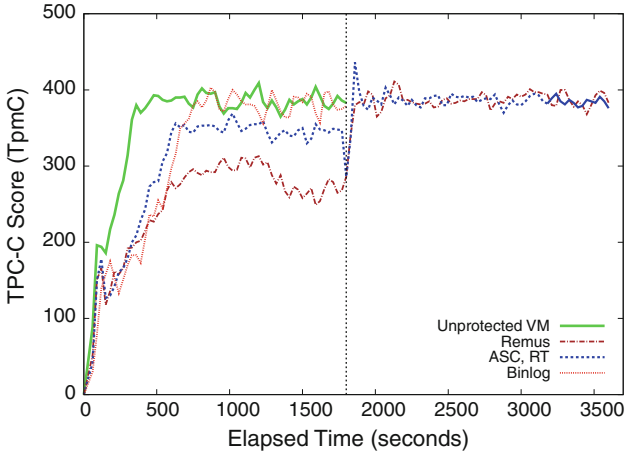

Fig. 6 TPC-C failover (MySQL)

ure point, that is, throughput immediately drops to zero. All clients lose connections to the database server and cannot reconnect until someone (e.g., a DBA) manually restores the database to its pre-failure state. After restart, the database will recover from its crash consistent state using standard log recovery procedures [22]. The time to recover depends on how much state needs to be read from the write-ahead log and reapplied to the database and is usually in the order of several minutes. Furthermore, the unprotected VM will have to go through a warm-up phase again before it can reach its pre-failure steady state throughput (not shown in the graph).

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 s 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 6 also shows results with MySQL's integrated replication solution, Binlog replication [26, 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 [19], provides performance very close to that of an unprotected VM and can recover from a failure with $\leq$5 s 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 where every write to the binary log is synchronized to disk [26, Ch. 16.1.1.1]. In case of a crash, this conservative setting will result in a loss of at most one statement or transaction from the binary log. This is because at any given time, the log record for only a single statement or transaction will be waiting to be synchronized to disk and can be lost in case of a crash before the synchronization is complete. In contrast, even with aggressive optimizations such as commit protection, RemusDB never loses transactions.
- *Transparency*. Client-side recovery is more complex with Binlog replication, which loses all existing client sessions at failure. To show Binlog performance after recovery in Figure 6, 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 [35, 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.

7.3 Reprotection after a failure

In Fig. 7, we show RemusDB's reprotection mechanism in action. Similar to the failover experiment in Sect. 7.2, we run the TPC-C benchmark against Postgres and plot throughput in transactions per minute (TpmC). We run the test for 1 h, a failure of the primary host is simulated at 30 min by cutting power to it. The performance of an unprotected database system (without HA) is also shown for reference. The setting used for these experiments is slightly different than the other experiments in this section. In particular, the storage backend used for reprotection experiments is different since it supports online resynchronization of VM disks after a failure. Because of that, the performance numbers are slightly lower than other experiments, as can be clearly seen from the line for unmodified Remus.

During the outage period, the VM does not incur any checkpointing overhead and hence the throughput rises to that of an unprotected system. After a 15-min outage period, the primary host is brought back online. Note that we let the outage last for 15 min in this experiment in order to adequately observe performance during an outage. In reality, we can detect a failure within 3 s and resynchronize storage within approximately 29 s. The time required to resynchronize can vary depending on the amount of data to be resynchronized. Once storage resynchronization completes, we restart the replication process: all of VM's memory is copied to the backup (the old primary is the new backup), and then Remus checkpoints are resumed. This process takes approximately 10 s in our settings for a VM with 2 GB of memory and a gigabit ethernet. After this point, the VM is once again HA, that is, it is protected against a failure of the backup. Note that after reprotection, the throughput returns back to pre-failure levels as shown in Fig. 7. For implementing reprotection, we utilized a new storage backend namely DRBD [9] which allows efficient online resynchronization of VM disks. This storage backend does not yet support read tracking. Hence, Fig. 7 only shows RemusDB's performance with the ASC optimization.

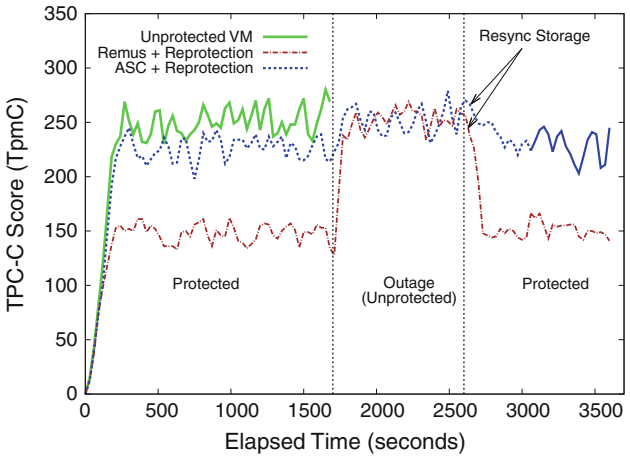

Fig. 7 TPC-C failover and reprotection after failure (Postgres)

7.4 Overhead during normal operation

Having established the effectiveness of RemusDB at protecting from failure and its fast failover and reprotection times, we now turn our attention to the overhead of RemusDB during normal operation. This section serves two goals: (1) it quantifies the overhead imposed by unoptimized Remus on normal operation for different database benchmarks, and (2) it measures how the RemusDB optimizations affect this overhead, when applied individually or in combination. For this experiment, we use the TPC-C, TPC-H, and TPC-W benchmarks.

Figures 8 and 9 present TPC-C benchmark results for Postgres and MySQL, respectively. In each case, the benchmark

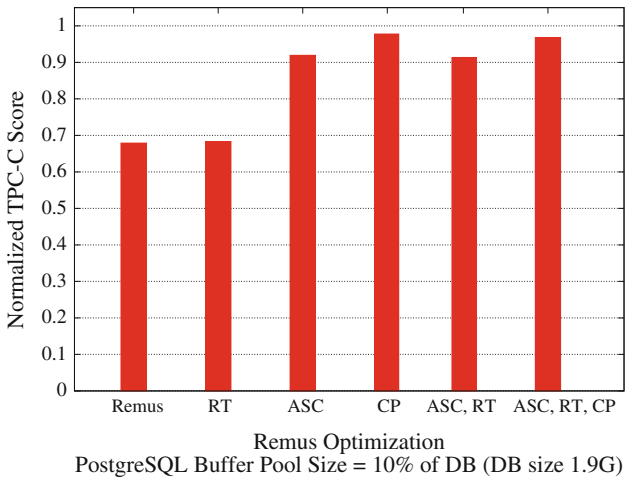

Fig. 8 TPC-C overhead (Postgres) [base score = 243 tpmC]

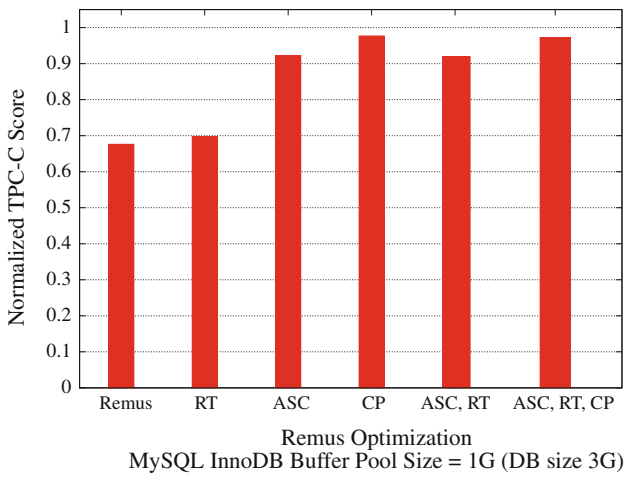

Fig. 9 TPC-C overhead (MySQL) [base score = 365 tpmC]

was run for 30 min using the settings presented in Table 2. 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 normalized score is defined as: (TpmC with optimization being evaluated)/(Base TpmC). The TpmC score reported in these graphs takes into account all transactions during the measurement interval irrespective of their response time requirements. Base VM scores are 243 and 365 TpmC for Postgres and MySQL, respectively. The score of unoptimized Remus (leftmost bar) is around 0.68 of the base VM score for both DBMSes—representing a significant performance loss. It is clear from the graph that without optimizations, Remus protection for database systems comes at a very high cost. The next three bars in the graph show the effect of each RemusDB optimization applied individually. RT provides very little performance benefit because TPC-C has a small working set and dirties many of the pages that it reads. However, both ASC and CP provide significant performance gains. Performance with these optimizations is 0.9–0.97 of the base performance. TPC-C is particularly sensitive to network latency and both of these optimizations help reduce 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 rightmost two bars in the graph show the effect of combining optimizations. 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. In multiple variations of this experiment, we have observed that the variance in performance is always low and that when the combination of (ASC, RT, CP) does not outright outperform the individual optimizations, the difference is within the range of experimental error. The improvement in performance when adding (ASC, RT, CP) to Remus can be seen not only in throughput, but also in latency. The average latency of the NewOrder transactions whose throughput is plotted in Figs. 8 and 9 for Postgres and MySQL, respectively is 12.9 and 19.2 s for unoptimized Remus. This latency is 1.8 and 4.5 s for RemusDB. Compare this to the latency for unprotected VM which is 1.2 s for Postgres and 3.2 s for MySQL. 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 10 presents the results for TPC-H with Postgres. In this case, the y-axis presents the total execution time of a warmup run and a power test run normalized with respect to the base VM's execution time (921 s). The normalized execution time is defined as: (base execution time)/(execution time with optimization being evaluated). 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

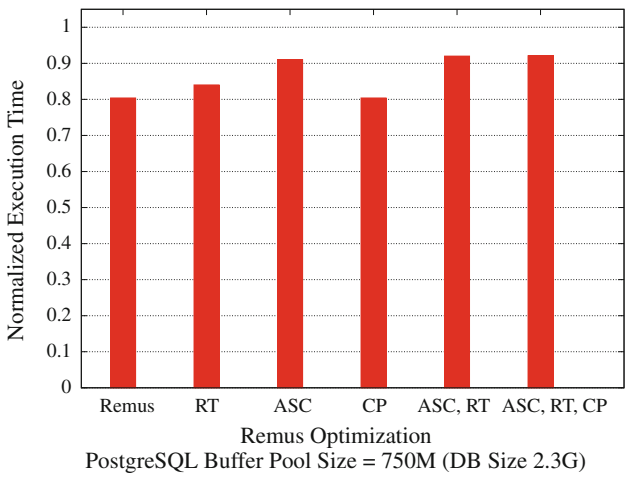

Fig. 10 TPC-H overhead (Postgres) [base runtime = 921 s]

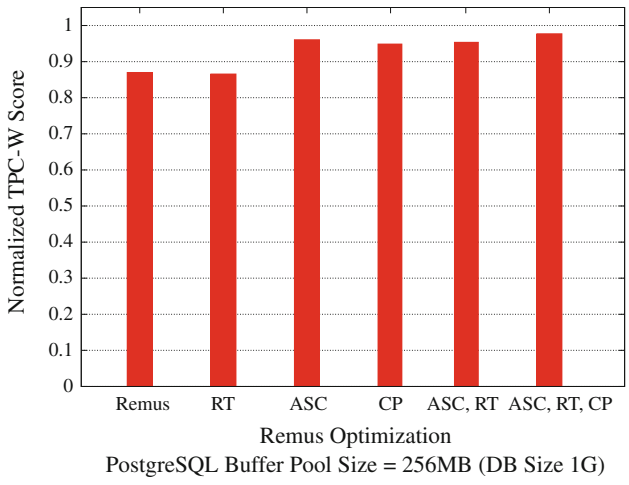

Fig. 11 TPC-W overhead (Postgres)

Sect. 4.2.2) is limited to 10 %. Therefore, we conclude that it is not worth the additional complexity to pursue it.

Finally, we present the results for TPC-W with Postgres in Fig. 11. Each test was run with the settings presented in Table 2 for a duration of 20 min. We drive the load on the database server using 252 Emulated Browsers (EBs) that are equally divided among three instances of Apache Tomcat, which in turn access the database to create dynamic web pages and return them to EBs, as specified by the TPC-W benchmark standard [37]. We use the TPC-W *browsing mix* with image serving turned off at the clients. The *y*-axis on Fig. 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 workload: ASC and CP provide the most benefit while RT does not provide any benefit.

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) works well for OLTP style workloads, while DSS style workloads gain 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.

7.5 Effects of DB buffer pool size

In the previous experiment, we showed that memory optimizations (ASC and RT) offer significant performance gains for the TPC-H workload. The goal of this experiment is to study the effects of database buffer pool size on different memory optimizations on a micro level. In doing so, we hope to offer insights about how each of these optimization offers its performance benefits.

We run a scale factor 1 TPC-H workload, varying the database buffer pool size from 250 to 1,000 MB. 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 250 MB buffer pool, we run the experiment in a 500 MB VM and so on. Results are presented in Fig. 12. The numbers on top of each bar show the relative overhead with respect to an unprotected VM for each buffer pool setting. We calculate this overhead as:

$$\text{Overhead}\,(\%) = \frac{X - B}{B} \times 100$$

where B is the total execution time for an unprotected VM and X is the total execution time for a protected VM with a specific RemusDB optimization.

Focusing on the results with a 250 MB buffer pool in Fig. 12, 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 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 Fig. 12 is that the benefit of RT decreases with increasing buffer pool size. Since the database size is 2.3 GB on disk (Table 2), with a smaller buffer pool size (250 and 500 MB), 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

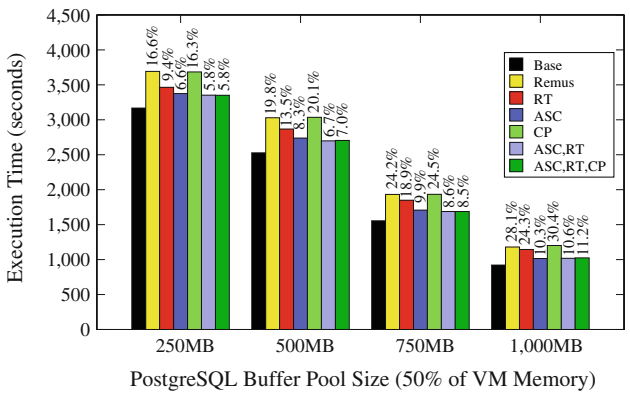

Fig. 12 Effect of DB buffer pool size on RemusDB (TPC-H)

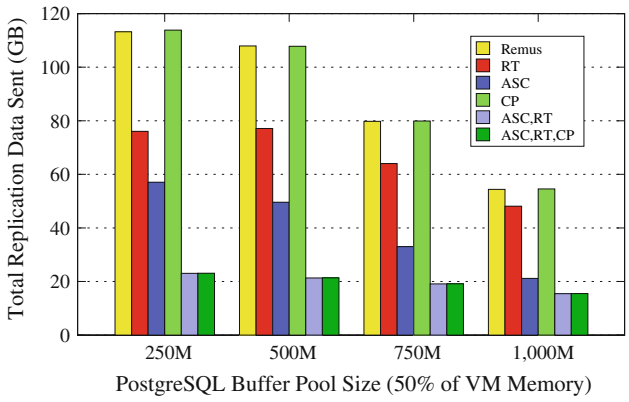

Fig. 13 Effect of DB buffer pool size on amount of data transferred during RemusDB checkpointing (TPC-H)

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.

In Fig. 13, we present the total amount of data transferred from the primary server to the backup server during checkpointing for the entire duration of the experiment. The different bars in Fig. 13 correspond to the bars in Fig. 12. With a 250 MB buffer pool size, unoptimized Remus sends 113 GB of data to the backup host while RemusDB with ASC and RT together sends 23 GB, a saving of 90 GB (or 80 %). As we increase the buffer pool size, the network bandwidth savings for RemusDB also decrease for the same reasons explained above: 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.

7.6 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 25 to 500 ms. Results are presented in Figs. 14 and 15 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 Fig. 14, we see that for (ASC, RT) TpmC drops significantly with increasing CPI, going from a relative overhead of 10 % for 25 ms to 84 % for 500 ms. 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 (Fig. 15) 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 25 ms CPI to 7 % for 500 ms. We see a similar trend for both (ASC, RT) and (ASC, RT, CP) since CP does not help TPC-H (recall Fig. 12).

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

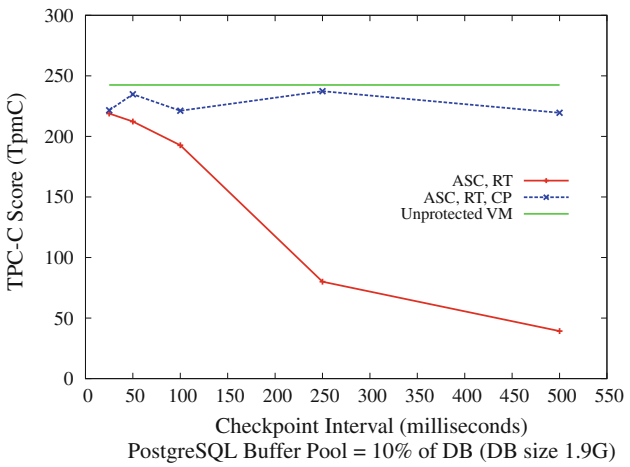

Fig. 14 Effect of checkpoint interval on RemusDB (TPC-C)

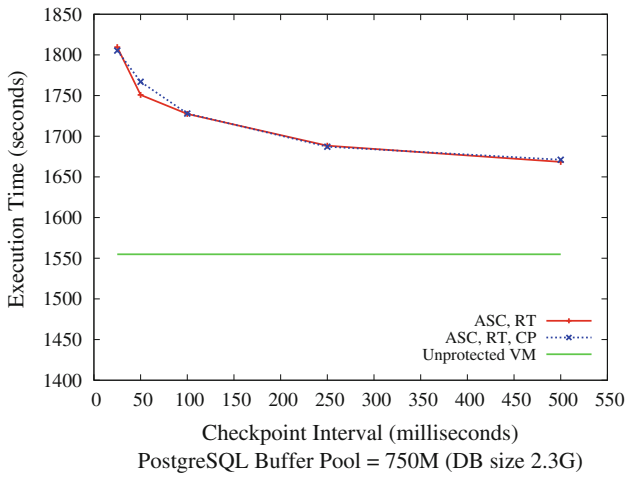

Fig. 15 Effect of checkpoint interval on RemusDB (TPC-H)

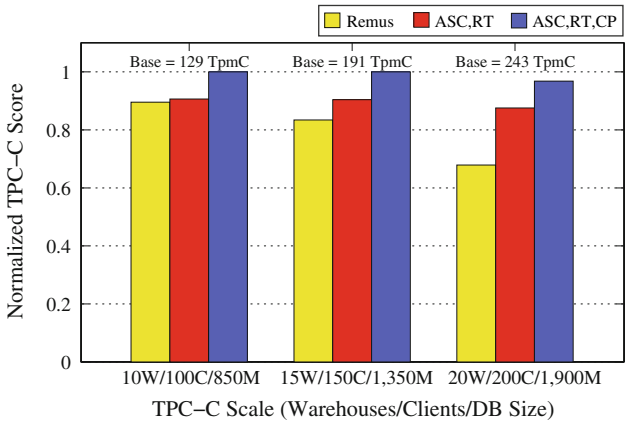

Fig. 16 Effect of database size on RemusDB (TPC-C)

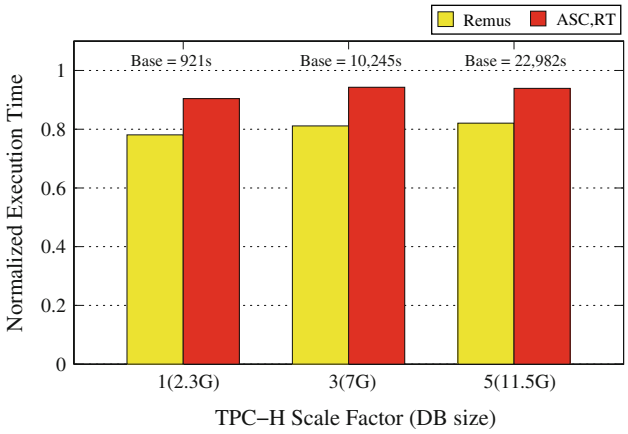

Fig. 17 Effect of database size on RemusDB (TPC-H)

latency (e.g., TPC-H), choosing a higher CPI actually helps performance.

7.7 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 Fig. 16. We use three different scales: (1) 10 warehouses, 100 clients, 850 MB database; (2) 15 warehouses, 150 clients, 1,350 MB database; and (3) 20 warehouses, 200 clients, 1,900 MB 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 Fig. 17. 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.

8 Related work

Widely-used logging and checkpointing techniques, such as ARIES [22], together with database backups, allow DBMS to recover from server failures. After a failure, the DBMS runs a recovery protocol that uses the contents of the log to ensure that the database (or a restored database backup) is in a consistent state that includes all of the effects of transactions that committed before the failure. Once the database has been restored, the DBMS can begin to accept new work. However, since the DBMS cannot perform new work until the database has been restored, the recovery process can lead to an unacceptably long period of unavailability. Thus, many DBMS provide additional high-availability features, which are designed to ensure that little or no down time will result from a server failure.

Several types of HA techniques are used in database systems, sometimes in combination. 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, for example, by mirroring it on multiple devices, so that it is reliable. In addition, the storage interconnect (e.g., a SAN), through which the servers access the stored data, must be 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 [25], which implements a virtual shared buffer pool across server instances, failover clustering in Microsoft SQL Server [17], and synchronized data nodes accessed through the NDB backend API in MySQL Cluster [23]. RemusDB differs from these techniques in that it does not rely on a shared storage infrastructure.

Active-standby approaches, which we introduced in Sect. 1, are designed to operate in a shared-nothing environment. Many database systems [6,17,23,24] implement some form of active-standby HA. In some cases, the primary and backup can be run in an active-active configuration, allowing some read-only application work to be performed against the slave database, which may be slightly stale with respect to the primary.

In active-standby systems, update propagation may be physical, logical (row-based), or statement-based. Propaga-

tion, which is sometimes known as *log shipping*, may be synchronous or asynchronous. In the former case, transaction commits are not acknowledged to the database client until both the active and standby systems have durably recorded the update, resulting in what is known as a *2-safe* system [14,28]. 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 standby. 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 standby. 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.

Like active-standby systems, multi-master systems (also known as update anywhere or group systems [13]) 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 [12,34] or on the availability of an underlying atomic broadcast mechanism [16], 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 need not address the update synchronization problems faced by multi-master systems.

Virtualization has been used to provide high availability for arbitrary applications running inside 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. While event logging requires much less bandwidth than whole-machine checkpointing, 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 [5,10] and becomes infeasibly expensive to enforce on shared-memory multiprocessor systems [2,11,39]. Respec [18] 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 intricate modifications to the operating system. It also 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 [30]. RemusDB uses whole-machine checkpointing, so it supports multiprocessor VMs.

9 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. Our experiments demonstrate that RemusDB provides fast failover and imposes little performance overhead during normal operation.

References

1. Aboulnaga, A., Salem, K., Soror, A.A., Minhas, U.F., Kokosielis, P., Kamath, S.: Deploying database appliances in the cloud. IEEE Data Eng. Bull. **32**(1), 13–20 (2009)
2. Altekar, G., Stoica, I.: ODR: output-deterministic replay for multi-core debugging. In: Symposium on Operating Systems Principles (2009)
3. Baker, M., Sullivan, M.: The recovery box: using fast recovery to provide high availability in the UNIX environment. In: USENIX Summer Conference (1992)
4. Barham, P.T., Dragovic, B., Fraser, K., Hand, S., Harris, T.L., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Symposium on Operating Systems Principles (SOSP) (2003)
5. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault-tolerance. In: Symposium on Operating Systems Principles (SOSP) (1995)
6. Chen, W.J., Otsuki, M., Descovich, P., Arumuggharaj, S., Kubo, T., Bi, Y.J.: High availability and disaster recovery options for DB2 on Linux, Unix, and Windows. Tech. Rep. IBM Redbook SG24-7363-01, IBM (2009)
7. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: Symposium on Networked Systems Design and Implementation (NSDI) (2005)
8. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: High availability via asynchronous virtual machine replication. In: Symposium Networked Systems Design and Implementation (NSDI) (2008)
9. Distributed Replicated Block Device (DRBD): http://www.drbd.org/ (2008)

10. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In: Symposium on Operating Systems Design and Implementation (OSDI) (2002)
11. Dunlap, G.W., Lucchetti, D.G., Fetterman, M.A., Chen, P.M.: Execution replay of multiprocessor virtual machines. In: Virtual Execution Environments (VEE) (2008)
12. Gifford, D.K.: Weighted voting for replicated data. In: Symposium on Operating Systems Principles (SOSP) (1979)
13. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: International Conference on Management of Data (SIGMOD) (1996)
14. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, Los Altos (1993)
15. Java TPC-W implementation, PHARM group, University of Wisconsin. http://www.ece.wisc.edu/pharm/tpcw/ (1999)
16. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In: International Conference on Very Large Data Bases (VLDB) (2000)
17. Komo, D.: Microsoft SQL Server 2008 R2 High Availability Technologies White Paper. Microsoft (2010)
18. Lee, D., Wester, B., Veeraraghavan, K., Narayanasamy, S., Chen, P.M., Flinn, J.: Respec: efficient online multiprocessor replayvia speculation and external determinism. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2010)
19. Linux-HA Project: http://www.linux-ha.org/doc/ (1999)
20. Llanos, D.R.: TPCC-UVa: an open-source TPC-C implementation for global performance measurement of computer systems. SIGMOD Rec. **35**(4), 6–15 (2006)
21. Minhas, U.F., Rajagopalan, S., Cully, B., Aboulnaga, A., Salem, K., Warfield, A.: RemusDB: Transparent high availability for database systems. Proc. VLDB Endow. (PVLDB) **4**(11), 738–748 (2011)
22. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. Trans. Database Syst. (TODS) **17**(1), 94–162 (1992)
23. MySQL Cluster 7.0 and 7.1: Architecture and New Features. A MySQL Technical White Paper by Oracle (2010)
24. Oracle: Oracle Data Guard Concepts and Administration, 11g Release 1 edn (2008)
25. Oracle: Oracle Real Application Clusters 11g Release 2. Oracle (2009)
26. Oracle: MySQL 5.0 Reference Manual. Revision 23486, http://dev.mysql.com/doc/refman/5.0/en/ (2010)
27. Percona Tools TPC-C MySQL Benchmark: https://code.launchpad.net/percona-dev/perconatools/tpcc-mysql (2008)
28. Polyzois, C.A., Garcia-Molina, H.: Evaluation of remote backup algorithms for transaction processing systems. In: International Conference on Management of Data (SIGMOD) (1992)
29. Rajagopalan, S., Cully, B., O'Connor, R., Warfield, A.: SecondSite: disaster tolerance as a service. In: Virtual Execution Environments (VEE) (2012)
30. Scales, D.J., Nelson, M., Venkitachalam, G.: The design and evaluation of a practical system for fault-tolerant virtual machines. Tech. Rep. VMWare-RT-2010-001, VMWare (2010)
31. Soror, A.A., Minhas, U.F., Aboulnaga, A., Salem, K., Kokosielis, P., Kamath, S.: Automatic virtual machine configuration for database workloads. Trans. Database Syst. (TODS) **35**(1), 1–47 (2010)
32. Strom, R., Yemini, S.: Optimistic recovery in distributed systems. Trans. Comput. Syst. (TOCS) **3**(3), 204–226 (1985)
33. TCP/IP Tutorial and Technical Overview: http://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf (2006)
34. Thomas, R.H.: A majority consensus approach to concurrency control for multiple copy databases. Trans. Database Syst. (TODS) **4**(2) (1979)
35. The TPC-C Benchmark: http://www.tpc.org/tpcc/ (1992)
36. The TPC-H Benchmark: http://www.tpc.org/tpch/ (1999)
37. The TPC-W Benchmark: http://www.tpc.org/tpcw/ (1999)
38. Xen Blktap2 Driver: http://wiki.xensource.com/xenwiki/blktap2 (2010)
39. Xu, M., Bodik, R., Hill, M.D.: A "flight data recorder" for enabling full-system multiprocessor deterministic replay. Comput. Archit. News **31**(2), 122–135 (2003)