Fast Memory State Synchronization for Virtualization-based Fault Tolerance

Maohua Lu*# Tzi-cker Chiueh*#

Stony Brook University* Symantec Research Laboratories#
{mlu,chiueh}@cs.sunysb.edu

Abstract

Virtualization provides the possibility of whole machine migration and thus enables a new form of fault tolerance that is completely transparent to applications and operating systems. While initial prototypes show promise, virtualization-based fault-tolerant architecture still experiences substantial performance overhead especially for data-intensive workloads. The main performance challenge of virtualization-based fault tolerance is how to synchronize the memory states of the Master and Slave in a way that minimizes the end-to-end impact on the application performance. This paper describes three optimization techniques for memory state synchronization: fine-grained dirty region identification, speculative state transfer, *and* synchronization traffic reduction using active slave*, and presents a comprehensive performance study of these techniques under three realistic workloads, the TPC-E benchmark, the SPECsfs 2008 CIFS benchmark, and a Microsoft Exchange workload. We show that these three techniques can each reduce the amount of end-of-epoch synchronization traffic by a factor of up to 7, 15 and 5, respectively.*

1. Introduction

The holy grail of fault-tolerant system research is to be able to support seamless fail-over with negligible run-time performance overhead and in a way that is completely transparent to applications and even operating systems. Because of its ability to cleanly package and transport the complete state of any virtual machines, virtualization offers a powerful building block for building transparent and seamless fault-tolerant systems. However, the run-time performance cost of virtualization-based fault tolerance (VFT) is still quite substantial under realistic workloads. The goal of this project is to develop and evaluate optimization techniques that can reduce this performance cost to the level that renders VFT commercially viable.

The first VFT system that we are aware of is XSFT [1], which was built at Symantec Research Labs in 2006 with the first prototype completed in January 2007. As in most fault-tolerant systems, XSFT assumes that there is a Slave server as a back-up for every Master server and a Slave will take over whenever its associated Master fails, and moreover, both Master and Salve servers are physically embodied as virtual machines. The type of servers at which XSFT targets are Internet-facing servers such as Web or DNS servers, which interact with a large number of external user machines as well as other servers. Instead of check-pointing and replaying, XSFT pioneered the concept of *transactional fault tolerance*, which requires that a Master should not send back a response to a request until the memory state updates triggered by the request are reflected in its corresponding Slave. More generally, XSFT holds off outgoing network packets and disk write requests associated with an input request until the memory states of the Master and the Slave are synchronized. By enforcing this invariant, XSFT ensures that a Slave's state is always consistent with its Master as far as their external clients are concerned. Consequently, when a Master fails, its corresponding Slave can immediately pick up where it is left off since the last externalized responses, and the fail-over delay is expected to be negligible.

Unfortunately, the transactional fault tolerance model entails a potentially steep performance cost. Every time a Master receives an input request, it needs to process the request, then propagates the associated memory updates to its Slave, and finally commits the associated network send and disk write actions. Therefore, the delay involved in propagating memory state updates from Master to Slave is added to the round-trip latency perceived by the requesting clients. Increase in the average request latency sometimes could also lead to decrease in throughput as seen by an individual client, especially when the underlying flow control mechanism is based on a stop-and-go model and thus is quite sensitive to the round-trip delay. To more efficiently synchronize the memory states of a Master and its Slave, XSFT aggregates the memory state updates on the Master that occur within an epoch, say 20 msec, and propagates them to the Slave at the end of the epoch. This aggregation is effective because memory state synchronization is typically done on a page-by-page basis and multiple memory updates that occur within the same epoch and within the same page can be propagated with a single page transfer. However, this epoch-based memory state synchronization approach further aggravates the perceived request latency.

Although the first XSFT prototype [1] successfully demonstrated the feasibility of the transactional fault tolerance model, it carried a serious run-time performance overhead, i.e. more than 200% slow-down for realistic data center workloads when the epoch is set to 20 msec. The main performance bottleneck identified is memory state synchronization. In this project, we develop three optimization techniques that are aimed to minimize the performance overhead associated with propagating modified memory pages at the end of every epoch in an XSFT-like system, and we assume there is a dedicated gigabit Ethernet link between a Master and its Slave specifically for memory state synchronization.

The first technique is *fine-grained dirty region tracking*, which keeps track of modifications to a Master's memory state at a granularity smaller than a memory page in order to reduce the total number of bytes required to be transported to its Slave. The second technique is *speculative state transfer*, which transports a Master's dirty regions to its Slave *during* an epoch rather than at the end of an epoch in order to mask some of the performance overhead associated with memory state transfer. The third technique is *synchronization traffic reduction using active slave*, which requires both a Master and its Slave to run concurrently and reduces the number of dirty bytes that need to be transported by virtue of the fact that the memory states of the Master and Slave are likely to be modified in a similar way because they execute the same binary and receive the same inputs.

The rest of this paper is organized as follows. Section 2 reviews previous research on fault-tolerant network services using process state or machine state check-pointing. Section 3 describes the three optimization techniques for memory state synchronization developed in this project. Section 4 presents the results of a trace-driven evaluation of the three proposed techniques and their detailed analysis. Section 5 concludes this paper with a summary of main research results and a brief outline of future work.

2. Related Work

Checkpointing and logging are two widely-used alternatives in modern Fault Tolerance (*FT*) solutions [2], [3]. For the checkpointing approach, states of healthy machines are preserved either locally or remotely. In case of failures, the system is rolled back to the most recent checkpoint. However, events and data updates between the most recent checkpoint and the failure point are lost. In contrast, the logging technique logs the events of healthy machines. Logging events are replayed online at the runtime to ensure identical backup with regard to the primary. For example, Bressoud et al. [3] proposed a hypervisor-based FT where the hypervisor logged each instruction-level operation to the primary and replayed the logged operations on top of the backup. In general, domain-specific knowledge is required to log events for the replaying purpose. With the help of virtualization, the logging can be done within the virtual machine monitor (*VMM*) [2]–[5]. Because logging is done within VMM, no modification is made to the application or operation systems involved, which makes virtualization-based logging and replaying approach appealing.

However, the virtualization-based logging and replaying approach is not widely advocated for two reasons. First, although logging can be easily done with the help of VMM, a deterministic replay of the logging events relies heavily on the target architecture [3]. Therefore, each deterministic replay mechanism requires an implementation on a specific VMM. Second, for multi-core CPUs, the deterministic replay depends on the order in which multiple cores access shared memory. However, to track the order of shared memory access is difficult although some projects [6], [7] suggested promising ad-hoc low-level mechanisms. *Flight data recorder* [6] sniffs cache traffic to infer the order in which shared memory is accessed. *Dunlap* [7] imposed a CREW protocol on share memory pages to track down the access order of the shared memory. Given the high overhead, it is not clear whether the deterministic replay is a feasible alternative of FT solutions [8].

Instead, check-pointing involves less domain-specific knowledge of either hardware or applications. There are two categories of state check-pointing based-on the granularity, per-process state check-pointing and the whole machine state check-pointing, respectively. Similar to the use of process migration in data center and cluster computing environments [2], [9]–[12], operating system virtualization [13] enables the migration of virtualized operating system and therefore can be used for the purpose of load-balancing, operating system isolation and efficient utilization of hardware resources [2], [14], [15].

Virtualization-based machine migration provides an encapsulation for migrating critical services [13], [16]–[18] while avoiding convoluted per-process coordination in migration. Clark [16] proposed a XEN-based live migration mechanism to separate the migration of virtualized operation systems from the external end users. Migration is divided into 5 stages to minimize the negative impact of live migration to both external end users and the physical machine. Services are only stopped at the stop-and-copy stage. However, only the memory states are migrated instead of both the memory and storage. Travostino et al. [17] takes one step further to migrate both the in-memory states and external storage states over a MAN/WAN. Existing commercial virtualization product such as VMotion also supports live migration of virtualized operation system by moving its in-memory states [18].

Virtualization-based fault tolerance has recently attracted considerable interest in recent years [1], [8], [19] for its cost-effectiveness. Memory states of the virtualized operating system is constantly check-pointed to a standby backup instance so that the backup can seamlessly take over when the primary virtualized instance fails. Performance overhead of such systems roots from the copying of memory states. To reduce the amount of transferred memory states, Remus [8] proposed to copy over only those dirtied memory states. The check-pointing frequency for Remus can be as high as every 25 msec. The aim of our work is to further reduce the amount of memory coped over from the primary instance to the backup instance. Remus buffered external events until the end of each synchronization between the primary and the backup virtualized instance. Instead of buffering external events, Kemari [19] checkpoints the primary instance when the VMM is going to duplicate the external events to the backup instance. Our work is orthogonal to the mechanism of ensuring consistency between the primary and backup virtualized instance and can compensate for performance overhead associated with each of such mechanisms.

3. Design Alternatives

3.1. Fine-Grained Dirty Region Tracking

In virtualization-based fault-tolerant systems such as XSFT and Remus, the states of the Master and Slave are synchronized at the end of each epoch. An obvious idea to decrease the performance overhead of this memory state synchronization operation is to identify the regions of the Master's memory modified since the end of the last epoch and ship only these modified regions to the Slave. Both XSFT and Remus exploit virtual memory hardware to detect memory pages that are dirtied within the most recent epoch. Basically, memory pages are marked as read-only at the beginning of an epoch; every time a memory page is modified, a write exception occurs, and the page is recorded in a dirty page list and turned to read-write; at the end of each epoch, pages in the dirty page list are sent to the Slave.

Although the above approach is conceptually straightforward, leveraging virtual memory protection hardware incurs a non-trivial performance overhead, especially when the epoch size is small. For example, imagine the overhead of servicing thousands of write protection faults within an epoch of 20 msec. To address this problem, XSFT [1] incorporates an optimization that exploits the fact that the dirty page lists of consecutive epochs are significantly overlapped with each other. This technique proves to be indispensable for efficient dirty page tracking when the memory state synchronization frequency is high.

Although virtual memory protection hardware is convenient to use and greatly reduces the run-time performance cost of identifying modified pages, it also limits the granularity of dirty memory region tracking to individual pages. This means that even a single byte in a memory page is modified in an epoch, the entire page is considered dirty and needs to be transferred to the Slave at the end of the epoch. We propose a fined-grained dirty region tracking (FDRT) technique to overcome this limitation.

The minimum unit of dirty region tracking in FDRT supports is a tracking block, which is smaller than the typical memory page size (4KB). At the beginning of an epoch, FDRT computes a hash value for each tracking block of every page and stores these hash values in a memory-resident fingerprint database. At the end of the epoch, FDRT computes a hash value for every tracking block of every page in the epoch's dirty page list, compares each such hash value with its counterpart in the fingerprint database, and if they do not match, marks the corresponding tracking block as dirty and replaces the stored fingerprint with the newly computed hash value. In the end, only the dirty tracking blocks are transferred to the Slave.

The smaller the tracking block size is, the more accurately FDRT can approximate the true dirty region, but the larger the in-memory fingerprint database needs to be. Assuming each hash value takes 8 bytes, the memory overhead associated with FDRT's fingerprint database is about 3.1% if the tracking block size is 256 bytes, 6.3% if the tracking block size is 128 bytes and 12.5% if the tracking block size is 64 bytes. One possible optimization to reduce the fingerprint database's memory overhead is to keep hash values not for all memory pages, but only for recently dirtied pages. In this scheme, if the fingerprint database does not have old hash values for a newly dirty page, every tracking block in that page is considered dirty. This technique enables the use of smaller tracking block size while minimizing the fingerprint database's memory overhead.

For each tracking block, FDRT incurs an additional overhead of computing its hash value besides its data transfer cost. On our test machine, the measured throughput of MD5 hash computation is about 320 Mbytes/sec, which is much higher than the sustained throughput of the dedicated Gigabit Ethernet link used for memory state synchronization, which is about 100 Mbytes/sec. Let $DP(T)$ and $DB(T)$ be the number of dirty pages and dirty blocks, respectively, when the epoch size is T. The hash value computation cost is $C_{hash} = \frac{DP(T)}{320 Mbytes/sec}$ and the data transfer cost is $C_{transfer} = \frac{DB(T)}{100 Mbytes/sec}$. To a first approximation, the total delay of an end-of-epoch memory state synchronization transaction takes $MAX(C_{hash}, C_{transfer})$, because hash value computation of a tracking block can be easily pipelined with its transmission.

3.2. Speculative State Transfer

In XSFT, pages dirtied in an epoch are sent from the Master to the Slave at the end of that epoch. One way to decrease the amount of time required to transfer dirty pages is to reduce the total number of bytes that need to be exchanged, and the other way is to speculatively ship some of the dirty pages during an epoch rather than at the end of an epoch. Speculative state transfer (SST) overlaps memory state transport with normal application execution, and thus can potentially mask some or even all of the end-of-epoch memory state synchronization overhead. Because we assume that there is a dedicated network link between the Master and Slave specifically for memory state synchronization, the impact of speculative state transfer on the networking performance of application execution is expected to be small. Moreover, the same idea can be applied to the computation used in memory state synchronization, e.g., computing the per-tracking-block hash values in FDRT during rather than at the end of an epoch.

If a page is sent to the Slave as soon as it is modified, the same page may be sent multiple times during an epoch even though only the last send is necessary. In the worst case, it is possible that SST generates so many redundant page transfers that even the dedicated synchronization network link cannot handle and the overall memory state synchronization delay is actually increased rather than decreased. Therefore, the main technical challenge of SST is how to balance the trade-off between the benefit of overlapping memory state synchronization with application execution and the risk of creating too much unnecessary synchronization traffic.

To strike a good balance between these two considerations, we perform a characterization of the temporal write patterns to memory pages. In particular, we measure each memory page's *write burst length*, which is the temporal distance between the first write and the last write to a memory page during an epoch. A page with a large write burst length means writes to the page are spread over a larger portion of an epoch. Empirically, we found the majority of memory pages have a write burst length of 1 msec or less for an epoch of 20-30 msec, although there is no easy way to identify pages that have a large write burst length. Based on this observation, the current speculative state transfer design uses the following heuristics:

- Schedule a dirty page to be sent to the client one msec after its first write, and mark the page as clean.
- At the end of an epoch, send the dirty pages in the dirty list. These pages correspond to those whose write burst length is larger than 1 msec, or whose write burst length is shorter than 1 msec but whose first write occurs within 1 msec away from the end of an epoch.

The above SST design guarantees each memory page is sent to the Slave at most twice, and is able to mask a significant portion of the memory state synchronization overhead (as shown in Section 4), because the write burst length of most memory pages under the workloads used in our study is smaller than 1 msec.

3.3. Synchronization Traffic Reduction Using Active Slave

In both XSFT and Remus, the Slave is passive in the sense that the Slave is not actively running but is ready to go when the Master dies. In the passive Slave design, a Slave never consumes any CPU resource of the physical machine on which it resides. Therefore, a single physical machine can host multiple passive Slaves simultaneously, i.e., N+1 fault tolerance rather than 1+1 fault tolerance.

An alternative to the passive Slave design is the active Slave design [20], in which the Slave runs concurrently with the Master, which is the only entity that directly interacts with external machines. To ensure that the Master and Salve receive the same network inputs, every input packet that the Master receives is duplicated, transformed and sent to the Slave. Proper transformation on the input packets is required because some sequence number fields in network protocols are randomly initialized, e.g., TCP. To ensure that the Slave is able to seamlessly continue the Master's interactions with external machines after the Master dies and it takes over, the Master returns a response to an external machine only after it receives the corresponding response from the Slave.

Because Master and Slave run the same code, receive the same network inputs and are frequently synchronized with respect to their responses to incoming requests, their memory states should be largely the same at the end of an epoch and accordingly the amount of data to be transferred for memory state synchronization is expected to be small. Because there is still non-determinism in the system, e.g., timer interrupts, the memory states of the Master and Slave are not identical to each other. The third optimization technique to decrease the memory state synchronization overhead is to run the Slave in the active mode.

In the active Slave configuration, at the end of each epoch, the Slave sends its dirty page list, which includes the page number and hash value of each dirty page, to the Master, which compares the received dirty page list with its own dirty page list and categorize the Slave's pages into the following 4 types: (A) clean pages that are also clean in the Master, (B) dirty pages whose new contents are available on the Slave, (C) dirty pages whose new contents must come from the Master, and (D) clean pages whose new contents must come from the Master because they are in the Master's dirty page list. The Master only needs to transfer pages that are of Type (C) and (D). Because the Master and Slave share many common dirty pages whose page number in the Master is different from that in the Slave, we decide to include per-page hash values into the dirty page list so as to classify such dirty pages into Type (B) pages rather than Type (C).

Even though the active Slave configuration could potentially cut down the memory state synchronization overhead, it incurs an implicit performance cost in addition to consuming more hardware resources: The response to every incoming request can be returned to the requesting client only when the slower of the Master and Slave successfully processes the request and responds. This performance cost could become quite visible if the Master and Slave each share a physical machine with other virtual machines and the hypervisors on the Master's and the Slave's machines are not coordinated.

3.4. Put Together

The above three optimization techniques contribute to the reduction of the memory state synchronization traffic in a way that is largely orthogonal to one another, and thus can be combined together to minimize the overall memory state synchronization overhead at the end of each epoch.

4. Comparative Evaluation

4.1. Methodology

We used an emulation approach to evaluate the effectiveness of the proposed three optimization techniques and assess how their performance is affected by relevant system configuration parameters. More concretely, we ran the three benchmarks on a test-bed consisting of two physical machines, one of them running one or multiple client virtual machines and the other running one or multiple server virtual machines. The hypervisor used in this study is Xen, and the server and client VMs ran either Linux or Windows, depending on the benchmark. The client machine is 3.20-GHz Pentium IV machine with 1 GB memory and a 5400 RPM and 160-GB SATA disk. The server machine is 2.66-GHz Pentium IV machine with 2 GB memory and a 7200 RPM and 250-GB SATA disk. The three benchmarks used in this study are as below:

- TPC-E Workload
  TPC-E [21] is a newly-introduced OLTP benchmark that simulates the OLTP workload of a brokerage firm. DBT-5 [22] is an open-source TPC-E implementation using PostgreSQL as the backend DBMS. DBT-5 initializes the brokerage database with 5,000 customers, the scale factor to 500 and the number of initial trade days to 200, and runs the TPC-E transactions for 1 minute for the active slave technique, and 1 hour for the other two optimization techniques. The TPC-E workload is a largely random workload with very poor data locality. The average disk read/write size is 8 KB, with 43% of the disk I/O requests being writes and the rest being reads.
- CIFS Workload
  The CIFS benchmark in SPECsfs 2008 [23] is a synthetic workload simulating the typical load on production-mode Windows file servers. In this workload, there are 10 concurrent client processes and 1 server, and the number of sustained CIFS operation increases from 10 to 100 with 10 as the increment. Each experiment run lasts for one minute under the active slave scenario but lasts for 10 minutes for the other two optimization techniques. The average disk I/O request size for reads and writes in the resulting trace is 4 KB. 25% of the disk I/O requests are writes and the remaining are reads.
- Exchange Workload
  The Exchange workload runs a Windows Exchange 2003 server with a load generator called LoadGen [24] developed by Microsoft Corporation. LoadGen simulates the workload of a medium-sized corporation's email server. The load generator runs for 1 minute with 1,000 email accounts and 1 user group. The average number of sustained tasks in each email is 132. The average disk I/O request size for reads and writes is 16 KB and 4 KB, respectively. 99% of the disk I/O requests are write requests and the remaining are read requests.

To evaluate the effectiveness of *FDRT*, we ran the Master server in a VM, marked all its memory pages as read-only, implemented a write exception handler in the Xen hypervisor to maintain a dirty page list, chose a tracking block size and computed per-tracking-block hash values for the Master VM's memory, stopped the test-bed at the end of each epoch, identified the tracking blocks that are modified during each epoch based on the dirty page list and per-tracking-block hash values, and compute the total number of bytes that need to be transferred at the end of each epoch for the chosen tracking block size. This set-up is referred to as the FDRT configuration hereafter.

To evaluate the effectiveness of *SST*, we used the FDRT configuration with the following modification: at the end of every msec during every epoch, the dirty page list is appended to an in-memory log and then cleared, and all the memory pages are marked as read-only. At the end of an epoch, we calculated the write burst length (at the granularity of msec) of every page modified in the epoch, and computed the number of dirty pages that need to be transferred to the Slave at the end of the epoch with varied eagerness delay.

To compare the difference in memory states between the active and passive slave configurations, we set up a Master VM and a Slave VM on the server machine, and modified the network driver in DOM0 to duplicate every incoming packet destined to the Master VM with proper TCP/CIFS sequence number rewriting and forward it to the Slave VM, and to defer an outgoing packet from the Master after receiving the corresponding packet from the Slave. In addition, at the end of every epoch, we stopped the Master and Slave VMs, computed the difference in the memory states of the Master and Slave VMs based on their per-page hash values, modified Xen's VM suspend/resume mechanism to copy the Master's VM state to the Slave over the network, and proceeded to the next epoch.

In all the experiments, we stopped the client VMs at the end of each epoch as well, so that the end-of-epoch processing delay due to statistics collection and bookkeeping is not visible to the clients, and therefore won't affect the input workloads.

4.2. Fine-Grained Dirty Region Identification

Figure 1(a) shows the average number of bytes transferred at the end of each epoch during the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the tracking block size of fine-grained dirty region tracking is varied from 64 bytes, 256 bytes and 1024 bytes to 4096 bytes. Let's call a contiguous range of bytes that are modified within an epoch a *dirty region* and divide each tracking block into 4 subblocks. When increasing the

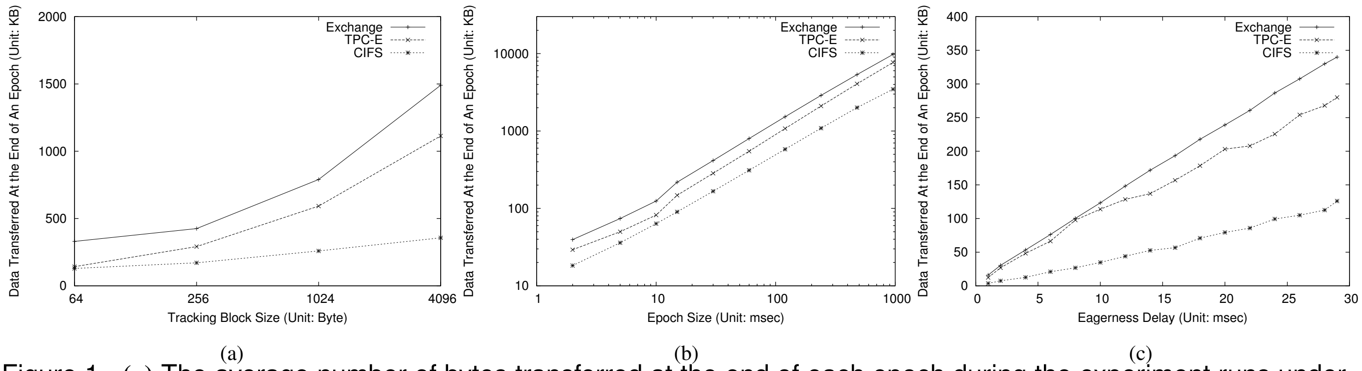


Figure 1. **(a)** The average number of bytes transferred at the end of each epoch during the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the tracking block size of fine-grained dirty region tracking is varied from 64 bytes, 256 bytes and 1024 bytes to 4096 bytes. The epoch size is fixed as 30 msec. **(b)** The average number of bytes transferred at the end of each epoch during the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the epoch size of fine-grained dirty region tracking is increased from 2 msec to 1 second. The tracking block size is fixed as 256 bytes. **(c)** The average number of bytes transferred at the end of each epoch during the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the eagerness delay of speculative state transfer is varied from 1 msec to 29 msec. The epoch size is fixed as 30 msec and the tracking block size is 256 bytes.

tracking block size from 64 bytes to 4096 bytes, the total amount of end-of-epoch synchronization traffic is increased by a factor of 4.5, 7.8, and 2.7 for the Exchange workload, the TPC-E workload and the CIFS workload, respectively.

In general, the slope in the increase of the amount of end-of-epoch synchronization traffic decreases with the increasing tracking block size. For the CIFS workload, the majority of the dirty regions are close to 4096 bytes, as a result using a smaller tracking block size than 4096 bytes does not provide as much reduction in the amount of end-of-epoch synchronization traffic as the other two workloads. For the Exchange and TPC-E workload, the distribution of the dirty region size follows a bimodal distribution that concentrates on 64-byte and 4096-byte regions. As a result, the percentage of subblocks in each tracking block that are dirty increases with the tracking block size. For example, if the average dirty region size is 64 bytes, and the average distance between adjacent dirty regions is 256 bytes, then the amount of end-of-epoch synchronization traffic for the 64-byte tracking block size would be four times smaller than that for the 256-byte tracking block size, which in turn would be the same as that for the 1024-byte or 4096-byte tracking block size. However, if the average dirty region size is 256 bytes, and the average distance between adjacent dirty regions is 2048 bytes, then the amount of end-of-epoch synchronization traffic for the 64-byte tracking block size would be the same as that for the 256-byte tracking block size, which in turn would be four times smaller than that for the 1024-byte tracking block size, which in turn is half as that for the 4096-byte tracking block size.

As expected, figure 1(b) shows that the amount of end-of-epoch synchronization traffic increases with the epoch size, but the slope of increase is smaller than linear and decreases with the epoch size because the same tracking block is likely to receive more updates during an epoch as the epoch size increases. For all three workloads, the amount of end-to-epoch synchronization traffic is less than 100 Kbytes or costs less than 1 msec assuming a 100 Mbytes/sec dedicated Gigabit Ethernet link, when the epoch size is 10 msec. This means that FDRT alone can already enable the use of an epoch size as small as 10 msec because the end-of-epoch synchronization delay is smaller than 10% of such an epoch.

4.3. Speculative State Transfer

Figure 1(c) shows the average number of bytes transferred at the end of each epoch during the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the eagerness delay of speculative state transfer is varied from 1 msec to 29 msec. Assuming the eagerness delay is D msec, under speculative state transfer, the pages that are transferred at the end of an epoch are those pages whose write burst length is larger than D msec, or those pages whose first write occurs within the last D msec of an epoch. It turns out that for the Exchange and CIFS workload, the write burst length of more than 90% of the pages dirtied in an epoch is less than 1 msec, whereas for the TPC-E workload, more than 82% of the pages dirtied in an epoch have a write burst length of less than 1 msec. Therefore, for all three workloads, most of the pages that are transferred at the end of an epoch are pages whose first write occurs within the last D msec of an epoch. As D increases, the number of such pages is increased, and so does the amount of end-of-epoch synchronization traffic.

When the eagerness delay is increased from 1 msec to 29 msec, the amount of end-of-epoch synchronization traffic is increased by a factor of 10.8, 9.9, and 15.1 for the Exchange, TPC-E and CIFS workload, respectively. Because

Workload	**Percentage of Dirty States (Unit: %)**		
	Master-Only	Common	Slave-Only
TPC-E	8	83	9
Exchange	9	79	12
CIFS	17	65	18

Table 1. *The average percentage of different types of dirty pages in a Master operating in the active Slave configuration under the three workloads. The epoch size is 15 msec.*

the percentage of dirty pages whose write burst length is less than 1 msec is smaller in the TPC-E workload than the other two workloads, the impact of pages whose first write occurs within the last D msec of each epoch on its curve is less and consequently its curve is less smooth than the other the curves associated with the other two workloads.

Although speculative state transfer could mask some of the memory state synchronization delay, it does this at the expense of transferring more data than absolutely necessary. Fortunately, Figure 2(a) demonstrates that the total number of bytes transferred during and at the end of an epoch when speculative state transfer is turned on is 1.09, 1.08, and 1.22 times that when speculative state transfer is disabled under the Exchange workload, the TPC-E workload and the CIFS workload, respectively. This results suggests the amount of unnecessary data transfer in speculative state transfer is kept to an acceptable level in practice. Moreover, in all three workloads, the during-an-epoch data transfer rate required is well below and thus can be easily accommodated by the sustained rate of the dedicated Gigabit Ethernet link, i.e., 100 Mbytes/sec or 3.3 Mbytes per 30 msec.

Surprisingly, Figure 2(b) shows that the amount of end-of-epoch synchronization traffic remains largely constant regardless of the epoch size. This result suggests that speculative state transfer not only can shift most of the memory state synchronization operations from at the end of an epoch to during an epoch, it is capable of performing this shifting equally effectively across all epoch sizes, because updates to the memory state are rarely concentrated towards the end of an epoch, no matter what the epoch size is.

Figure 2(c) shows the overhead in terms of total numbers of bytes during a whole epoch. Smaller eagerness delay leads to larger overhead because pages dirtied multiple times within an epoch are not clusterred in a narrow range and smaller eagerness delay transfer more of those dirty pages. The Exchange workload has a larger ratio of pages dirtied multiple times within an epoch than the TPC-E workload and the CIFS workload, while that of the TPC-E workload is larger than that of the CIFS workload.

4.4. Synchronization Traffic Reduction Using Active Slave

When the Slave is active, at the end of each epoch, the pages at the Master can be classified into 4 types: A, B, C, and D, as described in Section 4.4. Type C and D pages need to be transferred from the Master to the Slave, and Type B pages need to be transferred if the Slave is passive. Therefore we use the ratio $\frac{NB}{NB+NC+ND}$ as a metric to evaluate the effectiveness of the Active Slave configuration in reducing the end-of-epoch synchronization traffic, where NB, NC and ND represent the number of Type B, Type C and Type D pages, respectively. Figure 2(d) shows how this ratio varies with the epoch size under the three workloads. As expected, the larger the epoch size, the more divergent the memory states of the Master and Slave are and the smaller the reduction in end-of-epoch synchronization traffic. When the epoch size is 15 msec, the ratio is above 83% for the TPC-E workload, 79% for the Exchange workload, and 65% for the CIFS workload, as shown in Table 1. This means the Active Salve configuration can reduce the end-of-epoch synchronization traffic by a factor of 5 for the TPC-E and Exchange workload but only a factor of 2.8 for the CIFS workload. When the epoch size is 30 msec, the synchronization traffic reduction factor is decreased to 3.6, 3.1 and 1.6 for the TPC-E, Exchange and CIFS workload, respectively.

Because the Master and Slave run the same program and are fed the same network inputs, the difference between their memory states mainly comes from non-determinism in the code, for example, TCP's sequence number that is initialized randomly. In the case of the CIFS workload, the CIFS protocol also has its own sequence number that appears to be initialized randomly. Therefore, the Master and Slave are not fed with exactly the same input packets. This additional non-determinism explains why the Active Slave configuration is less effective at reducing the end-of-epoch synchronization traffic for the CIFS workload than for the other two workloads.

Figure 2(e) shows analytical curves illustrating how the number of bytes transferred at the end of each epoch varies with the epoch size. The analytical curves multiply the total bytes of dirty data blocks with the ratio of Type-B dirty blocks. As expected, the number of bytes transferred at the end of each epoch increases with the epoch size.

4.5. Analysis of Performance Overhead

The cornerstone of all optimization techniques to reduce the volume of the dirty data blocks at the end of each epoch is the mechanism to track the dirtiness of a data block. There exists two ways to track the dirtiness of data blocks: to scan the dirty bitmap of the whole memory space of the guest virtual machine, or to mark the target pages as write-protected to trigger an exception whenever the target page is modified. It is time-consuming to use the dirty bitmap alone, and it is also expensive to use the write-protection mechanism (0.164 msec to process a write-protection exception under XEN). To mitigate the overhead associated with both of these two techniques, we developed a mechanism to combine these two techniques together as follows. At the beginning of each epoch, all pages that are previously dirtied (denoted as Old-Dirty-Region) do not have the write-protection, and all other pages have the write-

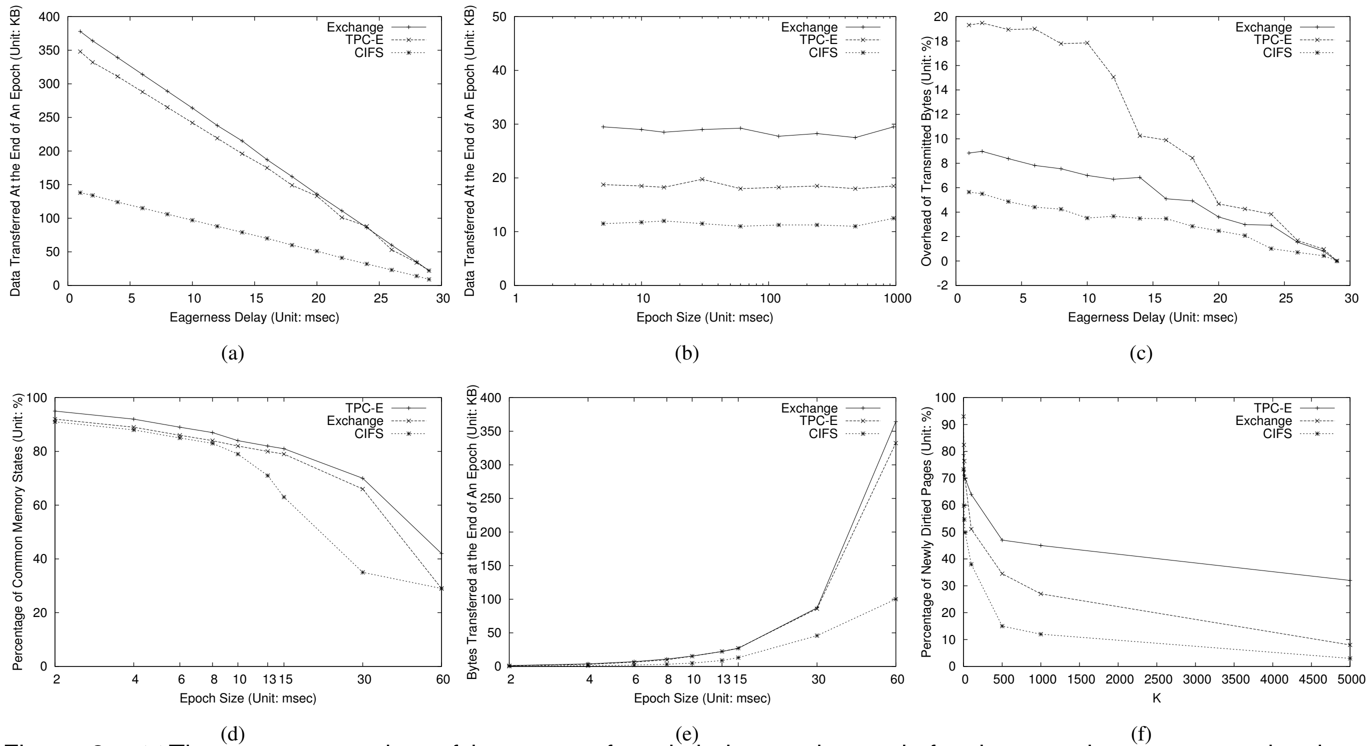
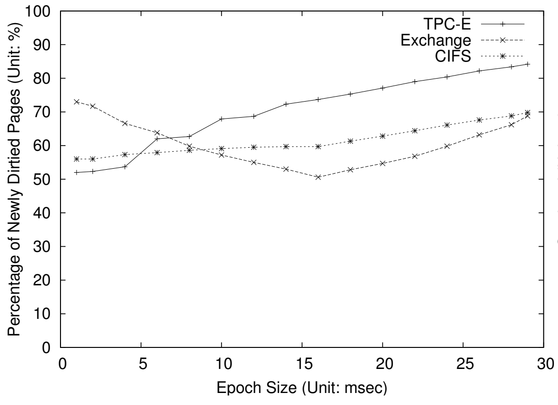


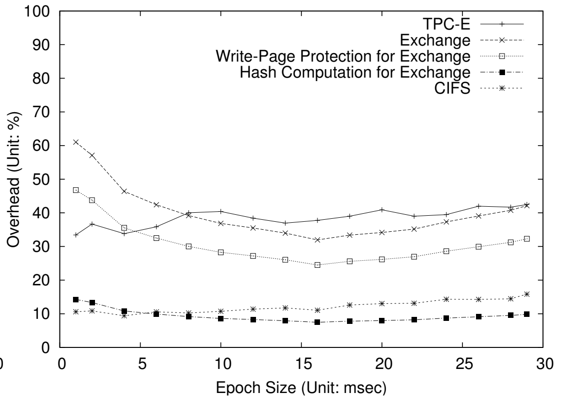
Figure 2. **(a)**The average number of bytes transferred during each epoch for the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the eagerness delay of speculative state transfer is varied from 1 msec to 29 msec. The epoch size is fixed as 30 msec and the tracking block size is 256 bytes. **(b)** The average number of bytes transferred at the end of each epoch during the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the epoch size is varied from 5 msec to 1 second. The eagerness delay of speculative state transfer is fixed as 2 msec and the tracking block size is 256 bytes. **(c)**The overhead in terms of bytes transferred during the whole epoch for the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the eagerness delay is varied from 1 msec to 29 msec. The epoch size is fixed as 30 msec and the tracking block size is 256 bytes. **(d)**The percentage of common memory states of all dirty memory blocks for both the master machine and the slave machine with varied epoch size under the Exchange workload, the TPC-E workload and the CIFS workload. The tracking block is 256 byte, and the X axis is in log scale. **(e)**The average number of bytes transferred at the end of each epoch for the experiment runs under the Exchange workload, the TPC-E workload and the CIFS workload, when the epoch size using Active Slave is varied from 1 msec to 60 msec. The tracking block size is 256 bytes. **(f)**The ratio of newly dirty data blocks with regard to the previous k epochs under the three workloads when k is varied from 1 to 5,000. The epoch size is 2 msec.

protection mechanism on. At the end of each epoch, scan the portion of dirty bitmap corresponding to Old-Dirty-Region to find out the dirtied pages in the Old-Dirty-Region, and rely on the write-protection mechanism to track the dirty pages out of Old-Dirty-Region. If the Old-Dirty-Region contains dirty pages in the previous k epochs, we denote it as k-Old-Dirty-Region. If the k-Old-Dirty-Region does not change significantly across epochs, we can greatly reduce the overhead associated with the write-page protection mechanism. Unfortunately, for all three workloads, it takes larger k to retrieve a k-Old-Dirty-Region that has more common pages with the current epoch. Figure 2(f) shows that the ratio of newly dirty regions decreases as k increases as expected. However, the ratio does not fall behind 30% for the TPC-E workload even when $k = 5,000$. It is surprising at first but scrutinizing of memory traces show that the average modification interval of the 30% newly dirtied regions is more than 10 seconds for the TPC-E workload, which indicates those pages are dirtied by different processes in the system and exhibits poor data locality.

Figure 3(a) shows that the ratio of newly dirty blocks across epochs does not follow a fixed trend with varied epoch sizes. For the TPC-E and CIFS workload, the ratio keeps increasing with the epoch size because larger epoch introduces more new dirty blocks. In contrast, for the Exchange workload, the distribution of dirty blocks is more bursty and the average number of new dirty blocks does not increase when the epoch increases from 1 msec to 16 msec.



(a)



(b)

Figure 3. (**a**)The ratio of newly dirty data blocks with regard to the previous 10 epochs under the three workloads when the epoch is varied from 1 msec to 29 msec. (**b**)The runtime performance overhead with the epoch size varying from 1 msec to 29 msec when FDRT, SpeccSS and Active Slave are combined for the Exchange, TPC-E and CIFS workload. The eagerness delay is fixed as 1 msec, the tracking block is 256 bytes, and $k = 10$.

When the epoch size is larger than 16 msec, the average number of new dirty blocks increases for the Exchange workload and the same for the ratio of newly dirty blocks.

Besides the overhead due to page write-protection, the computation of MD5 hash values of each data block contributes the other significant run-time overhead. It takes on average 0.05 msec to compute the hash value of a whole 4KB page. As the computation of the MD5 hash value is proportional to the size of the data block, we assume it takes 0.05 msec to compute hash values of all data blocks within a 4KB page regardless the size of the data block. The overhead of hash computation is proportional to the number of 4 KB pages because for each 4 KB page, we have to compute the hash values for all of its data blocks to figure out the newly dirtied data blocks in the current epoch.

The performance overhead consists of three components: 1) the overhead due to write-protection page exception, 2) the computation of hash values, and 3) the transmission of dirty data blocks. As the computation and the transmission can be pipelined efficiently and the transmission has a larger throughput (100 MB/s compared with 80 MB/s for the computation of hash values), we only model the overhead due to 1) and 2). Figure 3(b) illusrates the overhead of all three workloads in terms of elapsed time with varied epoch sizes. The overhead is largely decided by the number of dirty 4KB pages and the curves follow similar trends as those in figure 3(a). Another observation is that the overhead does not necessarily decrease as the epoch size increases. For the Exchange workload, the overhead is decomposed into two components corresponding to 1) and 2), respectively. For the overhead due to 1) and 2), the overhead due to 1) dominates and attributes to 76% of the overhead.

5. Conclusion and Future Work

The key design decision in virtualization-based fault-tolerant systems such as XSFT and Remus is an epoch-based client server model, in which incoming requests received within an epoch are processed by the Master in a batch, and their associated external operations, in particular, disk write accesses and network packets transmission, are deferred until the memory states of the Master and Slave are synchronized at the end of each epoch. In general, the larger the epoch, the lower the memory state synchronization overhead, and the higher the average request latency as perceived by external clients. Moreover, for network applications whose throughput is sensitive to the round-trip delay, larger epoch size also results in lower perceived throughput. Therefore, how to reduce the epoch size while minimizing the memory state synchronization overhead is a key technical challenge for these high-availability systems. This paper describes the design of three optimization techniques that are devised to reduce the memory state synchronization overhead in three separate orthogonal ways, and presents a comprehensive evaluation of them under three data-intensive server benchmarks. More concretely, the contributions of this research to fault tolerance research include

- A characterization study of the network traffic requirement of Master-Slave memory state synchronization in virtualization-based fault-tolerant systems under data-intensive server benchmarks.
- Development of three optimization techniques to minimize the amount of synchronization traffic at the end of every epoch, including fine-grained dirty region tracking, speculative state transfer, and synchronization traffic reduction using active Slave.
- A comprehensive evaluation of the proposed three optimization techniques under representative enterprise server workloads, including their effectiveness individually.

Both XSFT and Remus require modifications to and thus are tied with the underlying hypervisor, Xen in both cases.

Some hypervisors are starting to expose a programming API for one virtual machine to take control when certain events in another virtual machine occur, and to access memory pages or intercept network or disk I/O operations associated with another virtual machine, e.g., the VMsafe [25] API from VMware. These APIs offer an opportunity to extend a hypervisor without introducing third-party code into the hypervisor. We plan to leverage this kind of programming APIs to implement the proposed optimization techniques in a way that is largely portable across different hypervisors. When the Slave in a virtualization-based fault-tolerant system is passive, a single physical machine can host multiple Slaves for multiple Masters, each of which runs on a separate physical machine. However, to prevent the memory state synchronization transactions associated with multiple Master-Slave pair from colliding with one another, the epochs of the Master-Slave pairs serviced by a Slave-hosting machine should be staggered so that the machine can service a different pair at a different time slot. How to coordinate the epoch timing of these Master-Slave pairs without modifying the underlying hypervisor is a non-trivial technical challenge.

References

[1] Dharmesh Shah and Sameer Lokray, “XSFT: Next Generation HA,” Technical Report, Symantec Research Labs, 2008, http://engweb.ges.symantec.com/cuttingedge/2007/presentations/SHAH1003_rev0.ppt.

[2] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott, “Proactive Fault Tolerance for HPC with Xen Virtualization,” in *ICS ’07: Proceedings of the 21st annual international conference on Supercomputing*, New York, NY, USA, 2007, pp. 23–32, ACM.

[3] T. C. Bressoud and F. B. Schneider, “Hypervisor-based Fault Tolerance,” in *SOSP ’95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, New York, NY, USA, 1995, pp. 1–11, ACM.

[4] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen, “ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 211–224, 2002.

[5] Samuel T. King, George W. Dunlap, and Peter M. Chen, “Debugging Operating Systems with Time-Traveling Virtual Machines,” in *ATEC ’05: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005, pp. 1–1, USENIX Association.

[6] Min Xu, Rastislav Bodik, and Mark D. Hill, “A ”Flight Data Recorder” for Enabling Full-System Multiprocessor Deterministic Replay,” *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 122–135, 2003.

[7] III George Washington Dunlap, *Execution replay for intrusion analysis*, Ph.D. thesis, Ann Arbor, MI, USA, 2006.

[8] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield, “Remus: High Availability Via Asynchronous Virtual Machine Replication,” in *NSDI’08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2008, pp. 161–174, USENIX Association.

[9] J. Duell, “The Design and Implementation of Berkeley Lab’s Linux Checkpoint/Restart,” Technical Report, Lawrence Berkeley National Laboratory, 2000.

[10] Michael L. Powell and Barton P. Miller, “Process Migration in DEMOS/MP,” *SIGOPS Oper. Syst. Rev.*, vol. 17, no. 5, pp. 110–119, 1983.

[11] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Vishal Sahay, and Andrew Lumsdaine, “The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing,” *International Journal of High Performance Computing Applications*, vol. 19, no. 4, 2005.

[12] Graham E. Fagg and Jack Dongarra, “FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World,” in *Proceedings of the 7th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, London, UK, 2000, pp. 346–353.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, “Xen and the Art of Virtualization,” in *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, NY, USA, 2003, pp. 164–177, ACM.

[14] Carl A. Waldspurger, “Memory Resource Management in VMware ESX server,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[15] Yang Yu, Fanglu Guo, Susanta Nanda, Lap chung Lam, and Tzi cker Chiueh, “A Feather-Weight Virtual Machine for Windows Applications,” in *VEE ’06: Proceedings of the 2nd international conference on Virtual execution environments*, New York, NY, USA, 2006, pp. 24–34, ACM.

[16] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield, “Live Migration of Virtual Machines,” in *NSDI’05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, Berkeley, CA, USA, 2005, pp. 273–286, USENIX Association.

[17] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg, “Live Wide-Area Migration of Virtual Machines including Local Persistent State,” in *VEE ’07: Proceedings of the 3rd international conference on Virtual execution environments*, New York, 2007, pp. 169–179, ACM.

[18] VMWare Inc, “VMWare VirtualCenter Version 1.2 User’s Manual,” 2004.

[19] Yoshi Tamura, “Kemari: Virtual Machine Synchronization for Fault Tolerance using DomT,” Technical Report, NTT Cyber Space Labs, 2008, www.getxen.org/files/xensummitboston08/tamura_xen_summit_presentation_final.pdf.

[20] Srikant Sharma, Jiawu Chen, Wei Li, Kartik Gopalan, and Tzi cker Chiueh, “Duplex: A Reusable Fault Tolerance Extension Framework for Network Access Devices,” in *In Proceedings of 2003 International Conference on Dependable Systems and Networks (DSN*, 2003.

[21] Transaction Processing Performance Council, “TPC Benchmark E,” http://www.tpc.org/tpce/tpc-e.asp, 2006.

[22] Mark Wong and Rilson Nascimento, “Digesting an Open-Source Fair-Use TPC-E Implementation: DBT-5,” http://www.pgcon.org/2007/schedule/attachments/35-TPC-Ek-Wong-Rilson-Nascimento.pdf, 2007.

[23] Standard Performance Evaluation Corporation, “SPEC SFS (System File Server) Benchmark,” http://www.spec.org/osg/sfs97/, 1997.

[24] Microsoft Corporation, “Microsoft Exchange Load Generator,” http://www.msexchange.org/articles/Microsoft-Exchange-Load-Generator.html, Jan, 2007.

[25] VMWare Inc, “VMware VMsafe Security Technology,” http://www.vmware.com/technology/security/vmsafe.html, 2008.