Data Replication Strategies for Fault Tolerance and Availability on Commodity Clusters

Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel
Department of Computer Science
Rice University
{amza, alc, willy}@cs.rice.edu

Abstract

Recent work has shown the advantages of using persistent memory for transaction processing. In particular, the Vista transaction system uses recoverable memory to avoid disk I/O, thus improving performance by several orders of magnitude. In such a system, however, the data is safe when a node fails, but unavailable until it recovers, because the data is kept in only one memory.

In contrast, our work uses data replication to provide both reliability and data availability while still maintaining very high transaction throughput. We investigate four possible designs for a primary-backup system, using a cluster of commodity servers connected by a write-through capable system area network (SAN). We show that logging approaches outperform mirroring approaches, even when communicating more data, because of their better locality. Finally, we show that the best logging approach also scales well to small shared-memory multiprocessors.

1 Introduction

We address the problem of building a reliable transaction server using a cluster of commodity computers, i.e., standard servers and system area networks (SAN). We use the Vista system as the transaction server [5]. Vista is a very high-performance transaction system. It relies on recoverable memory [2] to avoid disk I/O, thereby achieving its very high throughput. Because Vista does not store its data on disk, but rather keeps it in reliable memory, the data remains safe when the machine fails, but it is unavailable until the machine recovers.

In contrast, our work uses data replication to provide both reliability and data availability. We consider a primary-backup solution in a cluster of computers, in which a primary normally executes the transactions. The data is replicated on the backup, which takes over when the primary fails. We focus on the problem of maintaining good transaction throughput in spite of having to update the backup. We do not address other cluster issues such as crash detection and group view management, for which well-known solutions are available [12].

Recent work has suggested that such clusters can be built in a fairly transparent manner, taking a high-performance single-processor transaction system and simply extending it to mirror its execution or its data on the backup machine using write through [15]. Our experience is different: we found that with current machines and SANs, the performance of such a straightforward implementation is disappointing. We attribute this different outcome to the fact that, in our environment, the processor speed is much higher than in the experiments reported in Zhou et al. [15], while the network speed is approximately the same. Based on this observation, we investigate ways to restructure the single-processor transaction system to achieve better performance. We develop and compare four protocols to communicate the modifications made by the transactions from the primary to the backup.

From an architectural viewpoint, we also investigate whether there is any gain to be had from actively involving the processor on the backup machine during the normal operation (i.e., when the primary is functioning and processing transactions), or whether it suffices for the backup to be passive and simply function as a mirror site. This issue has implications for the extent to which the backup can or should be used to execute transactions itself, in a more full-fledged cluster, not restricted to a simple primary-backup configuration.

In our experiments, we use a 600Mhz Compaq Alpha 21164A (EV5.6) processor as both the primary and the backup, and a second-generation Memory Channel as the SAN. The Memory Channel is an instance of a network with "write through" capability, i.e., a memory region on one node can be mapped to a memory region of another node, and writes by the first node to that memory region are written through to the second node. Such capabilities are also available in the VIA standard [3]. We believe the chosen environment — processor, network, and transaction processing system — reflects state-of-the-art hardware and software.

Our conclusion is that log-based approaches with a log designed for good locality of access lead to the highest transaction throughput. The advantages of spatial locality of access in the logging approaches are two-fold. First, locality in the memory accesses, means better cache utilization on the primary. Second, locality in the I/O space accesses, offers better opportunities for data aggregation and thus optimal bandwidth utilization on the SAN between the primary and backup.

In the environment we are using, the difference between an active and a passive backup is moderate (14% to 29%). It is essential, however, that the log be designed with locality and reduced communication to the backup in mind, because of bandwidth limitations in current SANs, relative to current processor speeds. This is especially true when executing multiple transaction streams on a multiprocessor primary as this puts increased stress on the SAN.

We have also developed mirroring approaches, which have the lowest data communication requirements among the passive backup versions, but their performance is inferior to logging, because of their lesser locality.

These results reflect commonly held wisdom in disk-based transaction processing systems, where sequential disk access resulting from using logs leads to better performance than mirroring approaches [4]. Somewhat surprisingly, this wisdom appears to apply to memory-based systems as well, because of the importance of locality and the limited bandwidth of SANs relative to processor speeds and memory bandwidths.

The remainder of this paper is structured as follows. Section 2 provides the necessary background for our work. Section 3 evaluates the straightforward implementation of Vista on our platform. Section 4 describes how we restructure Vista for improved standalone performance. In Section 5 we report the performance of these restructured versions using a passive backup. A comparison to using an active backup is presented in Section 6. We discuss scaling to larger databases in Section 7. We investigate how the use of a small shared-memory multiprocessor as a primary affects the results in Section 8. Section 9 discusses related work. Section 10 concludes the paper.

2 Background

This section describes the transaction API, the implementation of that API in (single-node) Vista, the relevant characteristics and performance of the hardware on which we did the experiments, and the benchmarks used.

2.1 API

We support a very simple API, first introduced by RVM [8] and later implemented by a variety of systems, including Vista [5]. The transaction data is mapped into the virtual address space of the server. The API contains the following routines:

```
begin_transaction()
set_range()
commit_transaction()
abort_transaction()
```

`begin_transaction`, `commit_transaction`, and `abort_transaction` implement the customary transaction semantics [4]. The `set_range` operation takes as its argument a contiguous region of virtual memory. It indicates to the system that the transaction may modify some or all of the data in this region.

As in Zhou et al. [15], our primary-backup implements a 1-safe commit mechanism [4]. In a 1-safe design, the primary returns successfully from a commit as soon as the commit is complete on the primary. It does not wait for the commit flag to be written through to the backup. This leaves a very short window of vulnerability (a few microseconds) during which a failure may cause the loss of a committed transaction.

The API contains no provisions for expressing concurrency control. It is assumed that concurrency control is implemented by a separate layer of software.

2.2 Vista

We use Vista because it is, to the best of our knowledge, the fastest open-source transaction system available. As such, it provides a good "stress test" for a cluster-based server.

Vista achieves high performance by avoiding disk I/O. Instead, it relies on the Rio reliable memory system [2] to achieve persistence. Rio protects main memory against its two common causes of failure, namely power failures and operating system crashes. An un-interruptible power supply guards against power failures. Guarding against operating system crashes is done by protecting the memory during a crash and restoring it during reboot. Extensive tests have shown that very high levels of reliability can be achieved by these methods (see Chen et al. for detailed measurement results [2]).

Besides avoiding disk I/O, the presence of a reliable memory underneath allows considerable simplification in the implementation of transaction semantics [5]. Vista stores the database proper and all of its data structures, including an undo log, in Rio reliable memory. On a `set_range` it copies the current contents of the specified region to the undo log. Modifications to the database are then made in-place. In the case of an abort or a crash before the transaction completes, the data from the undo log is re-installed in the database during abort handling or during recovery. In the case of a commit, the undo log entry is simply deleted. No redo log is necessary, because the updates are already made in-place.

2.3 Hardware Environment

Compaq's Memory Channel network enables a processor within one machine to write directly into the physical memory of another machine without software intervention. In reality, a processor writes to an I/O space address that is backed by its local Memory Channel interface. That interface transmits the written value to the remote machine's interface, which then performs a DMA operation to deposit the value into physical memory. Thus, the remote processor is not involved in the transfer, but the normal cache coherence mechanism ensures that it sees the new value promptly. The kernel (software) and the remote processor are only involved at initialization time, when a mapping is created between the I/O space on the sending machine and the physical memory on the receiving one.

Only remote writes are supported; remote reads are not. This asymmetry gives rise to the double mapping of shared data: one (I/O space) mapping is used to write the data and another (ordinary) mapping is used to read the local copy of the data in physical memory. When the Memory Channel interface is configured in "loopback" mode, it applies any changes to the local copy in addition to transmitting them over the network. Loopback is not, however, instantaneous. There is a substantial delay between the write to I/O space and the change appearing in the local copy. This presents a problem, because a processor may not see its own last written value on a subsequent read. Consequently, the most practical method for implementing shared data is to disable loopback and perform "write doubling". In other words, the same write is performed on the local copy and the I/O space.

In our experiments, we use AlphaServer 4100 5/600 machines, each with 4 600 MHz 21164A processors and 2 GBytes of memory. Each AlphaServer runs Digital Unix 4.0F, with TruCluster 1.6 (Memory Channel) extensions. Each processor has three levels of cache, two levels of on-chip cache and an 8 Mbyte, direct-mapped, board-level cache with a 64-byte line size.

The servers are connected with a Memory Channel II network. We measured the network latency and bandwidth characteristics by a simple ping-pong test sending increasing size "packets" over the Memory Channel (we write contiguous chunks of memory of the "packet" size to simulate this). Uncontended latency for a 4 byte write is 3.3 microseconds. When one processor writes to another processor and the network is otherwise idle, we measured a maximum bandwidth of 80 Mbytes per second (for 1 Mbyte "packet" sizes).

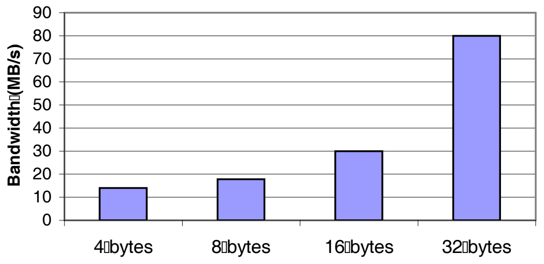

Figure 1. Effective Bandwidth (in Mbytes/sec) with Different Packet Sizes

Secondly, we used a test program to approximate the bandwidth variation of the system with the Memory Channel packet size. The Alpha chip has 6 32-byte write buffers. Contiguous stores share a write buffer and are flushed to the system bus together. The Memory Channel interface simply converts the PCI write to a similar-size Memory Channel packet. The current design does not aggregate multiple PCI write transactions into a single Memory Channel packet, so the maximum packet size supported by the system as a whole is 32 bytes. We measured the bandwidth variation by writing large regions with varying strides (a stride of one would create 32-byte packets, a stride of two, 16-byte packets and so on). Figure 1 shows the measured process-to-process bandwidths for 4 to 32-byte packets.

2.4 Benchmarks

We use the Debit-Credit and Order-Entry benchmarks provided with Vista [5]. These benchmarks are variants of the widely used TPC-B and TPC-C benchmarks.

TPC-B models banking transactions [10]. The database consists of a number of branches, tellers, and accounts. Each transaction updates the balance in a random account and the balances in the corresponding branch and teller. Each transaction also appends a history record to an audit trail. The Debit-Credit benchmark differs from TPC-B primarily in that it stores the audit trail in a 2 Mbytes circular buffer in order to keep it in memory.

TPC-C models the activities of a wholesale supplier who receives orders, payments, and deliveries [11]. The database consists of a number of warehouses, districts, customers, orders, and items. Order-Entry uses the three transaction types specified in TPC-C that update the database.

In both Debit-Credit and Order-Entry we issue transactions sequentially and as fast as possible. They do not perform any terminal I/O in order to isolate the performance of the underlying transaction system.

	Debit-Credit	Order-Entry
Single machine	218627	73748
Primary-backup	38735	27035

Table 1. Transaction Throughput for Straightforward Implementation (in transactions per second)

	Debit-Credit	Order-Entry
Modified data	140.8	38.9
Undo log	323.2	199.8
Meta-data	6708.4	433.6
Total data	7172.4	672.3

Table 2. Data Communicated to the Backup in the Straightforward Implementation (in MB)

The size of the database is 50 Mbytes, unless we explicitly say otherwise. This is the largest possible size for which we can do the appropriate mapping to the Memory Channel for all versions and configurations. We have also experimented with other database sizes (up to 1 Gbyte) and we show the results in section 7. For the experiments in section 8, where we run multiple transaction streams within a node, we use a 10 Mbyte database per transaction stream.

3 Straightforward Cluster Implementation

In an environment with a write-through network, the most straightforward extension of Vista to a primary-backup system is to simply map all of Vista's data — the database, the undo log, and the internal data structures — on to the backup node using the Memory Channel. Double writes are used to propagate writes to the backup. Other than inserting the double writes, this extension is completely transparent. On a failure, the backup simply takes over using its data structures, invoking the recovery procedure to undo any uncommitted transactions.

Table 1 presents the results for this implementation.

In short, throughput drops by a factor of 5.6 for Debit-Credit and by a factor of 2.7 for Order-Entry. This large drop in throughput is explained by the large amount of data that needs to be sent to the backup in this implementation, 7172 Mbytes for Debit-Credit and 672 Mbytes for Order-Entry. This communication and the implied extra local memory accesses necessary for the write-doubling adds 104.9 seconds to the 22.8 seconds single-machine execution time for Debit-Credit, resulting in a 5.6-fold decrease in throughput. Similarly, for Order-Entry, 672 Mbytes of data increase the execution time from 6.2 seconds to 17.1 seconds, or a 2.7-fold decrease in throughput. Closer inspection reveals that a very large percentage of the data communicated is meta-data, and only a small percentage reflects data modified by the transaction (see Table 2).

These numbers clearly indicate room for significant improvement by restructuring the software to use and communicate less meta-data. First, we discuss how we re-structure the standalone copy of the Vista library. Several versions are presented, reflecting further optimizations (see Section 4). While this re-structuring is obviously done with the intent of achieving better performance in a primary-backup configuration, it turns out that it improves standalone performance as well. Second, we show how to adapt those versions for primary-backup execution in which the backup remains a passive entity (see Section 5). We mean by this that the CPU on the backup is not used. All data travels from the primary to the backup by virtue of a double write on the primary's data structures. Third, we present primary-backup versions in which the backup takes an active role (see Section 6). In other words, the data is communicated from the primary to the backup by message passing, not by write through. The backup CPU polls for incoming messages, and executes the commands in those messages.

4 Restructuring the Standalone System

4.1 Version 0: The Vista Library

A `set_range` call results in an undo log record being allocated in the heap and put in the undo log, which is implemented as a linked list. The base address and the length of the range are entered in the appropriate fields of this record. A second area of memory is allocated from the heap to hold the current version of the data, a pointer to that memory area is entered in the log record, and a `bcopy` is performed from the range in the database to this area. Database writes are in-place. On commit, a commit flag is set, the undo log record and the memory area holding the old data are freed.

4.2 Version 1: Mirroring by Copying

This version avoids the dynamic allocations and linked list manipulations of version 0. First, the linked list structure of the undo log is replaced by an array from which consecutive records are allocated by simply incrementing the array index. On a `set_range`, we only update this array to record the `set_range` coordinates. Second, a "mirror" copy of the database is introduced. The mirror copy is initialized to the same values as in the database. Writes to the database are then done in-place. On commit, a commit flag is set and for each `set_range` record, the corresponding range in the database is copied into the mirror. The undo log records are de-allocated by simply moving the array index back to its original location.

4.3 Version 2: Mirroring by Diffing

The mirror copy can also be maintained in a different fashion. As in Version 1, we maintain an array to record the areas on which a `set_range` has been performed, and we maintain a mirror copy of the database. As before, writes to the databases are done in-place. On a commit, however, for each `set_range` performed during the transaction, we compare the database copy and the mirror copy of the corresponding areas, and we update the mirror copy if necessary.

Version 2 has fewer writes than Version 1, because it only writes the modifications to the mirror, while in Version 1 the entire `set_range` area is written to the mirror. Version 2 does, however, incur the cost of the comparisons, while Version 1 performs a straight copy.

4.4 Version 3: Improved Logging

This version avoids the dynamic allocations and linked list manipulations, as do Versions 1 and 2. In addition, rather than using a mirror to keep the data for an undo, this data is kept in-line in the undo log. Specifically, rather than using a log record with a pointer to an area holding the old data as in Vista, we use a log record that includes the data. On a `set_range` we allocate such a log record by simply advancing a pointer in memory (rather than by incrementing an array index), and writing to this area the offset and the length of the `set_range` area followed by the data in it. Database writes continue to be done in-place. On a `commit` the corresponding undo log records are de-allocated by moving the log pointer back over the appropriate amount.

Version 3 has the same write traffic as Version 1. Its writes are, however, more localized than Versions 1 or 2. Version 3 only writes to the database and the undo log, while in Versions 1 and 2, the mirror copy is also written.

4.5 Performance

Table 3 reports the performance for the two benchmarks for each of the four versions. Although done with the intention of improving primary-backup performance by reducing the amount of data written to the backup, all the restructured versions improve the standalone performance of the transaction server. The improvement in Versions 1 and 2 is mainly a result of avoiding linked list manipulations and dynamic memory allocations. Comparing these two versions, we see that the cost of performing the comparison between the database and its mirror over the `set_range` areas in Version 2 outweighs the gains achieved by performing fewer writes.

	Debit-Credit	Order-Entry
Version 0 (Vista)	218627	73748
Version 1 (Mirror by Copy)	310077	81340
Version 2 (Mirror by Diff)	266922	74544
Version 3 (Improved Log)	372692	95809

Table 3. Standalone Transaction Throughput of the Re-structured Versions (in transactions per second)

More importantly, the additional substantial improvement in Version 3 results from the increased locality in the memory access patterns of the server. Accesses are strictly localized to the database and the undo log, while Versions 1 and 2 also access the mirror copy, which is much larger than the undo log.

5 Primary-Backup with Passive Backup

5.1 Implementation

For each of the Versions 0 through 3, we can define an equivalent primary-backup version by simply mapping a second copy of the data structures in Memory Channel space, and double writing any updates to those data structures. The primary-backup Version 0 is what we used for the experiments in Section 3.

A slight modification allows primary-backup implementations of Versions 1 and 2 that are more efficient during failure-free operation, at the expense of a longer recovery time. When mirroring is used, we maintain the undo log on the primary, but we do not write it through to the backup. This reduces the amount of communication that has to take place. On recovery, the backup will have to copy the entire database from the mirror, but since failure is the uncommon case, this is a profitable tradeoff. The primary-backup results for these Versions reflect this optimization.

5.2 Performance

Table 4 presents the results for Versions 0 through 3 using a passive backup strategy. In addition, Table 5 shows the data transmitted over the network for each version, broken down into modified transaction data, undo data, and metadata.

Several conclusions may be drawn from these results. First, and most important, Version 3, with its improved logging, continues to outperform all other versions by a substantial margin, and this regardless of the fact that it writes much more data to the backup than Version 2, mirroring by diffing. Better locality in its writes, translates to better coalescing in the processor's write buffers with larger Memory

Benchmark	Version	Modified Data	Undo Data	Meta-data	Total Data
Debit-Credit	Version 0 (Vista)	140.8	323.2	6708.4	7172.4
	Version 1 (Mirror by Copy)	140.8	323.2	40.4	504.4
	Version 2 (Mirror by Diff)	140.8	140.8	40.4	322.1
	Version 3 (Improved Log)	140.8	323.2	141.4	605.4
Order-Entry	Version 0 (Vista)	38.9	199.8	433.6	672.3
	Version 1 (Mirror by Copy)	38.9	199.8	3.7	242.4
	Version 2 (Mirror by Diff)	38.9	38.9	3.7	81.5
	Version 3 (Improved Log)	38.9	199.8	14.5	253.2

Table 5. Data transferred to Passive Backup for Different Versions (in MB)

	Debit-Credit	Order-Entry
Version 0 (Vista)	38735	27035
Version 1 (Mirror by Copy)	119494	49072
Version 2 (Mirror by Diff)	131574	51219
Version 3 (Improved Log)	275512	56248

Table 4. Primary-Backup Throughput (in transactions per second)

Channel packet sizes as a result. Thus, even if the total data communicated is higher, Version 3 makes much better use of the available Memory Channel bandwidth. Second, the differences between the mirroring versions are small, with Version 2 better than Version 1 (unlike in the standalone configuration). The overhead of the extra writes in Version 1 becomes more important as these writes now have to travel through the Memory Channel to the backup. All modified versions are better than Version 0 (Vista), by 208% to 610% for Debit Credit and 80% to 108% for Order-Entry.

6 Primary-Backup with Active Backup

6.1 Implementation

Unlike the versions described in Section 5, in which the CPU on the backup node is idle, in active backup schemes the backup CPU is actively involved. With an active backup, the primary processor communicates the committed changes in the form of a *redo* log. It is then the backup processor's responsibility to apply these changes to its copy of the data. Since this is the backup processor's only responsibility, it can easily keep up with the primary processor. If the log were to fill up, the primary processor must block.

We only consider active backup approaches based on a redo log. The reason is that they always communicate less data. The redo log-based approaches do not have to communicate the undo log or mirror, only the changed data.

The redo log is implemented as a circular buffer with two pointers: one pointer is maintained by the primary (i.e., the producer) and the other pointer is maintained by the backup (i.e., the consumer). Quite simply, the backup processor busy waits for the primary's pointer to advance. This indicates committed data for the backup to consume. At commit, the primary writes through the redo log and only *after* all of the entries are written, does it advance the end of buffer pointer (the producer's pointer). Strictly speaking, the primary must also ensure that it does not overtake the backup, however unlikely that event may be. To avoid this possibility, the backup processor needs to write through its pointer back to the primary after each transaction is applied to its copy.

With active backup, although it is unnecessary for the primary to communicate mirror or undo log data to the backup, it must still maintain such information locally. We use the best local scheme, i.e., Version 3, to do this.

6.2 Performance

Table 6 compares the results for the best passive strategy to the active strategy. In addition, Table 7 shows the data transmitted over the network for these strategies, broken down in modified transaction data, undo log data, and meta-data.

The Active backup outperforms the Passive backup, by 14% for Debit-Credit and 29% for Order-Entry. The gains of the Active backup over the Passive backup for the Debit-Credit benchmark are comparatively smaller than the gains of either logging version over mirroring (14% versus 100%). We attribute this result to the significant reductions in communication overheads (PCI bus transactions and Memory Channel packets) due to coalescing already achieved by the best passive backup scheme. Compared to Debit-Credit, the throughput improvement of the Active backup over Passive backup is relatively higher in Order-Entry. The increased locality in the Active backup logging and further reduced communication overheads have more impact as the difference between the data transferred by the two versions is larger for this benchmark.

From Table 7, we see that the active approach tends to produce more meta-data to be sent to the backup. In the passive approach, the undo log carries some meta-data, but since the undo log is created as a result of `set_range` operations, a single piece of meta-data describes a whole

	Debit-Credit	Order-Entry
Best Passive (Version 3)	275512	56248
Active	314861	73940

Table 6. Passive vs. Active Throughput (in transactions per second)

contiguous region corresponding to the `set_range` arguments. In the active approach, in contrast, the meta-data describes modified data, which tends to be more spread out over non-contiguous regions of memory and therefore requires more meta-data entries. On the other hand, the `set_range` data is usually much larger than the actual data modified which results in a factor of 2 and 4 respectively decrease for the total data communicated.

7 Scaling to Larger Database Sizes

The Active backup is the only version where we are not limited by the Memory Channel space available and we can scale to any database size.

Table 8 presents the throughput variation of the Active backup version when increasing the database size. We see a graceful degradation in performance (by 13% and 22% respectively with a 1 Gbyte database). This is mainly due to the reduced locality of the database writes which results in more cache misses.

	Database sizes		
Benchmark	10 MB	100 MB	1GB
Debit-credit	322102	301604	280646
Order-entry	76726	69496	59989

Table 8. Throughput for Active Backup (in transactions per second) for Increasing Database Sizes

8 Using a Multiprocessor Primary

Parallel processing can increase the transaction throughput by the server. For throughput increases by a small factor, on the order of 2 to 4, such parallel processing can most economically/easily be done on commodity shared memory multiprocessors by a comparable number of processors. If the transaction streams are independent, or at least relatively independent, throughput increases should be near-linear. Synchronization between the different streams on an SMP is relatively inexpensive, and memory bandwidth appears to be sufficient to sustain multiple streams. When operating in a primary-backup configuration, this increase in throughput, however, puts increased stress on the SAN between the primary and the backup. We would therefore expect the use of an SMP as a primary to favor those solutions that reduce bandwidth usage to the backup.

To validate this hypothesis, we carried out the following experiment. We used a 4-processor SMP as the primary, and executed a primary transaction server on each processor. The data accessed by the stream fed to different servers has no overlap, so the servers can proceed without synchronization, thus maximizing the possible transaction rates and exposing to the fullest extent possible any bottlenecks in the SAN.

The results are shown in Figure 2 for the Debit-Credit benchmark and in Figure 3 for the Order-Entry benchmark. We see that the Active logging version, shows a nearly linear increase in the aggregate throughput, while all the other versions do not scale well. The better scalability of the logging versions compared to the mirroring versions is again due to the fact that their accesses to I/O space memory show more locality with better opportunities for coalescing into larger I/O writes. This results in fewer transactions on the I/O bus and fewer Memory Channel packets sent, hence, lower communication overheads even when the total data communicated is higher.

The two benchmarks modify data mostly in small size chunks in random locations of the database with a large fraction of writes in 4-byte chunks (especially in Debit-Credit). With 32-byte packets, the process-to-process effective bandwidth of the system is 80 Mbytes/sec, while for 4-byte packets, the effective bandwidth is only about 14 Mbytes/sec (Figure 1). The Active logging version sends 32-byte packets, and thus takes advantage of the full 80 Mbytes/sec bandwidth and does not become bandwidth limited for either benchmark. The Passive logging version produces mixed packets (32-byte packets for the undo data and small packets for the modified data). Furthermore it sends more total data than the Active logging and becomes bandwidth limited at 2 processors. The two mirroring protocols do not benefit at all from data aggregation between consecutive writes and see an effective bandwidth below 20 Mbytes/sec with practically no increase in aggregate throughput with more processors.

9 Related Work

Clusters have been the focus of much research and several commercial products are available. Older commercial systems, such as Tandem [1] and Stratus [9], use custom-designed hardware. More recent efforts, such as Microsoft Cluster Service [12] and Compaq's TruCluster [13], use commodity parts.

Zhou et al. [15] address the more general problem of transparent process primary-backup systems, using check-

Benchmark	Version	Modified Data	Undo Data	Meta-data	Total Data
Debit-Credit	Best Passive (Version 3)	140.8	323.2	141.4	605.4
	Active	140.8	0	141.4	282.2
Order-Entry	Best Passive (Version 3)	38.9	199.8	14.5	253.2
	Active	38.9	0	24.7	63.6

Table 7. Data transferred to Active Backup vs. Passive Backup (in MB)

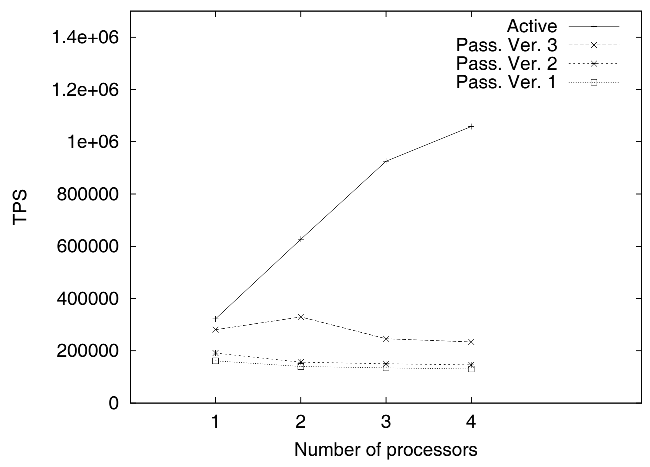

Figure 2. Transaction Throughput Using an SMP as the Primary (Debit-Credit Bench mark, in transactions per second)

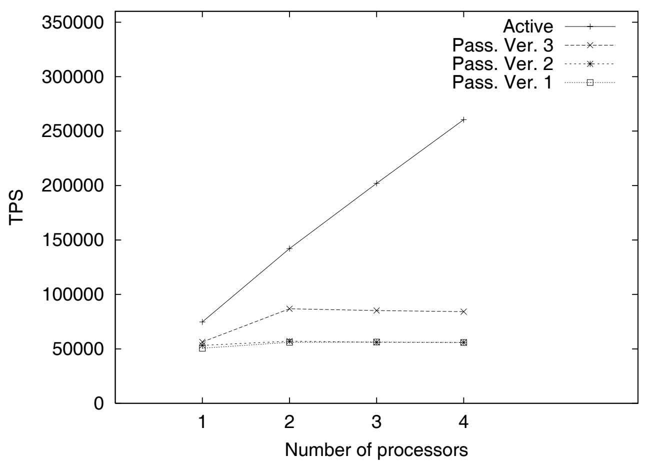

Figure 3. Transaction Throughput Using an SMP as the Primary (Order-Entry Benchm ark, in transactions per second)

pointing and rollback, while our work focuses exclusively on transaction processing. They do, however, use the Vista transaction processing system as an example application of their system. Their reported results were measured on a Shrimp network [6] connecting 66Mhz Pentiums. They came to the conclusion that transparent write through, with some optimizations, leads to acceptable performance in their environment. By virtue of doing solely transaction processing, our straightforward Vista implementation (Version 0) implements essentially the same optimizations, but performance remains disappointing. We attribute the difference in outcome to their use of a much slower processor (66Mhz Pentium vs. 600Mhz Alpha), while the networks are comparable in bandwidth. Shrimp also snoops on the memory bus, and therefore does not require double writing, which may be another factor affecting the results.

Papathanasiou and Markatos [7] study the use of remote memory to backup transaction processing. Although they do not use Vista, their standalone implementation appears similar, and the primary-backup implementation uses write through of the data structures of the primary. Their implementation uses 133Mhz Pentium processors running Windows NT and connected by a SCI ring network. They also use the Vista benchmarks to measure the performance of their system. Their results are difficult to calibrate because they only provide primary-backup performance results, and no indication of standalone performance.

We use the same API as RVM [8] does to implement recoverable virtual memory, but the RVM system is based on disk logging. All disk-based systems, even optimized to write to a sequential log on disk, are ultimately limited by disk bandwidth. Vista used the RVM API to implement transactions based on reliable memory [5], thereby considerably improving its performance, but data availability is reduced because all data remains in memory. Our system overcomes Vista's main limitation, limited data availability, in an efficient way. Similar limitations exist in other systems based on reliable memory, e.g. Wu and Zwaenepoel [14].

10 Conclusions

Primary-backup services are essential in clusters attempting to provide high-availability data services. In this paper we have focused on transaction processing systems. We started from a standalone high-performance transaction processing system, Vista, that relies on reliable memory to achieve high transaction throughput. Although Vista would seem a prime candidate for conversion to a primary-backup system using the write through capabilities of modern system area networks, we found that such an implementation leads to transaction throughput much below that of a standalone server. Significant restructuring of the server was needed to achieve good performance. In particular, versions of the server that use logging optimized for locality in local and I/O space memory access were found to offer much better primary-backup performance than the original Vista system (by up to 710%) or systems that use mirroring (by up to 160%). Moreover, this result was found to be true not only for primary-backup configurations, but also for standalone servers. The primary-backup configuration with the best locality had both the best throughput and scaled well to small shared-memory multiprocessors.

Acknowledgments

This work was supported in part by NSF Grants CCR-9457770 and MIP-9521386 and by the Texas TATP program under Grant TATP-003604-0009-1997. We would also like to thank the anonymous reviewers for their comments, and Sandhya Dwarkadas, Robert Stets and Umit Rencuzogullari for their help and advice during our work on this paper.

References

[1] J. F. Bartlett. A Non Stop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, Dec. 1981.

[2] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Oct. 1996.

[3] Compaq, Intel and Microsoft Corporations. Virtual interface architecture specification, Version 1.0, Dec. 1997.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[5] D. Lowell and P. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[6] M. Blumrich et al. Design choices in the SHRIMP system: An empirical study. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[7] A. Papathanasiou and E. Markatos. Lightweight transactions on networks of workstations. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, 1998.

[8] M. Satyanarayanan, H. Mashburn, P. Kumar, and J. Kistler. Lightweight recoverable virtual memory. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 146–160, Dec. 1993.

[9] D. Siewiorek and R. Swarz. *Reliable Computer System Design and Evaluation*. Digital Press, 1992.

[10] Transaction Processing Performance Council. TPC benchmark B standard specification, Aug. 1990.

[11] Transaction Processing Performance Council. TPC benchmark C standard specification, revision 3.2, Aug. 1996.

[12] W. Vogels and et al. The design and architecture of the microsoft cluster service. In *Proceedings of the 1998 Fault Tolerant Computing Symposium*, 1998.

[13] W.M. Cardoza et al. Design of the TruCluster multicomputer system for the digital unix environment. *Digital Equipment Corporation Technical Systems Journal*, 8(1), may 1996.

[14] M. Wu and W. Zwaenepoel. eNVy: A non-volatile main memory storage system. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 86–97, Oct. 1994.

[15] Y. Zhou, P. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. In *Proceedings of the 1999 International Conference on Supercomputing*, June 1999.