CARP: Handling silent data errors and site failures in an integrated program and storage replication mechanism

Lanyue Lu[†], Prasenjit Sarkar, Dinesh Subhraveti, Soumitra Sarkar[*], Mark Seaman, Reshu Jain, Ahmed Bashir
Rice University[†], IBM Almaden Research Center, IBM Watson Research Center[*]
Email: ll2@rice.edu, psarkar@almaden.ibm.com, {dineshs,sarkar,seamanm,jainre,abashir}@us.ibm.com

Abstract

This paper presents *CARP*, an integrated program and storage replication solution. *CARP* extends program replication systems which do not currently address storage errors, builds upon a record-and-replay scheme that handles nondeterminism in program execution, and uses a scheme based on recorded program state and I/O logs to enable efficient detection of silent data errors and efficient recovery from such errors. *CARP* is designed to be transparent to applications with minimal run-time impact and is general enough to be implemented on commodity machines. We implemented *CARP* as a prototype on the Linux operating system and conducted extensive sensitivity analysis of its overhead with different application profiles and system parameters. In particular, we evaluated *CARP* with standard unmodified email, database, and web server benchmarks and showed that it imposes acceptable overhead while providing sub-second program state recovery times on detecting a silent data error.

1. Introduction

Service downtime is one of the major reasons for revenue loss in enterprises that is typically addressed by providing redundant software and hardware. One approach to providing software redundancy is to mirror the state of a running program to a set of replicas such that, in case of a failure, one of the replicas assumes the place of the previously running application instance. This provides the paradigm of continuously available replica programs, where a replica can take over the functionality of the failed primary without any service downtime [1], [2]. Also, this technique avoids the costly task of starting a new application instance on a different machine.

However, program replication is not sufficient if critical applications have to survive site failures. In the event of a site failure, the program as well as its underlying storage has to be replicated to a different site in a synchronized manner. It is not possible to implement program and storage replication with continuous availability (transition to the replica program upon detection of a failure in the primary without application outage) by separately leveraging program replication and storage replication, where the latter could be implemented using synchronous storage system of Logical Volume Manager (LVM) mirroring. One needs a more integrated approach as described in Section 2.2.

Integrating storage replication introduces a new set of complications. Specifically, two types of errors need to be handled to prevent replica divergence: *hard errors* and *silent data errors*. Hard errors include complete disk failures [3] and latent sector errors [4], which are due to the mechanical characteristics of hard drives. Silent data errors [5] represents a class of errors where the read data returned by the storage system does not match the data that was last written to the same disk address. While hard errors are detectable and easier to recover from using standard program replication techniques, silent data errors are not detected by conventional protection techniques and are growing in magnitude due to increasing disk capacities. The byte error rate of silent data error is 10^{-7} reported in [6]. The percentages of disks affected by silent data error per year for nearline and enterprise class disks are 0.466% and 0.042% respectively [5]. Traditional program replication techniques do not handle silent data errors, and the execution of a program replica which operates on corrupted data will eventually diverge from other copies. In other words, if undetected, the corrupted data can contaminate the program state.

In this paper, we introduce *Continuously Available Replicated Programs (CARP)*, an integrated program and storage replication system that provides high availability, data replica consistency, and also implements an efficient scheme for program replicas to detect and recover from silent data errors. *CARP* addresses a problem that is fundamentally broader in scope than that of file system and database management schemes, which handle silent error detection and correction but do not address application-level recovery from site failures. *CARP* extends program replication systems which do not address storage errors, builds upon a record-and-replay scheme that handles nondeterminism in program execution, and introduces recorded program state and I/O logs to enable efficient detection of silent data errors (without a continuous checksumming overhead) as well as efficient recovery from such errors. We targeted three key goals for *CARP*: (i) *efficiency* in that the system should only introduce minimal overhead for the replication; (ii) *fast recovery* in that the replica programs should quickly

recover from the corruption introduced by silent data errors; (iii) *transparency* in that the system should not require applications to be modified since the source code of most commercial applications is not available.

We implemented *CARP* as a prototype on commodity Linux machines and conducted a series of micro-benchmark experiments to determine its sensitivity to different application profiles and system parameters. We also ran unmodified email, database, and web application benchmarks on top of *CARP* to measure its overhead in real-life situations. Our results show that the overhead of *CARP* in these applications ranges from 11% to 40%, with a recovery time of less than a second in response to silent data errors.

The rest of the paper is organized as follows: Section 2 presents a high-level overview of *CARP*. Section 3 describes the detailed architecture of *CARP* with particular emphasis on the algorithm to detect and recover from silent data errors and site failures. Sections 4.1 and 4.2 present the results of our micro-benchmark experiments and the real-life behavior of *CARP* respectively. Related work is introduced in Section 5, and we present conclusions in Section 6.

2. Overview of *CARP*

2.1. Program Replication

The program replication technique that *CARP* builds on is called *Record and Replay (RR)* [7]. In principle, *CARP* can be built on top of any program replication system that provides similar capabilities, such as [8], [9]. *RR* is designed to support the recording and subsequent replay of the execution of unmodified applications running on multiprocessor systems. Multiple instances of an application are simultaneously executed in separate virtualized environments called *Containers*, which facilitate state replication between the application instances by resolving resource conflicts and providing a uniform view of underlying operating system resources across all replicas. In general, *RR* addresses the replication of relevant operating system state information including the state maintained within the kernel, such as network state to preserve network connections across failures, as well as the state resulting from non-deterministic interleaved accesses to shared memory in SMP systems and asynchronous order of writes.

The underlying RR system addresses each source of non-determinism visible to the application and ensures its deterministic replay. In particular, nondeterminism originating in the system calls is addressed by serializing them, recording the result along with their relative order and playing back the same result and order. This approach addresses nondeterminism due to system calls such as `gettimeofday`, and also interleaved asynchronous `writes`. Nondeterminism due to concurrent accesses to shared memory is addressed by efficiently recording shared memory accesses and enforcing identical interleaving. Nondeterminism originating within the network stack is addressed by recording the state of the application sockets prior to each nonabortable send and restoring it on failover. Kernel state which is not directly visible to the application is not explicitly replicated.

A key limitation of *RR* is that it addresses program replication without considering the storage system state. *CARP* considers the storage related issues and in particular, extends *RR* to support recovery from silent data errors in a replicated program environment.

2.2. Integrated Replication

As mentioned in Section 1, a combined storage and program replication mechanism is required in order to recover cleanly from a site failure. For example, if an application and its associated storage both fail at the primary site, the mechanism must ensure a synchronized switch-over to a replica site. Otherwise, if the program replication mechanism performs a failover first, the program may be started at a replica site with inconsistent storage data. In particular, all writes in the operating system buffer cache that are not yet committed to disk in the primary must also be reflected across replicas in order for the program to recover correctly.

Motivated by the need to integrate storage and program replication, we investigated three possible mechanisms that can provide continuous availability and immunity to site failures. The first mechanism is based on LVM mirroring, which replicates the data writes between the LVM layers in primary and replica to keep the contents of the logical volumes synchronized. The second mechanism uses general block level synchronous replication [10] between the primary and replica storage systems to attain the same goal. In both of these mechanisms, only storage system writes are replicated, not the contents of the buffer cache. In the third mechanism, the program and its replicas are responsible for storage replication. The program replicas access local storage independently, and by virtue of having their executions synchronized using program replication techniques, their respective storage remains identical.

Of the three mechanisms considered above, only the third can support continuous application availability upon site failure. With the first two schemes, the application on the secondary node cannot mount a file system on the replica volume since it is read_only and being constantly updated to reflect changes in the primary. Therefore, the secondary application's read/write operations have to be monitored and its input data is supplied from the primary using the program replication channel. However, on failure of the primary node, the only recovery option is to mount the secondary's file system on the replica volume after changing it to read-write state, at which time the file system will be oblivious of the secondary application's open file states. This problem can be addressed by restarting the application, but that would violate the continuous availability constraint. The third scheme works since it ensures that all replica file

systems have the same data, and the loss of uncommitted writes at the primary site does not result in loss of data at replica sites, allowing failover to the secondary application without a restart.

A simple overview of the *CARP* architecture is shown in Figure 1. Although the principles underlying *CARP* can be extended to multiple replicas, for simplicity and to reflect our prototype implementation, we consider the case of a single secondary replica supporting a primary application. In the description of the design, we assume without the loss of generality that program A and its clone B run on different hosts, each in an *RR* Container. Each program is backed by a file system and back-end storage environment, and the two environments are equivalent in that the file system name-space relevant to programs A and B are identical. A and B each issue system calls (e.g., open, close, read, write) on separate files of the same name on each host, and each host's file system stores the underlying data on a separate storage system. A replication link between the two containers is used to exchange program state information (needed for program replication) as well as I/O-related information (needed to handle silent data errors) to ensure the deterministic execution of program B with respect to A. *CARP* is transparent by design and works with unmodified applications since *CARP* operates only by intercepting file system calls made by these applications.

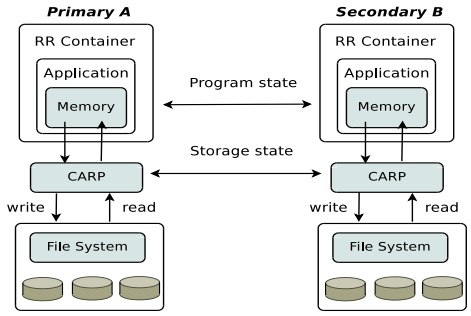


Figure 1. An architectural overview of *CARP*

2.3. Silent Data Errors

An integrated program and storage replication mechanism must deal with new issues introduced by storage replication, particularly those that may potentially contaminate program memory and cause program replicas to diverge. *Silent data errors* represent one such issue that is understood to occur whenever a read or write disk head seeks data improperly. Silent data errors are largely workload dependent and highlight an important limitation of conventional error-handling techniques, which offer protection from issues that are largely orthogonal to those posed by silent data errors [5].

CARP implements silent error detection using checksumming. The checksum is stored independently of the data and computed after all layers of the storage stack have completed processing the data. Recovery from silent data errors requires multiple copies of the data to identify the correct copy. In the program replication environment, recovery involves restoring not only the correct data but also the program state that has been affected by corrupt data.

In the next section, we describe how *CARP* combines an integrated program and storage replication mechanism with checksumming to provide quick, low overhead recovery from silent data errors.

3. Architecture and Design

We first describe a simple approach to detect silent data errors. Assuming the executions of programs A and B are synchronized at time T, when A reads data from a file on its host, a checksum of that data is computed and communicated to the *RR* Container of program B. When B performs the corresponding read operation from its local file system, the checksum computed on the data read is compared with that sent by A's Container. Any discrepancy in the checksums indicates a silent data error.

Exchanging checksum information on every read can be expensive, especially given that silent data errors are relatively rare. *CARP* implements an optimized version of this approach where the checksum exchange overhead is amortized over a number of read operations. Instead of checking for possible silent data errors on every read, checksum comparison is performed periodically in one of three ways: (a) Periodic cross-Container checking of read checksums. (b) Before writing data to the external environment. (c) After detection of the divergence of execution states between the program replicas by the *RR* program replication system.

This periodic checksum comparison is referred to as a *CARP* checkpoint and is distinct from the *RR* program replication checkpoint. In the rest of the paper, the term checkpoint refers to the *CARP* checkpoint. On detection of a silent data error, the state of the program instance that used corrupt data would no longer be valid and has to be restored to the current state of the surviving instance. For efficiency, *CARP* only restores the state of the program that may have been polluted by the corrupt data. It tracks the side effects of program execution - the virtual memory pages updated, and the read and write operations performed - between periodic checkpoints. Tracking of updated virtual memory pages allows efficient program memory state recovery from silent data errors without requiring a complete restart.

The rest of the section provides the detailed design of *CARP* with respect to the various phases involved.

3.1. Initialization

For each execution cycle, the initial condition that must be ensured is that the contents of all existing files to be accessed by programs A and B are identical and error-free. A simple approach for checking this is to calculate the checksum of the complete file when it is first opened by a program, and

to subsequently compare it with the checksum computed by the Container of the replica program instance when it executes the open call. During execution, the content of each file accessed by programs A and B are kept synchronized and therefore, a similar check is not necessary when each program terminates.

Ideally, this initial condition would only have to be checked the very first time the program replicas are run. However, running the check at the beginning of each execution cycle improves robustness since it eliminates human errors (such as running only one copy without replication, which could cause file contents to diverge).

The initialization phase is invoked at the start of the execution cycle of programs A and B. Each program maintains a set of in-memory state variables. The *running_read_checksum* variable serves as a unique signature of all data read by the program since the previous checkpoint, or since the start of the program if no checkpoint has been performed yet. It is updated continuously after every read request. The *dirty_pages_list* variable keeps track of all virtual memory pages updated by the program since the last checkpoint (or program start). The *read_operations_queue* variable keeps track, in temporal order, of the set of read operations (on all files) performed by the program instance since the previous checkpoint (or program start).

The primary program A also maintains one *write_operations_log* for each file that is created or updated by the program. This log serves as the third copy of each file and is persisted on the file system of the primary server. The log is maintained in memory during program execution (with appropriate cache flushing policy) for efficiency. *CARP* stores the *write_operations_log* and the original data on different disks in A. This greatly reduces the probability of both copies of the data in different disks having silent data errors at the same time. The *write_operations_log* file for a data file is cached appropriately into an in-memory hash table for efficient reference.

3.2. Read and Write Processing

The read and write processing phases are executed on each program instance during the processing of each read and write system call. *CARP* intercepts every read and write system call (file, offset, length) issued by the program instance. In case a read or write is done through a memory map, the processing phases are conducted when the respective pages are read or written from the storage by instrumenting the kernel buffer cache. The algorithm itself is agnostic to the level at which file accesses are intercepted. For efficient checksum computation and storage space usage, we chose to calculate the data checksum at the page level instead of the whole file or data block level. If the application only accesses a small part of a big file, then file-level checksum computation is not efficient. However, a block-level checksumming strategy requires multiple checksum computations for typical references, as well as a larger checksum storage overhead. For these reasons, we calculate the checksum at the page level, using the page size of 4 KB in Linux. For both read and write operations, all pages (of a configured size) of file data that completely contain the region of data being read or written are accessed. If the operation is targeted at the end of the file such that the last page boundary in the file contains less than a page of data, then only the partial page is accessed.

Read processing involves computing the signature of the request and the checksum of the actual data read. For every file page accessed, *CARP* creates a new *read_operations_queue* entry to record the tuple <request signature, read data checksum>. The request signature is the tuple <file, page_no, length>. The data read is also used to update the *running_read_checksum* variable.

The write processing phase is performed only on program A (the primary instance). The *write_operations_log* is updated with an entry per page which records the information <request signature, data checksum>. Each in-memory *write_operations_log* is flushed to disk in a lazy fashion: there is no dependency between the flushing of a page and that of its checksum record, but when a close or fsync call is issued on a file, the corresponding in-memory *write_operations_log* entries should be flushed to maintain sync-on-close semantics for the log. When the file is closed or deleted by the application, *CARP* keeps the in-memory *write_operations_log* until the next checkpoint. If no errors are detected at the checkpoint, then *CARP* frees the *write_operations_log* in memory for the closed or deleted file. For a deleted file, the *write_operations_log* on disk is also deleted at the checkpoint.

One key concern in checksum computation is the impact on the overhead of *CARP*. Cryptographic hash functions such as MD5 and SHA1 which provide strong collision resistance and unpredictability are used to compute the checksum. However, these functions are also compute intensive and can steal cycles from the application if invoked repeatedly. In the implementation of *CARP*, we evaluated two methods: a software MD5 implementation and the hardware CRC32 instruction [11]. The latter is a new feature of Intel SSE 4.2 instruction set that only needs 0.4 cycles to compute the checksum on one byte of data as opposed to 5-15 cycles in a typical software implementation [11].

3.3. Virtual Memory Tracking

Tracking of virtual memory page updates enables efficient program restart on detection of a silent data error. If virtual memory tracking is not in place, the program that accessed dirty (corrupted) data would have to be restarted from a *RR* program replication checkpoint of the surviving program instance, and all its memory pages would have to be replaced. This form of checkpoint restart would be expensive for an

application with a large memory footprint. The use of virtual memory tracking allows us to limit memory page refreshes to those which are potentially affected by a silent data error. Our scheme can result in some virtual memory pages of the failed program being replaced even if they are not affected by corrupted file data being read. However, our experimental evaluation demonstrates that the overall benefit of virtual memory tracking outweighs the overhead of the scheme.

We evaluated two methods of virtual memory tracking in *CARP*. The first method is illustrated in [2] and is based on intercepting page faults. At the beginning of a checkpoint, *CARP* changes the access control setting on all virtual memory pages of the program to READ_ONLY. During normal program execution, each update to a new virtual memory page by the program results in an access control type of page fault. Upon detection of the page fault, the *CARP* trap handler modifies the page permissions to READ_WRITE and adds the page address to the *dirty_pages_list* for error recovery purposes. Tracking page faults can be expensive for applications that have large memory footprints. The second method evaluated in *CARP* addresses this problem by using a page table scanning method, where the dirty bit in the page table is used to identify the virtual memory pages modified by the application in between checkpoints. At the start of a checkpoint, *CARP* clears all the dirty bits of pages in the application's memory space. When the program modifies a virtual memory page, the hardware automatically marks the dirty bit for that page in the page table. Once a silent data error is detected on a checksum mismatch, *CARP* scans the page table of the application's memory space to find the dirty pages since the last checkpoint. A performance comparison between the two methods is discussed in Section 4.1.

3.4. Silent Data Error Detection

The overall process of silent data error detection and recovery is illustrated in Figure 2. The periodic checking mechanism works as follows. During the processing of read system calls, for every N read calls processed (where N is a configured value), the current *running_read_checksum* value is sent to the replica. In the *CARP* scheme, since there is a clearly defined primary copy, e.g., program A, this is sent from A to B. If the two values do not match, then a silent data error has occurred in one of the instances and error recovery is initiated. Checking is also enforced before sending output to the user and during program termination regardless of N.

All data written by a program instance must be checked against its replica for the possibility of divergence caused by a read error. This is important because any communication of erroneous information to the external world can result in an irreversible action by an end user or a dependent program. Therefore, the second checking mechanism leverages the Container's ability to trap system calls that write to the console or a network pipe (socket). On detecting such a system call, the mechanism performs a cross-Container validation of the checksum of the data written to ensure that the two copies are writing identical data. If the checksums do not match, the recovery phase is initiated.

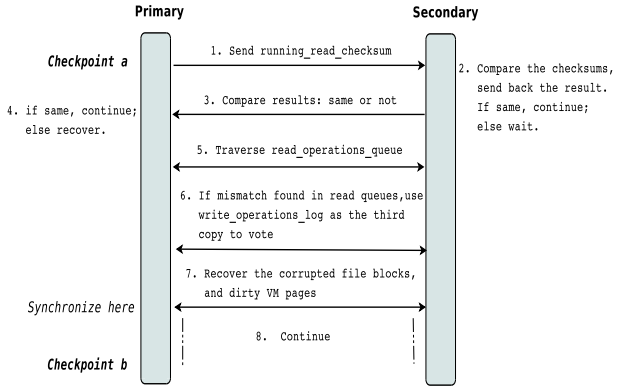


Figure 2. The overview of silent data error detection and recovery process

The third checking scheme exploits the program replication mechanism in *RR*, which can detect that program B has issued a monitored system call that A never issued, thus indicating that state divergence has occurred. Though it is possible that something other than a silent data error caused this divergence, *CARP* assumes that a silent data error is the cause and initiates error recovery.

A key concern about the above mechanisms is the communication overhead between the replicas. Exchanging synchronous network messages between the primary and the secondary is not advisable as message latency on the networks can add to *CARP* overhead. Instead, we use an asynchronous scheme for exchanging state information between the replicas. Once the state information is sent, the primary continues its execution without waiting for a response. When the result of the comparison is returned, the primary decides whether to continue execution or perform recovery processing.

During a checkpoint, the comparison of the *running_read_checksum* between replicas may not yield any evidence of a silent data error, in which case *CARP* clears the in-memory *read_operations_queue* and the *dirty_pages_list* variables, and clears the dirty bit of each virtual memory page of the program. The state of the program at this point is marked as a known good state. The read processing, write processing and virtual memory tracking phases are then repeated.

3.5. Storage and Program Recovery

The recovery phase is invoked when a silent data error is detected, implying the presence of corrupted data in one of the two systems. For *CARP* to determine which file is correct requires the use of a third copy as a tie-breaker. The *write_operations_log* serves as the source of checksums of

the data file content, and is used to determine the correct version of the data file to use to replace the corrupted file.

The recovery steps are performed in a synchronized fashion between the two Containers. First, the algorithm traverses the *read_operations_queue* on both A and B, comparing the request signature (file name, page number and length) and data checksum value of each entry. If the request signatures match but the checksums do not, then one instance has corrupted data and the algorithm proceeds to correct the content of the corrupted file. To identify the corrupted file, *CARP* retrieves the checksums corresponding to the pages read from the *write_operations_log* on the primary and compares the checksums with those obtained by actually reading those pages from the files on both machines. The file whose page checksums match the ones from the write operations log has the correct copy of the pages and that copy is used to replace those pages in the corrupted copy of the file. Since we assume that the file systems of the primary and secondary nodes are checksum identical at the beginning of execution cycle, thus the reads of blocks that are not written to are also guaranteed to be identical. The algorithm continues the comparison step for subsequent entries in the *read_operations_queue*. Since the *RR* program replication system can detect all divergences in program state between A and B, the request signatures in the *read_operations_queue* for programs A and B are guaranteed to be identical.

Once the effects of silent data error have been corrected on the individual file systems, the virtual memory pages in the *dirty_pages_list* of the correct program are copied to the corrupted program instance. All registers and other critical resources (e.g., file descriptors) from the correct program instance are also copied to the corrupted copy. While CARP restores only memory pages that potentially contain corrupt data, it does not attempt to restore the kernel state incrementally. Application's state is dominated by its memory footprint. Kernel state itself is negligible in comparison and it is checkpointed in its entirety and restored. Next, each *CARP* instance clears the in-memory *read_operations_queue* and *dirty_pages_list* variables, clears the dirty bits of all virtual memory pages of the program, and resumes program execution.

It is theoretically possible that there is a mismatch between the primary, the secondary and the *write_operations_log* for a page, making the determination of the correct copy unresolvable. In that case the efficient recovery scheme described above is not feasible and both programs have to be restarted from a previous *RR* program replication checkpoint. Since this scenario indicates multiple silent data errors occurring in close temporal proximity and affecting the same page, we believe that the probability of this event is very small.

3.6. Site Failure Detection and Recovery

CARP is built on top of *RR*, which provides the basic infrastructure for communication (and synchronization) of state information between the primary and secondary nodes. *RR* depends on distributed systems techniques (IBM's RSCT [12]) to reliably alert it when its partner node has failed. RSCT implements a heartbeat-based topology services abstraction on top of which a group services abstraction is provided. The group services mechanism forms the basis of a group-membership-based majority quorum. The system has been adapted in the past to include a disk-based quorum determination algorithm for 2-node clusters, which uses disk-based heartbeating and a role-based backoff protocol to decide which node is in charge after a failure. The overall scheme only works well in a LAN environment.

Because the RR secondary flushes and commits the event log before the primary releases any information to the external world, the secondary's state is always sufficiently synchronized with that of the primary to enable it to take over upon primary failure. If the primary is told that the secondary is dead, program execution continues on the primary without replication. If the secondary is told that the primary is dead, then IP address takeover by issuing a reverse ARP packet on the network must first be carried out, which the secondary can do reliably since the quorum algorithm above ensures that the primary is dead (possibly because it shut itself down).

The failback operation involves restarting the application on the (old) primary and restoring replication from the (old) secondary. This includes the file system state, the RR runtime, the RR-RR communication link, and the (old) secondary's application execution state. To restore the file system state on the (old) primary node from the (old) secondary, a well known file system replication technique as in Coda is used. Once the file systems are synchronized, the RR runtime system is started on the (old) primary node and the communication link with the RR on the (old) secondary node is reestablished. Next, the execution state of the secondary application is set to STOPPED, and file system replication is terminated since any further data synchronization will be achieved by CARP + RR replication. A full checkpoint of the secondary application is created using the local RR runtime, transmitted to the primary node's RR runtime over the network link, and restored in the primary node in the STOPPED state. Finally, the application is resumed (its execution state changed to RUNNING) on both nodes, at which point program and storage replication using RR + CARP is restored and failback is complete.

4. Evaluation

We have implemented the *CARP* prototype in the Linux 2.6.18 kernel as a loadable kernel module, which provides the full functionality of checksum comparison, virtual memory page tracking, storage and program state recovery. To

understand the sources of overhead, we first evaluated *CARP* using a detailed micro-benchmark analysis with different application profiles and system parameters. To validate the hypotheses of low overhead and quick recovery of *CARP* in real application environments, we tested with unmodified Internet email, database and web server benchmarks. All the experiments were conducted on two commodity servers with quad-core Intel Xeon 3.0 GHz processors, 2 GB of main memory, 80 GB SAS disk and 1 GBps Broadcom Netxtreme network interface.

Parameter	Default
Number of reads between checkpoints	50
Total number of operations	100,000
Read/write ratio between checkpoints	0.5
Number of dirty memory pages generated between reads	5
Application memory usage	1 MB
Application file size	1 MB

Table 1. Micro-benchmark parameters and default values

We used the execution time overhead of *CARP* over the *baseline* (bare machine) and *baseline-RR* (RR program replication system), and the number of memory pages transferred for application recovery as the principal metrics of comparison. The first metric indicates the penalty imposed on applications to run in the *CARP* environment. The second metric is the principal factor that determines how fast an application can recover from a silent data error assuming a well provisioned network that is typical in disaster recovery scenarios. We also report on other metrics such as the disk space and memory overheads imposed by *CARP*. The evaluation of handling site failures is our future work.

4.1. Micro-benchmark Evaluation

This section explores the parameter space to identify the overhead of *CARP*. Our micro-benchmark issues a series of reads and writes to a test file with an equal number of random and sequential accesses, while generating dirty pages at random memory locations between successive reads. There are several configuration parameters which we varied to evaluate the sensitivity of *CARP* to these parameters. These parameters and their default values are shown in Table 1.

We compared four variants of *CARP*: *baseCARP*, *baseCARP+asyn*, *baseCARP+asyn+crc* and *baseCARP+asyn+crc+pte*. The *baseCARP* sends synchronous network messages for exchanging the checksum at each checkpoint, uses software MD5 to compute the checksum, and marks the virtual memory pages read-only to track the dirty memory pages using the page fault handler. The *baseCARP+asyn* adds the optimization of sending asynchronous messages for checksum exchange. The *baseCARP+asyn+crc* adds the use of the hardware CRC32 instruction for computing the checksum instead of MD5. Finally, the *baseCARP+asyn+crc+pte* adds the mechanism of scanning the page table to track the dirty memory pages instead of trapping page faults. In the following experiments, we will use *C1* to *C4* to denote the above four versions of *CARP*.

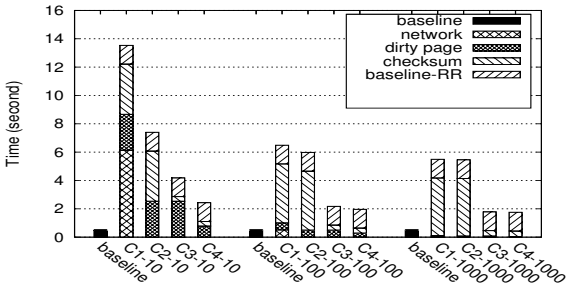


Figure 3. Analysis of the overheads for checkpoint frequencies

The total execution time of *CARP* in the micro-benchmarks is partitioned to four parts: network delay, dirty page tracking, checksum and *baseline-RR*. The *network delay* overhead is that of exchanging the checksum between the primary and the replica. The *dirty page tracking* overhead is that of detecting the dirty memory pages since last checkpoint. The *checksum* overhead is that of checksum computation and management. The *baseline-RR* overhead is that of executing the benchmark in the *RR* program replication environment on which *CARP* is based. While the *baseline* is running the benchmark on the bare machine without RR or *CARP*.

4.1.1. Effect of Checkpoint Interval. A value of *N* for the checkpoint interval implies that the primary and the replica exchange the checksum for every *N* read operations. With a default set of parameters, we varied *N* from 10 to 1000. As Figure 3 shows, the total execution time of the *CARP* variants is partitioned into four parts as discussed in our experimental methodology. The execution time of the *baseline* is not affected by the variation of *N*. The total execution time of all the *CARP* variants decrease with an increase in *N* because a larger value of *N* leads to less frequent checkpointing, which results in less network delay and dirty page tracking overhead. For a value of 10 for *N*, the execution time of *C1* is 10.23 times that of the *baseline-RR*, while for *C4* the execution time is 1.84 times that of the *baseline-RR*. For a value of 1000 for *N*, the execution time of *C1* is 4.16 times that of the *baseline-RR*, while for *C4* the execution time is 1.33 times that of the *baseline-RR*. *C4* significantly outperforms *C1* due to the optimizations described in Section 4.1 above. For $N = 1000$, we find that the network delay and dirty page tracking overheads are very small for all the *CARP* variants, but the checksum overheads of *C1* and *C2* are still 10.57 times that of *C3* and *C4* due to use of hardware CRC instructions rather than performing the MD5 calculation in software. For comparison of the *baseline*, for $N = 1000$, the execution time of *C4* is 3.5 times of the *baseline*, since this experiment is very system call intensive, leading to big overhead of interception of system

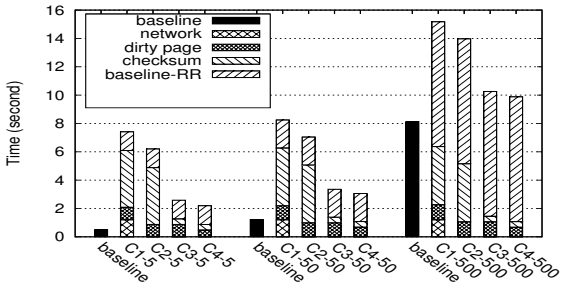


Figure 4. Analysis of the overheads for dirty memory pages between reads

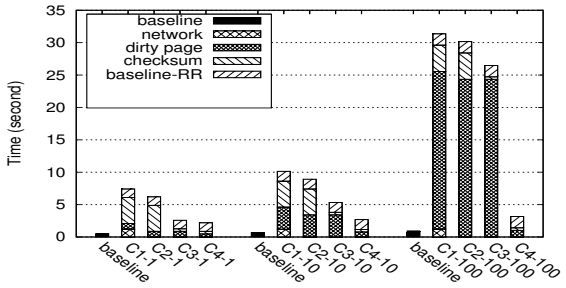


Figure 5. Analysis of the overhead for memory pool sizes (MB)

call by RR system. The overhead of *CARP* is acceptable in the real applications which is shown in Section 4.2.

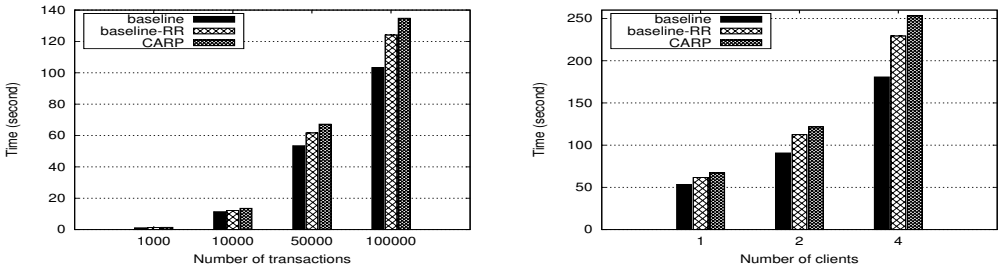
4.1.2. Effect of Dirty Memory Pages. The benchmark performs operations which generate dirty virtual memory pages between successive reads. With an increase in the ratio of dirty memory page operations between reads, the relative number of file system calls in the benchmark is lower, leading to a lower overhead of checkpointing, but a higher overhead of dirty page tracking. Both the *baseline-RR* and *baseline* execution time increase because the number of memory operations increases, leading to increase of total benchmark processing time. As Figure 4 shows, with an increase in the ratio of dirty memory pages generated between reads from 5 to 500, the overhead of all the *CARP* variants decreases accordingly. When the ratio of dirty memory page generated between reads is 5, the execution time of *C1* is 5.59 times that of the *baseline-RR*, while that of *C4* is 1.66 times that of the *baseline-RR*. When the ratio is 500, the execution time of *C1* is 1.72 times that of the *baseline-RR*, while that of *C4* is only 1.12 times that of the *baseline-RR*. For comparison of the *baseline*, for the ratio of 500, the overhead of *C4* is 21.3% of the *baseline*. So, if the application contains a small percentage of file system calls, the overhead of *CARP* over the *baseline* can be within an acceptable range. The reason why the dirty page tracking overhead does not increase significantly is that the memory pool size is only 1 MB and the dirty memory page operations will touch nearly all the memory pages of the pool when the ratio of dirty pages generated between read operations is 5. When this ratio is 500, a similar phenomenon occurs resulting in similar overhead. Performance analysis with various memory pool sizes is presented next.

4.1.3. Effects of Memory Pool Size. We measured the sensitivity of the *CARP* variants as the memory pool size of the benchmark is varied from 1 MB to 100 MB. This experiment has the potential to compare the virtual memory tracking mechanisms as the dirty page tracking overhead is proportional to memory pool size given a fixed ratio of dirty memory page operations generated between reads. As Figure 5 shows, the execution time of *C1* grows by a factor of 4 while that of *C4* grows by a factor of 1.5 in this experiment reflecting lower overhead for the virtual memory tracking mechanism involving page table entries. We can see that the dirty page tracking overhead increases significantly with a larger memory pool size for the virtual memory tracking method involving page fault handling [2]. When the memory pool size changes from 1 MB to 100 MB, the dirty page tracking overhead of *C1* to *C3* grows by a factor of about 25, while the dirty page tracking overhead of *C4* only increases by a factor of 2, reflecting a lower overhead of the virtual memory tracking mechanism involving page table entries. Compared with the *baseline*, the execution time of *C4* is 3.6 times of the *baseline*, since this experiment is also system call intensive (the ratio of dirty memory page operations between reads is 5).

4.1.4. Micro-benchmark Overhead Summary. In this section, the overhead of the most optimized *CARP* variant *C4* has an average of 26% and a minimum of 12.0% over the *baseline-RR*. For system call intensive experiments, the execution time of *C4* is very high compared with the *baseline* as the experiments shown above. While this gives us an initial estimate under a large spectrum of parameters and provided insight for optimizations in *CARP*, testing with real life applications gives a better idea of the overhead in practical situations. For disk space overhead, the size of *write_operations_log* for each file is only 0.3% of the file size because there is a log entry for each file page in the *write_operation_log* for its checksum. For memory usage overhead, *CARP* maintains a hash table to store the *write_operations_log* in memory for fast access. In this micro benchmark, for a file size of 1 MB, the extra memory usage is 3 KB, which is a small percentage of the memory used by the benchmark.

4.2. Macro-benchmark Evaluation

As mentioned earlier, an evaluation of *CARP* in several representative real applications will give a better insight into the overheads. In that vein, we chose three benchmarks: Postmark file system I/O benchmark, PostgreSQL



(a) Variable number of transactions (x-axis) (b) Variable number of clients (x-axis)

Figure 6. The execution time of the baseline and *CARP* for PostgreSQL

database benchmark and Apache web server benchmark. We used the most optimized variant of *CARP* (*C4*) with the default system parameter of 50 as the checkpoint interval, and compared it to the baseline (bare machine) and the *baseline-RR* (RR program replication system) with respect to execution overhead. We also calculated recovery times by measuring the transfer time of sending dirty memory pages over a long-distance gigabit link when a silent data error is encountered; the experiment also validated that the program behaves correctly after recovery. Finally, we measured disk and memory overhead of *CARP*, but they are not reported as the overhead is minimal ($< 5\%$). Due to space limitations, the results of the Postmark benchmark are not shown here.

4.2.1. PostgreSQL. PostgreSQL [13] is an object-relational client-server database management system that supports standard SQL transactions. The server process manages the database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the clients. The *pgbench* benchmark is a client application of PostgreSQL, which runs queries similar to those of the TPC-B benchmark on the database.

CARP monitors the server process of PostgreSQL to intercept the read and write operations to the database files. In Figure 6(a), we varied the number of transactions from 1000 to 100,000 for one client. The execution overhead of *CARP* is 8.6% to 11.1% slower than that of the *baseline-RR*, while 18.6% to 30.4% slower than that of the *baseline*. The average dirty memory pages generated in each checkpoint is 8.3% of the total.

To test the overhead of *CARP* in a multi-client environment, we varied the number of clients from 1 to 4 with 50,000 transactions for each client as seen in Figure 6(b). The overhead of *CARP* ranges from 8.1% to 10.4% over the *baseline-RR* and 25.7% to 40% over the *baseline* in this experiment. The number of average dirty memory pages generated for each process is 6.1% of the total. The results above show that the execution overhead and recovery times of *CARP* meets our goals in the *pgbench* benchmark.

4.2.2. Apache Httpd. We are also interested in the behavior of *CARP* in web server applications. The Apache HTTP Server [14] accepts HTTP requests from the clients and sends back the requested data. The *httperf* benchmark [15] measures web server performance by generating web requests and measuring the resultant response rates.

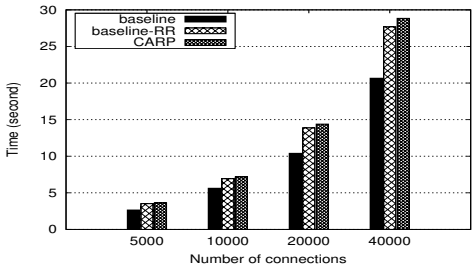


Figure 7. The execution time of the *baseline* and *CARP* for *httperf*

In the experiment illustrated in Figure 7, the number of HTTP requests from one client to the web server was varied from 5,000 to 40,000. The overhead of *CARP* ranges from 3.4% to 4.2% over that of the *baseline-RR* and 37% to 40% over the *baseline*. Since the HTTP workload involves intensive TCP/IP message processing, only a small percentage of time is for the I/O processing, leading to a smaller overhead of *CARP* over the *baseline-RR* in *httperf* compared to that in Postmark and PostgreSQL. The number of average dirty pages for each Apache server process is 6.3% of that of the total. Consequently, we achieve low recovery times for *CARP* in the web benchmark.

5. Related Work

A lot of work has been devoted to the development of checkpointing techniques for rollback recovery in program replication. Remus [2] describes a system that incrementally checkpoints the virtual machine state at a high frequency between the primary host and a secondary host. Outputs of the virtual machine are buffered until the next checkpoint is committed to the passive backup machine. These checkpointing mechanisms only focus on program replication and replay to handle the site failures, but cannot detect or correct silent data errors of storage systems. Byzantine fault-tolerant systems [16] use state machines for program or storage replication, requiring the program or storage to either be deterministic, or to only support limited nondeterminism.

However, *CARP* implements integrated program and storage replication and supports both program and storage-level nondeterminism such as multi-threading and silent data errors.

To address silent data errors, the T10 standards committee has proposed the SCSI Block Command Standard-3 [17] as an enhancement to the SCSI protocol. However, the proposal does not cover the detection of all types of silent data errors such as dropped writes [18]. Teradata's Database System [19] replicates the DBMS on a cluster of failback mirrors. It periodically compares the primary and failback copies of data, reports any discrepancies, and reconstructs the corrupted copy from the good replica. Tandem's Non-Stop Server [1] provides fault tolerance of enterprise-class applications using a combination of hardware and software solutions. They reinforce the system through component redundancy and extensive error checking using both transparent and non-transparent mechanisms. All of the above mechanism are based on high end hardware or software systems, such as advanced disk arrays or expensive database clusters. In contrast, *CARP* can be deployed on systems running on commodity hardware and can handle a wider range of applications.

Several commercial file systems handle silent data errors. Sun's ZFS [20] and Google's GFS [21] implement end-to-end checksumming of all data and constantly verified data correction on each read operation. They recover the corrupted data from a replica copy. *CARP* also uses a checksum mechanism and a third copy to detect and recover from silent errors. However, *CARP* addresses a broader and more complex problem than file systems do, that of silent data error detection and correction in the context of program replication. It implements a more efficient error detection scheme that does not incur the overhead of checksum computation for every data read, and leverages saved program state in both replicas to enable efficient recovery when a silent error is detected.

6. Conclusions

This paper presents *CARP*, a transparent low-overhead integrated program and storage replication solution that provides efficient detection and recovery from silent data errors. *CARP* addresses a problem that is fundamentally broader in scope than that of file system and database management schemes that handle silent error detection and correction. We implemented *CARP* on Linux and demonstrated acceptable overhead and quick recovery times ($<$ 1s) with unmodified email, database and web benchmarks on detecting a silent data error.

References

[1] W. Bartlett and L. Spainhower, "Commercial Fault Tolerance: A Tale of Two Systems," in *IEEE Transactions on Dependable and Secure Computing*, Jan. 2004.

[2] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *NSDI*, 2008.

[3] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure Trends in A Large Disk Drive Population," in *FAST*, 2007.

[4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An Analysis of Latent Sector Errors in Disk Drives," in *SIGMETRICS*, 2007.

[5] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea, "An Analysis of Data Corruption in the Storage Stack," in *FAST*, 2008.

[6] B. Panzer-Steindel, "Data Integrity, CERN IT Group," 2007.

[7] P. Bergheaud, D. Subhraveti, and M. Vertes, "Fault Tolerance in Multiprocessor Systems Via Application Cloning," in *ICDCS*, 2007.

[8] C. Basile, Z. Kalbarczyk, and R. Iyer, "A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas," in *DSN*, 2003.

[9] M. Russinovich and B. Cogswell, "Replay for Concurrent Non-deterministic Shared-memory Applications," in *PLDI*, 1996.

[10] J. Orcutt, "Data replication strategies," 2005, sun Microsystems White Paper.

[11] S. R. King, F. Berry, and M. E. Kounavis, "Performing A Cyclic Redundancy Checksum Operation Responsive to A User-level Instruction," Intel Corporation Patent, 2007.

[12] "Reliable Scalable Cluster Technology (RSCT)," www.redbooks.ibm.com/abstracts/tips0090.html?Open.

[13] "PostgreSQL," http://www.postgresql.org.

[14] "The Apache HTTP Server Project," http://httpd.apache.org.

[15] D. Mosberger and T. Jin, "httpperf: A tool for measuring web server performance," *Performance Evaluation Review*, 1998.

[16] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *OSDI*, 1999.

[17] M. Evans, "Information Technology-SCSI Block Commands-3(SBC-3)," http://www.t10.org/cgi-bin/ac.pl?t=f&f=sbc3r17.pdf, 2008.

[18] J. L. Hafner, V. Deenadhayalan, W. Belluomini, and K. Rao, "Undetected Disk Errors in RAID Arrays," in *IBM Journal of Research and Development. VOL.52*, 2008.

[19] J. Dietz and L. Hedegard, "The Benefits of Enabling Fallback in The Active Data Warehouse," www.teradata.com/tdmo/v07n01/pdf/AR5201.pdf, 2007.

[20] J. Bonwick, "ZFS," in *LISA*, 2007.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP*, 2003.