Using Remote Memory to Stabilise Data Efficiently on an EXT2 Linux File System

Francisco Brasileiro, Walfredo Cirne, Erick Passos, and Tatiana Stanchi

Universidade Federal da Paraíba
Coordenação de Pós-Graduação em Informática
Av. Aprígio Veloso, s/n, Bodocongó
58109-970, Campina Grande, PB, Brazil
Tel: (+55) 83 310 1123 Fax: (+55) 83 310 1124
{fubica,walfredo,edric}@dsc.ufpb.br tatiss@svn.com.br

Abstract

Stable storage is an important requirement for many applications. It is usually implemented over traditional file systems via synchronous write operations to disk. However, due to the small throughput and high latency of disks, writing data synchronously decreases the performance of applications. A workaround is to use UPS devices to prevent data cached in volatile memory from being lost after the occurrence of system software or power supply failures, obviating the need for synchronous write operations. This strategy, however, may not tolerate hardware failures, *i.e.* processor, memory, controllers, etc. This paper presents an implementation of stable storage over a file system that substitutes synchronous write operations by remote replication of data locally cached, therefore removing slow disk operations from the critical path of storing data in stable storage. The implementation does not rely on any special hardware and is resilient to both hardware and system software faults. Our preliminary performance evaluation has shown that replicating data remotely can be nearly 11 times faster than writing data synchronously to disk.

Keywords: Stable storage; replication; file system recovery; fault tolerance; Linux.

1 Introduction

Many fault-tolerant applications require the availability of part of their state after a failure occurs. For these applications, stable storage is a fundamental building block [Jal94]. Stable storage offers a service that guarantees that data stored is not destroyed or corrupted by failures. Checkpointing [Jal94] and transactions [Jal94] are two classical examples of fault tolerance mechanisms that are normally implemented with the support of a stable storage service.

Due to their non-volatility, disks are the preferred devices to implement stable storage. Although disks are reasonably reliable, more demanding applications might require stable storage to be implemented over redundant disks [PGK88], or via carefully replicated operations over a single disk [Jal94]. For the sake of simplicity, in this paper we assume that disks are reliable enough to fulfill the dependability requirements of applications. Nevertheless, our system can be easily adapted to a scenario where disks and write operations are replicated to tolerate disk faults and data corruption. Our main concern, therefore, is with the performance gains that the file system implementation presented in this paper brings to storing data in a stable way.

Over the years disk technologies have evolved faster in increasing storage capacity than in reducing access time. As a result, disks remain rather slow devices, specially when their throughput and latency are compared to that achieved by other devices such as main memory, processors and networks. Thus, the performance of current file systems is largely influenced by caching mechanisms.

Caching improves performance by maintaining in main memory information that is likely to be accessed in the near future. However, unless UPS (Uninterruptible Power Supply) devices [APC96] are used, information kept in main memory will be lost in the event of a power supply failure. If data kept in memory has not yet been updated to disk, then inconsistencies arise. Further, even when UPS devices are available, hardware failures and system software bugs may also be sources of unreliability for the operation of the file system. This is because, when such failures occur, it might not be possible to perform a controlled warm re-boot that preserves the contents of main memory [CNC+96, CS01].

To overcome the unreliability problem discussed above, most file systems provide more reliable operation modes at the expenses of a decrease in the performance. Normally this is implemented by a mechanism that bypasses the caching mechanism whenever an operation that changes the state of the file system is performed. That is, by performing a synchronous write operation directly to the disk whenever there is a state change.

While disk speed has been improving slowly, network speed keeps advancing at a fast pace. This state of affairs has prompted many researchers into investigating how memory of remote machines can be used to improve file system performance, a trend that has been encouraged by evidences that ample amounts of memory are available in a typical local network [AS99]. In this paper we present an implementation of a stable storage service, based on the Linux EXT2 file system. It uses replication protocols that allow local data to be reliably stored at remote hosts [BCPS01]. By taking the disk I/O out of the critical path of writing to stable storage, performance can be increased [LGG+91, PLP98, IMS98].

The remaining of the paper is structured in the following way. Section 2 discusses related work on the use of remote memory and on the implementation of efficient stable storage services. In Section 3 we present the Salius approach to stable storage, and discuss the general behavior of the replication protocols that are behind the stable storage service proposed. Section 4 is devoted to the discussion of the implementation of a Salius file system based on a Linux EXT2 file system. Then, in Section 5 we analyse the performance of the prototype implemented. Finally, Section 6 concludes the paper with our final remarks.

2 Related Work

A significant part of the work done so far uses remote memory to enhance the reading cache of a file system, by allowing requests not satisfied by a machine's local cache to be satisfied by the cache of another machine, in what has been termed *cooperative caching* [DWAP94, SH96, KPR99]. Dahlin *et al.* propose various schemes to perform cooperative caching and use trace-driven simulations to investigate the performance of such schemes [DWAP94]. Sarkar and Hartman propose a decentralized algorithm for cooperative caching based on inexact information (hints) that provides performance comparable to that of existing centralized algorithms [SH96]. Korupolu *et al.* explore algorithms for cooperative caching in a hierarchical network under some assumptions for file access and machine characteristics [KPR99]. Since a Salius file system focus in the write performance of a reliable file system, the work in cooperative caching is complementary to our own.

Distributed file systems are another line of research that explores remote memory as means to improve file system performance [FMP+95, ADN+95, HO95]. Distributed file systems are a replacement for traditional file system, being often implemented as part of a distributed operating system. The major advantages are the low overhead and the ability to integrate paging into the file system design, allowing for fast virtual memory, which is backed by remote memory. The main disadvantage is that distributed file systems typically do not provide support for stable storage. Rather, they use the same delayed-write technique of traditional file systems, trading therefore reliability for performance.

Harp [LGG+91] is close in spirit to a Salius file system. Harp, however, is more concerned in providing high availability through the replication of the file system, leading to more complex and costly algorithms.

The work by Plank, Li and Puening on diskless checkpointing [PLP98] and that by Ioannidis, Markatos, and Sevaslidou on transaction-based systems [IMS98] also use remote memory to avoid the performance penalty of writing data to disk. A Salius file system uses remote memory available in the local network to provide efficient and highly-reliable stable storage. Our approach is to provide a more generic service upon which other higher level services, such as conventional checkpointing and transactional services, can be implemented without the performance burden normally associated to them.

A different approach to increase the performance of stable storage is to use UPS devices to preserve data stored in main memory after a crash, eliminating the need of synchronously writing data to disk. In this way, stable storage can be totally [CS01] or partially [CNC+96] implemented in main memory. The Rio file cache [CNC+96] uses a sophisticated mechanism that prevents the main memory from being reset after a warm re-boot and then restores the file system via the information that has been kept safe in memory. Cunha and Silva [CS01] follow a similar approach to implement a diskless stable storage service for embedded systems. However, in contrast with the Salius approach, these approaches do not tolerate failures that require an off-line maintenance procedure to be executed.

3 The Salius Approach to Stable Storage

As mentioned in the Introduction, most file systems are able to operate in two different modes which enable the application to trade in performance for reliability. In the best performing mode, a caching mechanism implemented in the volatile main memory is used to temporarily store data that are latter written to disk asynchronously. To increase reliability, an alternative operation mode is used, which forces data to be synchronously written to disk. Figure 1 illustrates these operation modes.

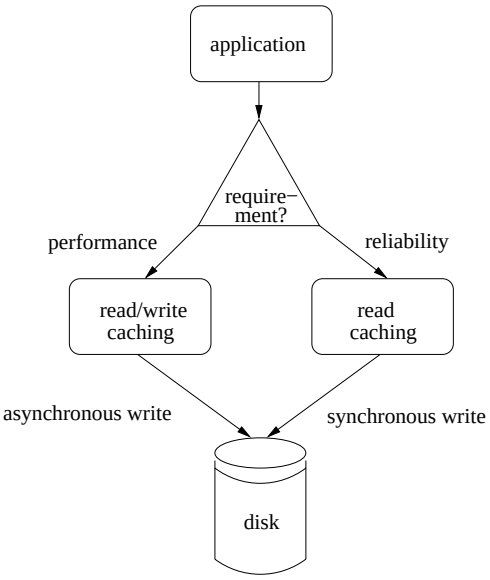

Figure 1: Operation Modes of a Conventional File System

The Salius approach to provide reliable semantics is based on the replication of data that are stored in the volatile cache of the machine that hosts the file system (from now on we will use the expression *local host* when referring to this machine) in the memory of remote machines that fail independently (see Figure 2). This procedure obviates the need to perform synchronous write operations.

The *Data Replication Service* (DRS) is the core of a Salius file system. It implements the protocol which guarantees that enough information will be stored remotely, so that, in the event of a crash of the local host, the state of the file system that was stored in the cache and had not been stabilised in disk will be successfully restored.

Due to the higher throughput and smaller latency of networks, and the fact that a Salius file system is also able to use a read/write caching mechanism, reliability can be achieved with a much smaller performance penalty, when compared to the conventional strategy based on writing data synchronously to disk (we will discuss this issue in detail in Section 5).

We say that a Salius file system has crashed when the information that is kept in the caching mechanism of the local host is permanently lost. This can be caused by events such as power supply failures, processor crashes, operating system hangs, etc. When a Salius file system crashes the following procedure must be executed: first the local host has to be

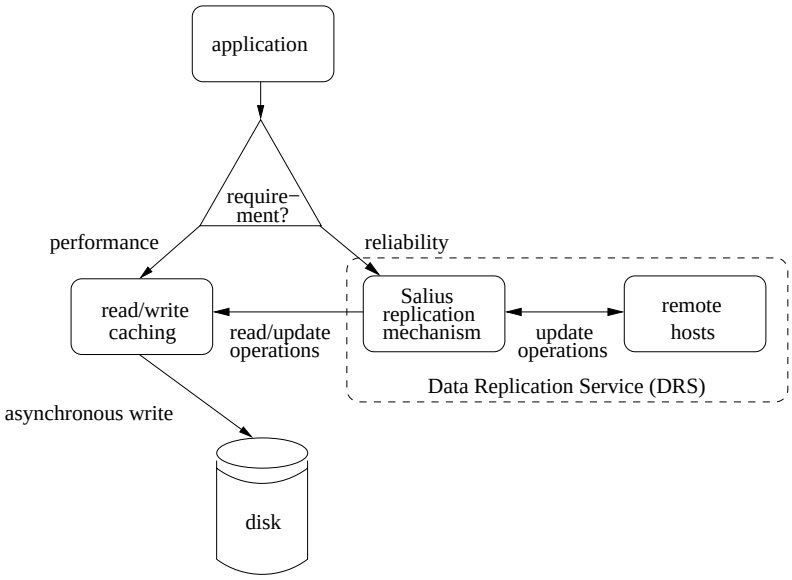

Figure 2: Operation Modes of a Salius File System

brought back into operation (*e.g.* via a re-boot); and then a *Disk Recovery Procedure* (DRP) is executed in the local host. The DRP implements a protocol that allows it to obtain from the DRS the information required to restore the crashed file system. Thus, implementing a Salius file system requires the implementation of both a DRS and a DRP.

In the above discussion we have implicitly assumed that the DRS is reliable and always available. Obviously, if all remote machines that implement the DRS crash at the same time that a Salius file system crashes, then it might be impossible to recover the file system. Likewise, if the network experiences a partition preventing the local host from communicating with the remote machines that implement the DRS, then a reliable file system operation might be delayed for a long time, until the partition heals. Notice that the part of the DRS that executes on the local host will only allow the reliable file system operation to return to the application when it has a guarantee that enough remote hosts have received the data sent.

A reliable and available implementation of the DRS can only be achieved when the network does not partition and the number of replicas is high enough so that the probability of all replicas crashing while the file systems crashes is negligibly small [Cri91]. For most applications a relatively small number of replicas (up to 3) is enough to guarantee their dependability requirements. On the other hand, partition-free networks are rare. Thus, to avoid long delays caused by partitions, the part of the DRS that executes in the local host must detect network partitions and circumvent this problem. When a network partition is detected all cached data that have been modified and have not been written to disk must be synchronously written to disk. Any subsequent update operation executed while the network is partitioned must be performed by a synchronous write operation to disk. Since network partitions are uncommon events, performance is not too harmed by the infrequent synchronous write operations they cause. Figure 3 illustrates the structure of a Salius file

system that is able to handle network partitions.

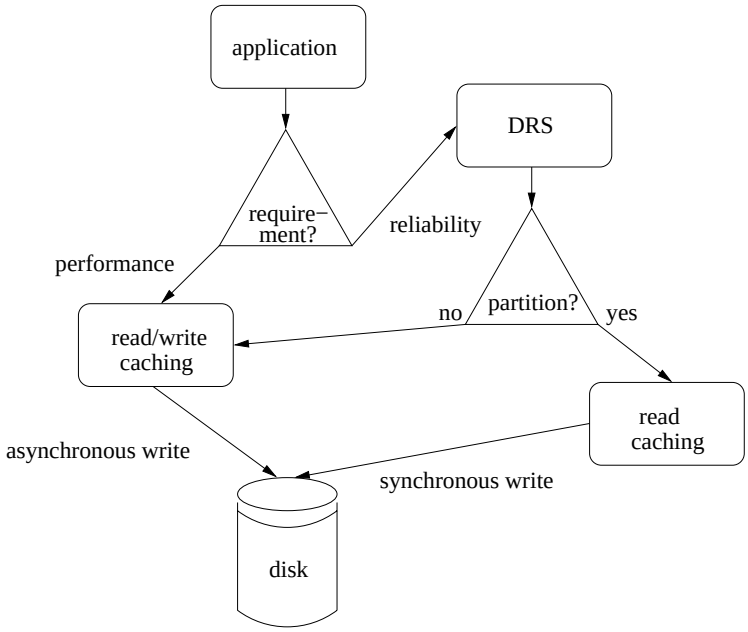

Figure 3: Operation Modes of a Partition-Aware Salius File System

The execution of the DRP after a crash of the file system must guarantee that the file system stored in disk will be left in a consistent state. This is to say that the data manipulated by all successfully completed operations executed through the reliable interface of the file system will be eventually stored in disk in a stable way.

Whenever a reliable file system operation that changes the state of the file system is executed, then data are sent to the remote hosts that implement the reliable DRS. These data must carry enough information so that, if required, the DRP will be able to leave the file system in a state that is equivalent to what would be achieved had the operations been synchronously written to disk at the time they were issued.

Notice that the recovery procedure must be idempotent. Thus, even if the local host crashes during the execution of the DRP, the procedure can be restarted as many times as required, and the final result achieved by a successful execution of the DRP will be the same no matter how many times it needed to be restarted.

4 An Implementation of a Salius File System

In the previous section we have outlined the main requirements to implement the DRS and DRP of a Salius file system. Although a Salius file system can be implemented from scratch, we believe that the most effective way to attain such a file system is to modify an existing one by introducing a suitable DRS and developing a matching DRP. If the original file system already provides a reliable operation mode, then its interface need not be changed.

In [BCPS01] we have introduced the Salius service for stable storage and, as a proof of concept, implemented it at the library level. That approach, however, has the disadvantage of the bigger overhead of accessing the network at user level and all the penalties imposed by the scheduler. In this paper we present a Salius implementation based on the Linux EXT2 file system. We conduct such implementation at the kernel level, *i.e.* we implement a clone of the Linux EXT2 file system augmenting it with the extra functionalities of the DRS. To use such functionalities one can mount any EXT2 partition as a Salius file system by executing the Linux *mount* command specifying the file system type with the option *-t salius*.

The modifications in the VFS module of the kernel to register the Salius file system required less than 50 lines of code. This modifications were necessary to: i) set general assembly macros for bit operations shared by some file systems implementations; iii) initialize the Salius module; and iii) bind the socket used for messages transmission. The Salius file system is based on the EXT2 implementation found in late 2.2.x Linux kernels. To this base implementation we have added extra code to control the file access mode, send replication messages, receive acknowledgments and validate them. This implementation required approximately 300 extra lines of C code.

4.1 System Model

Our target system environment is formed by a local area network of machines executing the Linux operating system. Thus, we assume a system model based on the timed asynchronous system model [CF99]. In the context of this paper, such model comprehends the following assumptions:

Assumption 1: processors fail only by crashing[1];

Assumption 2: correct processors have access to a datagram communication service that does not corrupt messages, in other words, messages can only be arbitrarily delayed, lost or duplicated;

Assumption 3: correct processors have access to physical clocks that advance within a linear envelope of real time and that can be used to measure time-outs;

Assumption 4: there is an unreliable bound δ used to define performance failures for communication links.

The DRS that allows the recovery of a crashed file system is implemented by n processors, from which, one is the local host. We assume that only f processors can fail and that $f < n$. Thus, in the worst case, when $n = f + 1$, if all remote hosts fail, then the file system will not crash; on the other hand, if the file system crashes, then at least one remote host will not fail and the information it holds will be used by the DRP to restore the crashed file system.

Since δ is an unreliable bound, the message transmission delay between two correct processors can be arbitrarily large. However, δ is chosen in such a way that, most of the

[1]As indicated before, a processor is considered crashed, when the information that it holds in main memory is permanently lost; this in turn can be caused by both hardware and system software faults.

time, message transmission delays between correct processors fall within δ. We say that the local host is partitioned from a processor P when the round trip message transmission delay between the local host and P exceeds $2 * \delta$. Thus, the local host can use time-outs to detect partitions.

4.2 The Interface

The Linux kernel offers a generic, POSIX compliant, file system interface called VFS (Virtual File System). Each file system implementation contains specific code which is accessed via the VFS interface. The Salius file system offers reliable operation mode via two functions, namely: *salius_open_file*, and *salius_file_write*. The VFS interface directs every call to the open and write functions to the *salius_open_file* and *salius_file_write* functions respectively. In the following, we briefly describe these functions.

fd = salius_open_file(struct inode *inode, struct file *fp) This function is used to open an existing file or to create and open a new one. The function returns a file descriptor that can be later used to issue other operations to the file that has been opened. The struct *inode* specifies the disk i-node allocated to the file being opened. The parameter *flags* in this i-node holds information that is used to control the operation of the function. In particular, if the *O_SYNC* bit of the parameter *flags* is set, then future write operations issued to the file associated to the *fp* structure must be carried out in a stable way. Two other bits of the *flags* parameter are important for the reliability of the recovery operation: *O_CREAT* and *O_TRUNC*. When the *O_CREAT* flag is set, the file should be created if it does not already exist. If the file already exists, then this flag is simply ignored. Also, when the *O_TRUNC* flag is set, the file should be truncated. Finally, the parameter *mode* of the i-node is only used when the flag *O_CREAT* is set; it is used to specify the access permissions associated to the file that is being created.

written = salius_file_write(struct file *fp, const char *buf, size_t n, loff_t *p) This function is used to write the "*n*" bytes stored in the memory positions starting at address "*buf*" to the file associated to the file structure pointed by *fp*. The function returns the number of bytes that have been effectively written. When the struct *fp* is associated to a file that has been opened with the *O_SYNC* flag set, the *replicated* flag indicates that the writing of data must be carried out in a stable way. In the standard implementation of the EXT2 file system, this is achieved via a synchronous write operation to disk, *i.e.* the blocks associated to that file in the buffer cache are marked as dirty and then stored on disk. Our Salius implementation uses the non-reliable operation mode of the EXT2 file system to increase performance and data replication to achieve reliability.

4.3 The DRS

The functionalities of the DRS are implemented by the Salius file system and thus run in the kernel. The replication servers execute as daemons at each of the remote hosts.

Whenever a call to the *open* function with the *O_SYNC* flag set is performed over a disk partition mounted as a Salius file system, all subsequent calls to the write function will use the DRS functionalities to achieve stable storage.

All invocations of the *write* function will yield the following steps to be taken. First, the write operation is performed asynchronously. Note that if the write operation cannot be performed (for instance, because the disk is full), then a negative number is returned. Moreover, if the file has been opened with the *O_SYNC* flag set, then a message is sent to the remote hosts. This message contains the following fields:

***time-stamp*:** which contains the current reading of the local physical clock;

***file*:** which contains the *umask*, *uId*, *gId*, *pathname*, *flags*, and *mode* fields of the entry of the table of stable files associated with the pair (calling process identification, file descriptor passed as parameter to the *write* function);

***nbytes*:** which contains the number of bytes actually written; and

***bytes*:** which contains the sequence of bytes that has actually been written; and

***offset*:** which contains the read/write offset of the file, indicating the point within the file from where the bytes started to be written.

Then, the bit *O_TRUNC* of the field *flags* of the file struct inside the kernel address space is unset (as will be seen shortly, this avoids the file being wrongly truncated during a recovery). Finally, the system call only returns after one of the following two conditions is met: i) f remote hosts have acknowledged the reception of the message; or ii) a $2 * \delta$ time-out has expired. In the latter case, just before returning, a synchronisation operation is executed to clear all dirty buffer related to this file on the cache to avoid loosing data.

If the local host fails, then it is guaranteed that no more than $f - 1$ remote hosts will fail. By waiting for the "ack" of f remote hosts, the local host guarantees that in the case of a file system crash, at least one correct remote host will hold a copy of every reliable operation performed. In the event of a failure all operations can be retrieved, and by using the associated time-stamp, the DRP can re-execute them in the correct order.

Since partitions and faults are rare, most of the time reliable invocations of the *write* function will return after the reception of f "acks", *i.e.* the time-out will seldom expire. Further, since messages are always sent to the remote replication servers, as soon as a partition heals, synchronous write operations to disk will stop being issued.

On the other hand, if n is set to its minimal value, *i.e.* $n = f + 1$, then after the failure of a single remote host, all subsequent invocations of reliable functions will require synchronous write operations to disk. If hosts do not recover from failures, or if this recovery can take an arbitrarily long time, the performance penalty depicted in the above scenario can be eliminated by increasing the number of replicas. In particular, if n is set to be greater or equal to $2 * f + 1$, then synchronous write operations will only be required when a network partition occur.

The functioning of the remote replication servers is very simple. They start an infinite loop waiting for messages sent by the local host. Three types of messages are possible: i)

replication messages; ii) start recovery messages; and iii) re-transmission recovery message. Whenever a replication message is received, an "ack" message containing the time-stamp of the received message is sent back. A start recovery message is replied with a message that contains the number of replication messages currently stored in the local log. Finally, a re-transmission recovery message is replied with a copy of the corresponding message stored in the log. (We detail the recovery procedure in the next subsection.)

To keep the log of the remote replication servers finite, the following mechanism is used. Whenever a replication message is received, the replication server reads its local clock and adds a second time-stamps to the message. Let us call this time-stamp the reception time. Every message that has a reception time-stamp that is $k * \tau$ units of time smaller than the largest reception time-stamp issued, can be discarded, where τ is the maximum time interval that a "dirty block"[2] can stay in the cache of the local host before being written to disk (for many implementation of EXT2 file systems, the default value for this time interval is 30 seconds). The k coefficient is used to compensate clock drifts, disk latency and any transient unavailability of the kernel to write a dirty block to disk within the τ bound (due to a burst of unexpected interruptions, for example). In our prototype implementation we have used $k = 2$.

4.4 The DRP

The DRP works in a very simple way. The first thing it does is to send to all remote hosts a recovery message. Then it waits for $n - f$ replies. (Remember that every reply contains the number of replication messages stored by the corresponding remote host.) After that, it sends as many re-transmission recovery messages as needed to receive all replication messages stored by the $n - f$ remote hosts that replied to its recovery message. It uses the time-stamp field of the replication messages received to detect and discard any duplicated message received. Finally, the remaining messages are ordered in increasing order of time-stamps and their corresponding operations are re-executed in that order.

As presented in the previous subsection, the operation contained in a replication message is characterised by the following fields: *file*, *nbytes*, *bytes*, and *offset*. Furthermore, the *file* field is formed by the following sub-fields: *umask*, *uId*, *gId*, *pathname*, *flags*, and *mode*. An operation is re-executed by executing the following calls in succession:
umask(*file.umask*)
fd = *open*(*file.pathname*, *file.flags*, *file.mode*)
lseek(*fd*, *offset*, *SEEK_SET*)
write(*fd*, *bytes*, *nbytes*)
fchown(*fd*, *file.uId*, *file.gId*)
fchmod(*fd*, *file.mode*)
close(*fd*).

Note that, the *umask*, *fchown*, *fchmod* and *close* operations are naturally idempotent. Likewise, open operations are also idempotent, as the *O_CREAT* flag is ignored when an open operation with the *O_CREAT* flag set is executed over a file that already exists and the

[2] A "dirty block", is a block that contain data that has been changed, but has not yet been written to disk.

O_TRUNC field may only be set in the first write operation executed on a file that has been opened with the *O_TRUNC* flag set. Also, since the *lseek* operation is always executed with an absolute value for the current offset, its execution is idempotent. Further, since there are no read operations during the recovery procedure, write operations are also idempotent. In summary, as each individual operation executed during the recovery procedure is idempotent, the whole execution of the DRP is idempotent.

5 Performance Evaluation

5.1 Description of the Environment

Our experiments were executed on a local area network of PCs. The PCs were connected via an isolated 100 Mbps shared Ethernet bus. Each PC has the following configuration: a Pentium III, 800 MHz processor; 192 Mbytes of RAM with average access time of 5 ns; and a 20 Gbytes IDE disk with average access time of 5.5 ms.

Each PC was executing the Linux 2.2 kernel. No other network services were executed by the PCs, which only run the evaluation applications. To artificially control the load on the network we used the version 0.0.1 of the traffic load generator[3].

5.2 Description of the Experiments

To evaluate the performance of our Salius implementation we used the version 3.71 of the ioZone benchmarking application[4]. We measured the performance of synchronous writes and rewrites of data using both conventional EXT2 and Salius implementation with variable record sizes.

The ioZone is a micro-benchmark tool that measures the time spent by each individual file system operation and delivers a summarized report of the results. The write test measures the cost of stable storage of a new file as well as the encapsulated cost of disk block allocation for synchronous operations. The rewrite test uses a file with all the necessary blocks already allocated and tries to overwrite data on it avoiding the performance penalties suffered by the EXT2 implementation on the previous test. For this reason we expect Salius to achieve its best performance gains over EXT2 in write operations.

Before executing the application, a brand new EXT2 file system was created. Thus, the application was always executed to create a file in a file system with the same initial state; in particular, at the beginning of the application execution the fragmentation percentage of the file system was 0%.

We executed two sets of experiments. In the first set, the network load was kept to a minimum, since no other application, but the Salius file system, used the network. In the second set, a network traffic generator was used to artificially load the network. Further, we measured the performance of the file system in four different scenarios:

[3]available at http://traffic.sourceforge.net/.

[4]available at http://www.iozone.org/.

sync: In this scenario we used the standard implementation of the EXT2 file system, based on synchronous write and rewrite operations;

Salius(2,1): In this scenario we used the Salius implementation with $n = 2$ and $f = 1$;

Salius(3,2): In this scenario we used the Salius implementation with $n = 3$ and $f = 2$; and

Salius(3,1): In this scenario we used the Salius implementation with $n = 3$ and $f = 1$.

5.3 Results and Analysis

Figure 4 shows the average write throughput for an unloaded network. As we can see, increasing the number of replicas and faults to be tolerated has little impact over the performance of our implementation. This is due to the use of UDP multicast when replicating data and the small size of the acknowledgment responses from the DRS servers running on remote machines. Our current Salius implementation can be nearly 11 times faster than an EXT2 synchronous file system when writing new data into a file.

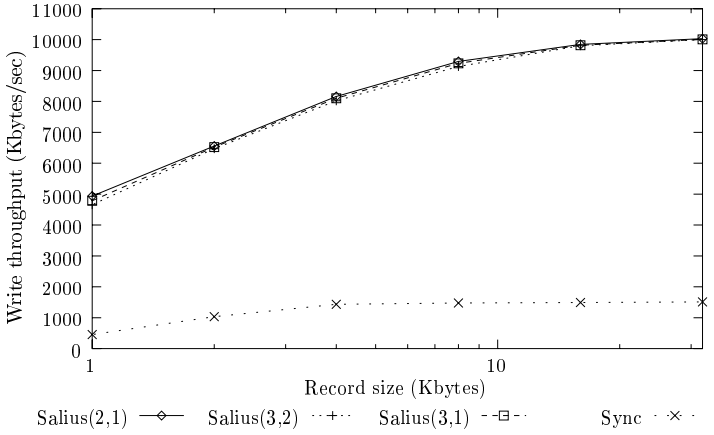

Figure 4: Average write throughput for an unloaded network

When trying to rewrite data in existent files the performance of Salius is not affected. On the other hand, the throughput of the standard implementation increases (see Figure 5). This is expected since Salius operation is virtually the same in both scenarios, while the standard implementation does not need to stabilise meta-data when rewriting data. Nevertheless, for our experiments Salius was up to 6.5 times faster.

In our final experiment we have analysed the impact of network traffic. Figure 6 shows the results for write throughput, and Figure 7 displays the results for rewrite throughput. We observe that network traffic has more impact over the performance than the increase in the number of replicas. Even though, when compared to the standard implementation of the EXT2 file system, Salius is still 5 times faster in average, in a heavily loaded network.

Figure 8 attempts to summarize our results by showing the performance figures of three scenarios: Synchronous EXT2 (the baseline), Salius (2,1) without network traffic (the best scenario for Salius), and Salius (3,2) with network traffic (the toughest scenario for Salius).

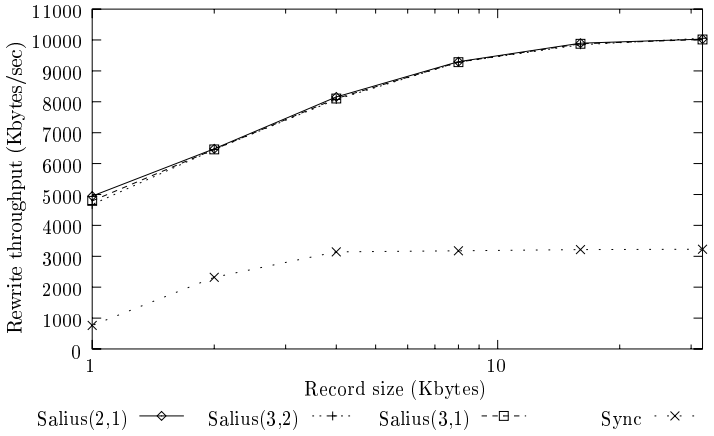

Figure 5: Average rewrite throughput for an unloaded network

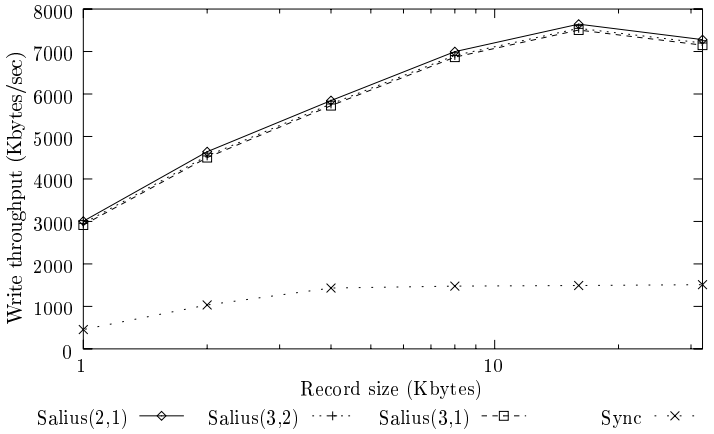

Figure 6: Average write throughput for a heavily loaded network

It is clear that Salius performs better than the traditional EXT2 synchronous operations in all tested scenarios. In our tests different file sizes have had little influence over the performance results. This is because the target partition used for write and rewrite tests is always unfragmented. In a fragmented disk, we expected synchronous operations to suffer more performance penalties than Salius because of the head positioning overhead. These penalties can be easily increased by concurrent accesses to the disk resulting in a far worse performance for conventional EXT2 file systems.

It should also be noted that the performance results change in function of the record size because of the overhead imposed by each separate operation when writing/rewriting memory buffers to disk/network. The bigger the record size, the smaller the overhead and in consequence bigger throughput is achieved.

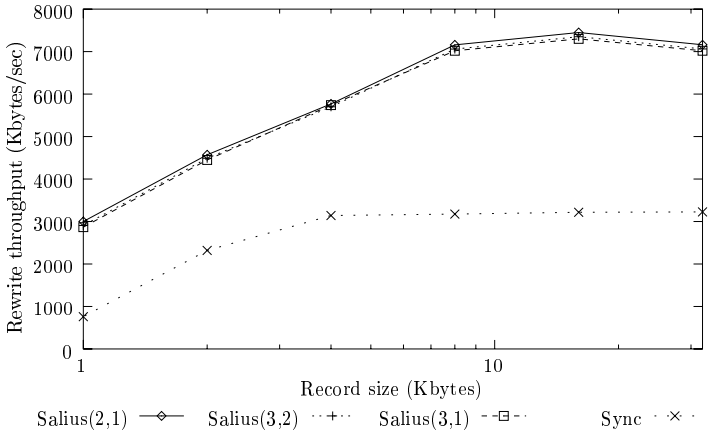

Figure 7: Average rewrite throughput for a heavily loaded network

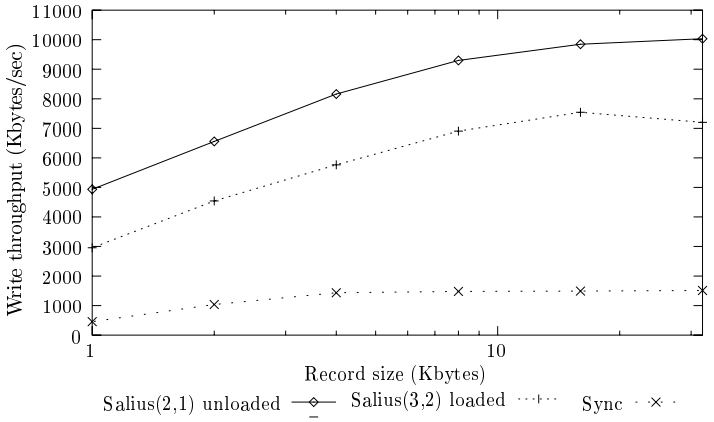

Figure 8: Average write throughput

6 Conclusion

We have presented an alternative implementation of stable storage. Instead of relying on synchronous write operations to disk, a Salius file system uses a fast network connection to replicate data at remote hosts. The information kept safe at the remote hosts can latter be used to recover a crashed file system.

Our Salius implementation based on the Linux EXT2 file system has shown that it can be nearly 11 times faster than its standard implementation based on synchronous write operations. Further, our experiments indicate that the increase in the number of replicas, and therefore the reliability of the file system, does not impact its performance substantially. Moreover, even when the network load is intense, the Salius implementation is considerably faster than its traditional counterpart.

It is important to mention that the reliable operation mode of a POSIX-compliant file system, like the Linux EXT2, only guarantees stability for the data written. Changes to the meta-data associated to a file that has been opened with the *O_SYNC* bit set are not

guaranteed to be stabilised in disk. For the sake of simplicity, our Salius implementation has kept the same semantics. However, implementing stable storage of meta-data in a Salius file system is a trivial task. It suffices to add the required data replication code at every function that changes the contents of the meta-data associated to a file.

References

[ADN+95] T. Anderson, M. D. Dahlin, J. Neefe, D. Paterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the* 15^{th} *Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, Dec 1995.

[APC96] The power protection handbook. American Power Conversion, Technical Report, 1996.

[AS99] A. Acharya and S. Setia. Availability and utility of idle memory in workstation clusters. In *Measurement and Modeling of Computer Systems*, pages 35–46, 1999.

[BCPS01] F. V. Brasileiro, W. Cirne, E. B. Passos, and T. S. Stanchi. Efficient stable storage through data replication. Technical Report UFPB/CCT/DSC/LSD, 2001. http://www.dsc.ufpb.br/~fubica/hp/publicacoes/relatorios/BCPS01.ps (submitted to publication).

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, Jun 1999.

[CNC+96] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: surviving operating system crashes. *ACM SIGPLAN Notices*, 31(9):74–83, 1996.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, Feb 1991.

[CS01] J. C. Cunha and J. G. Silva. Software-implemented stable storage in main memory. In *Brazilian Symposium on Fault Tolerance*, Florianópolis, Brazil, Mar 2001.

[DWAP94] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, Nov 1994.

[FMP+95] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Symposium on Operating Systems Principles*, pages 201–212, 1995.

[HO95] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[IMS98] S. Ioannidis, E. P. Markatos, and J. Sevaslidou. On using network memory to improve the performance of transaction-based systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[Jal94] P. Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1994.

[KPR99] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[LGG+91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of the* 13^{th} *ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, 1991.

[PGK88] D. Patterson, G. Gibson, and R. Katz. The case for RAID: Redundant arrays of inexpensive disks. In *Proceedings of the ACM SIGMOD Conference*, pages 106–113, Chicago, IL, May 1988.

[PLP98] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, Oct 1998.

[SH96] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.