Unifying File System Protection

Christopher A. Stein [1], John H. Howard [2], Margo I. Seltzer [1]

[1] *Harvard University*
[2] *Sun Microsystems*

Abstract

This paper describes an efficient and elegant architecture for unifying the meta-data protection of journaling file systems with the data integrity protection of collision-resistant cryptographic hashes. Traditional file system journaling protects the ordering of meta-data operations to maintain consistency in the presence of crashes. However, journaling does not protect important system meta-data and application data from modification or misrepresentation by faulty or malicious storage devices. With the introduction of both storage-area networking and increasingly complex storage systems into server architectures, these threats become an important concern.

This paper presents the protected file system (PFS), a file system that unifies the meta-data update protection of journaling with strong data integrity. PFS computes hashes from file system blocks and uses these hashes to later verify the correctness of their contents. Hashes are stored within a system log, apart from the blocks they describe, but potentially on the same storage system. The write-ahead logging (WAL) protocol and the file system buffer cache are used to aggregate hash writes and allow hash computations and writes to proceed in the background.

PFS does not require the sharing of secrets between the operating system and the storage system nor the deployment of any special cryptographic firmware or hardware. PFS is an end-to-end solution and will work with any block-oriented device, from a disk drive to a monolithic RAID system, without modification.

1 Introduction

For a variety of economic and management reasons, server operating systems are moving towards a looser coupling with their storage subsystems [6]. There are two important components to this change. First, the technology connecting the server operating system to the underlying storage system is changing. Busses are being replaced by networking fabrics known as storage area networks (SAN) that utilize protocols such as Fibre Channel or Gigabit Ethernet [2]. In the future we may even see IP networks within the SAN. The IETF IP Storage working group is designing a protocol for transporting block-level storage commands over IP networks [10]. This shift from protected busses to networking fabrics introduces an increased risk of malicious intrusion.

Storage Service Providers (SSPs) offer storage outsourcing services, connecting a client's application server to their storage system, which can be partitioned across several clients. In these cases, strong protection guarantees become important because the network between the server operating system and the storage system now spans commercial boundaries. The server and the storage are no longer under a single administrative control. The client will undoubtedly have a data integrity contract with the SSP, but an alarm should be raised immediately if the SSP, either intentionally or unintentionally, delivers bad data.

Second, the complexity of storage is increasing. Storage systems have evolved from direct attached hard disks to large network-attached disk arrays with internal RAIDs. These systems have complicated firmware, and many run non-trivial operating systems. This new complexity introduces software failure modes that are not described by the traditional fail-stop model of hardware.

Many problematic scenarios can arise in this new environment. For example, malicious intruders spoof blocks to the server, a software bug on the storage corrupts data, or the storage receives or transmits a different block than the one requested. The server operating system must protect itself against all such failures. Blocks containing file system meta-data are particularly important because they contain operating system state and will crash the server operating system, and consequently all applications, if they are corrupted.

This paper introduces the Protected File System (PFS). PFS is intended for server operating systems and provides strong integrity at the level of file system blocks. Block hashes are computed with a collision-resistant hash function and are subsequently used to verify block reads from the storage. PFS does not change the file system interface. Applications are free to open, close, read and write files as they always have. PFS also makes no changes to the on-disk layout of data and meta-data. This allows for backward compatability with existing data and the freedom to use existing utilities. PFS unifies block verification with the existing file system protection mechanism, the file system journal. This unification is achieved using a generic system service known as the write-ahead file system (WAFS) [19]. The WAFS stores records and provides its clients with log sequence numbers (LSNs) for their record writes. PFS uses the WAFS for both journaling and hash logging. This centralizes recovery in a single component, consolidates potentially disparate I/O streams into a single sequential stream, and avoids code duplication.

PFS uses collision-resistant hashes to protect data integrity. Collision-resistant hashes have the property that it is computationally infeasible to find two different data blocks with the same hash value. SHA-1 [4] is one popular such hash, mapping a data block of any size to a 20-byte hash value. MD-5 [17] is another, mapping to a 16-byte value.

Section 2 outlines the properties that we sought in a solution and discusses some candidate architectures. The goal is to give readers an understanding of the process we followed in order to arrive at the PFS architecture. Section 3 discusses related work. Section 4 provides an overview of the PFS architecture, and Section 5 discusses some of the more important issues in detail. Section 6 presents our performance results, and Section 7 concludes.

2 Desired Properties and Potential Solutions

At the beginning of this project, we set out the following requirements for our architecture.

- Detect all bad blocks. With a very high degree of certainty, any unauthorized block modification must be detected by the file system. This includes both data and meta-data blocks. With the same degree of certainty, the file system must detect that the storage returns a block other than the one requested.
- No changes to storage systems. No special cryptographic logic, either hardware or software, should be necessary. File system block sizes must remain a power of two so that they can be expressed efficiently in terms of storage blocks (e.g., sectors), which tend to default to a power of two smaller than traditional file system blocks.
- Straightforward to implement in both legacy and modern file systems. The on-disk structure of the file system must be maintained. The architecture must interact well with the operating system buffer cache. The popular file system interface must remain unchanged.
- Minimal requirements for local non-volatile storage. Centralizing non-volatile state on the storage has the added benefit that a failed server can be quickly replaced by another trusted server.
- High performance. Sacrifice as little performance as possible.

A first potential solution would be to compute a hash for each block and store it along with the block, either appended to it or in a special header. When the data is faulted in, the operating system computes the hash and compares it with the hash value stored alongside the block. There are several problems with this solution. First, the storage system block size must be larger than the file system block size by the size of the hash, which is 20 bytes under SHA-1 and 16 bytes for MD-5. This may rule out a large class of storage devices that lack flexible block sizes. Second, malicious intruders with knowledge of the hash function can spoof data and generate a hash, which will be verified successfully, but for incorrect data. Third, a class of storage failures are not protected against. For example, the operating system requests block X and the storage system returns block Y along with the hash appended to Y, the operating system will compute the hash from the value of Y and compare with the appended one. The two, assuming no other failure, will be the same, and the operating system will incorrectly conclude that the block is correct. Simple self-certifying blocks can solve the third of these concerns. A self-certifying block combines both the hash of the data and the block number, so if a block other than the one requested is delivered, that fact will be detected. Signed self-certifying blocks can solve the second and third of these concerns, using a private key to generate a signature so that malicious intruders cannot spoof blocks with correct hashes. However, neither simple nor signed self-certifying blocks can solve the first concern. Under both the simple and signed self-certifying block potential solutions, the storage block size must be larger than the file system block size to accommodate the block number or the hash.

A second potential solution stores the hashes with the file system pointers. For example, the UFS inode, the on-disk structure representing a file, contains an array of pointers to data blocks and, for larger files, pointers to blocks containing pointers. Data block hashes are stored alongside the pointers, within the meta-data. This solution separates the storage of hashes from their data blocks, overcoming the need to increase the storage system block size. Unfortunately, this solution has several problems of its own. First, new dependencies are introduced between file data and meta-data. Suppose the block's hash was written alongside the inode's block pointer and before the block write. If the system crashed before the modified block reached storage, the hash and block would be inconsistent after a reboot. The hash and block would also be inconsistent if the hash was to be written after the block, but the system crashed before the hash could be committed. The update of the block and hash needs to be atomic or at least subject to strict write ordering constraints. Unfortunately, while it might be possible to overcome the ordering challenges associated with this potential solution by applying sophisticated design, a more serious problem exists.

Storing the hashes with the file system pointers presents another problem that is more serious. Many pieces of meta-data are indexed directly and not referenced via pointers. These include the blocks containing inodes, cylinder groups, and superblocks. User data, if corrupted, will not crash the system because it is opaque and passed through to applications. Meta-data, on the other hand, is interpreted by the operating system and its corruption can crash the system. The protection of such meta-data is critical, because these are the blocks whose correctness is most important for system stability. If the inodes are not protected from modification, then a malicious entity could change the hash value associated with a block pointer to match that of a spoofed block. Again, a signature scheme could be used to sign meta-data. These techniques could be feasible under read-only workloads, but become problematic under workloads with frequently changing meta-data, due to the introduction of update dependencies and the need to sign meta-data.

The best features of the above-mentioned two solutions—hash protection of all blocks, both data and meta-data, and the separation of hashes from their corresponding blocks—are both achieved through a technique that we have developed called *hash logging*. Hashes, along with a unique block identifier, are written as records into a log, a separate append-only system file. The hashes of all blocks, both data and meta-data, are stored within this log and are thereby separated from the blocks they describe. Log records are keyed by monotonically increasing LSNs. When record writes reach the end of the log, they start anew at the beginning (this necessitates a log space reclamation mechanism, which will be described in section 5).

3 Related Work

Fu et al. describe the read-only secure file system (SFSRO) [5]. SFSRO is most like the second potential solution described above. File system blocks are named and requested by their hash value. The traditional disk address pointers contained in file system index nodes are replaced by hash values. This scheme depends on the collision-resistant property of cryptographic hashes. As SFSRO is designed for read-only workloads it does not handle frequent updates well. If the contents of the file system change, a database must be reconstructed on a trusted server and shipped to the untrusted server, where it cannot be modified. In contrast, PFS allows for modification directly on the untrusted storage server. PFS protects data and meta-data at the level of blocks, including all the blocks required to implement a file system on storage: data blocks, allocation bitmaps, inodes, and superblocks. SFSRO protects data blocks and inodes, but not other meta-data.

The Trusted Database (TDB) implements a database buffer cache on an untrusted store [14]. TDB shares several characteristics in common with PFS and SFSRO. All three use collision-resistant hashes to verify blocks. TDB uses a log-structured store for both block and hash writes. PFS and TDB have different philosophies, however. TDB is a monolithic database system that presents a new interface and a new on-disk storage format. In contrast, PFS is an existing file system modified internally to do hash data protection. The logic fits within the prior file system architecture — the most interesting characteristic being the unification of hash and meta-data logging within the file system journal. The popular and well-documented file system interface remains unchanged so that applications can benefit from PFS protection transparently. Also, the on-disk format of the file system remains unchanged, so that large file system partitions run under PFS immediately without copy-overs and utilities that access the raw disk interface continue to work (e.g., dump, restore, fsck).

Tripwire is a user-level system administration tool that computes hashes on a per-file basis and stores these in a protected database [12]. To verify and check for intrusion, Tripwire computes the hashes of specified files and compares these against the database, raising a flag for the system administrator if the two differ. Tripwire will

not update a file's hash value when the file is modified by a trusted party. Tripwire simply identifies changed files. Figuring out if a file was changed by a malicious user or a trusted user is a problem left to the system administrator. PFS protects the system at a lower level of abstraction; the file system block, protecting file system meta-data in addition to the file data. PFS block updates will never be committed to storage without the hash being computed and committed beforehand.

For high performance, PFS commits block hashes and meta-data update (journal) records to the same log. The idea of shared logging was investigated within the Quicksilver operating system project [9]. Quicksilver was a distributed microkernel operating system built at IBM research in the mid-1980s for System/6000 workstations. The salient feature of Quicksilver was its use of transactions for recovery, failure notification, and resource reclamation. As a microkernel, Quicksilver typically implemented system services in user-level servers. The log manager server was used primarily by the transaction manager, but was theoretically available to other subsystems. Later work explored sharing the log service across multiple resources [3]. The WAFS used by PFS is also a service for shared logging, built into a monolithic kernel architecture. Other work has explored sharing the WAFS service between the kernel and user applications [20].

4 PFS Architecture Overview

4.1 Block Modification and Writes

When buffered file system blocks are modified, either through direct user write system calls or internally generated meta-data updates, PFS flags the buffers to indicate that their hashes must be computed and logged before they may be written to the storage system. A background system thread known as *hashd* wakes periodically and cycles through the dirty buffers, identifying those buffers for which a hash must be computed. Once such a buffer is identified, hashd applies the hash function to the buffer's data block, yielding a hash value that is packed into a record along with the block identifier and written to the log. The log write returns an LSN.

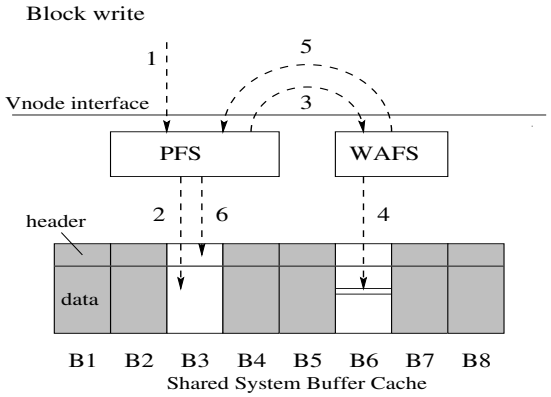

Figure 1: Block Write Example. (1) Data is written by an application. This step will not occur if the block modification is an internally generated meta-data update. (2) PFS writes data into block buffer B3 currently resident in the buffer cache. A buffer flag is set to indicate that the hash must be computed. At this point, the system is free to return. Steps 3-6 happen asynchronously within the context of the background process hashd. (3) PFS applies a hash function to the data contents of B3. This hash and B3's physical block number are encapsulated within a record and written to the WAFS via the vnode interface. (4) The WAFS takes the record and does a buffered write to the head of its log in B6. (5) It returns the LSN. (6) PFS stores the LSN in B3's buffer header.

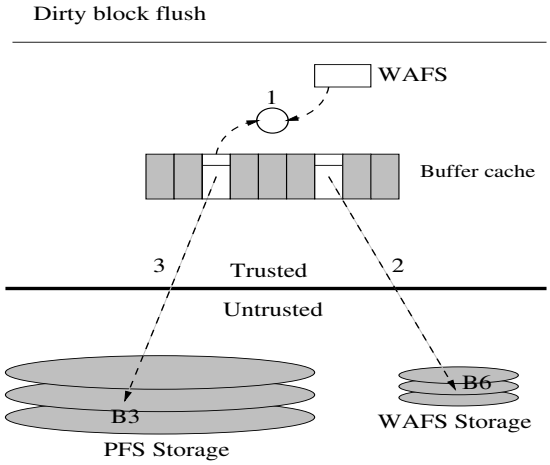

Figure 2: Dirty Block Flush Example. (1) This is the write-ahead comparison. The LSN stored in buffer B3 is compared to the WAFS highest stable LSN. If the buffer's LSN is less than or equal to the highest stable LSN, then the log record describing B3 must be stable. It is safe to write B3 without forcing the log (go to Step 3). (2) The hash record describing B3 needs to be committed. Flush the log up to the LSN stored in B3's buffer. The record happens to be stored within B6, so B6 is written. (3) Write block B3.

This LSN is stored in the block's buffer. These steps are shown in Figure 1. Steps 1 and 2 of that figure occur synchronously with the system call. Steps 3 through 6 take place in the background in the context of hashd. None of these steps require communication with the storage system.

To ensure the integrity of the data, PFS uses the well-known write-ahead logging protocol (WAL) [7]. Before PFS finally decides to write a buffer, it must ensure that the corresponding hash record has reached stable storage. If the block were written otherwise and the system crashed, a subsequent comparison of the old hash against the new data would erroneously conclude that the data was bad. Therefore, hashes must reach stable storage before their corresponding blocks. LSNs are used to enforce this dependency.

Before a block is written to disk, the LSN stored within its buffer header is compared with the stable LSN of the log. If it is greater than the stable LSN, then in order for this write to proceed, the log must first be flushed up to the buffer's LSN. This is shown in Figure 2.

Using an asynchronous thread to compute and log the hashes has several important advantages. First, it removes the hash computation from the system-call path, reducing the likelihood of needless hash recomputations. Second, it increases the likelihood that a hash will be stable by the time its corresponding dirty block reaches the device driver. If the hash were not stable by this time, two I/Os would be necessary. A synchronous write of the block's hash to the log followed by the data block write. Third, using an asynchronous thread allows hash records to be batched together. As long as WAL is followed, integrity will be protected and the hash record can be written asynchronously.

4.2 Reading a Block

The *block map* is the PFS data structure used to verify block reads. It contains the committed hash for every block, indexed by file system block number. The block map resides in kernel virtual memory and need not be entirely resident in physical memory. The portion of the map present in memory is a function of file system locality. Using paging to cache the recently accessed pages of the block map requires a trusted swap device. The block map is a volatile data structure. When the system crashes, it is reconstructed from the hash log records contained within the WAFS.

On a read, if PFS does not find a block in the buffer cache, it issues a read request to the storage system. Upon receiving the block from storage, PFS computes the hash and compares this value with the value in the block map. If the block fails this verification step, it is expelled and the associated system call fails.

4.3 Journaling and Hash Logging

File systems that do not employ techniques such as journaling or soft updates do not perform well under meta-data intensive workloads [19]. This is because they must perform synchronous writes to order meta-data updates. Journaling solves this problem by allowing the file system to buffer meta-data updates, using a log and WAL to enforce dependencies, much as hash logging does. Journaling can be thought of as protecting the operating system from itself or, at a finer level of detail, protecting the file system from the buffer cache, which is responsible for committing delayed writes, but has no knowledge of crucial file-system-specific update dependencies. Hash logging benefits from the delayed meta-data updates of journaling. If Step 2 of Figure 1 were a synchronous meta-data update, then all of the following steps within Figure 1 and the two writes of Figure 2 would have to be performed synchronously. By allowing meta-data updates to be cached delayed-write, journaling gives the system the freedom to perform hash computation in the background and to batch hash record writes.

Rather than have two logs in the system, each serving its own LSNs to enforce WAL, it makes sense to unify the two logs into one single system WAL service. PFS uses the *write-ahead file system* (WAFS) [19] for this purpose, employing it for both meta-data and hash logging. The WAFS resides on a separate partition from PFS. This allows the WAFS to be located on a different storage system from PFS, potentially removing disk head contention. On systems with NVRAM, the WAFS could be placed there.

The WAFS is not confined to trusted local storage. Therefore, it must be protected from the same classes of faults and attacks as PFS. If an adversary were able to spoof WAFS blocks, then PFS would no longer be able to protect its own blocks. During a server crash, an adversary could read the log records, which are stored in the clear on the storage, find the most recent record describing a PFS block, modify the corresponding PFS block, compute the new hash and insert it in the WAFS block in place of the old value. From this example, it is clear that the protection of WAFS blocks is paramount to the protection of PFS blocks.

WAFS does not rely on an external mechanism for its protection, instead storing an *authentication tag* in the

header of every WAFS block. The authentication tag is generated by a message authentication code (MAC). Unlike hash functions, MACs are parameterized by a secret key. Recent work has shown that MACs can be constructed from hash functions with a degree of security that is provably strong [1]. The benefit of this approach is performance, which is essentially that of the underlying hash function.

Unlike PFS, the WAFS is able to include the tag within the file system block because its units of read/write access are variable length records. Any notion of file system blocks is withheld from its clients, which read and write records, not blocks. The WAFS header also includes an LSN. This is protected by the authentication tag and used to verify that the correct block was returned. Since the server is the only system that requires verification of WAFS blocks, public key technology need not be used.

The authentication tag is computed when the WAFS block is written to storage. Since WAFS blocks are neither read nor rewritten during normal operation, the authentication tag will only need to be recomputed for partial block writes that are forced to storage.

The secret key used for HMAC computation must be secure and protected on a trusted device. It could be located on the server's trusted disk and managed as a dynamically loadable kernel module. More advanced technology such as smart cards could be used.

Figure 3 shows the journal and hash log multiplexed on the WAFS.

5 PFS Architecture Details

5.1 Asynchronous I/Os

After the server sends the write request to the storage and before it receives the I/O completion notification, the block may be either in the new or old state. If the server crashes before receiving notification, then it must have both hashes stable for comparison in the next session. If the write did not make it to the storage system, the block is still correct so the old hash must be available for comparison. The block map contains the committed hash of every block. Hashes are only entered into

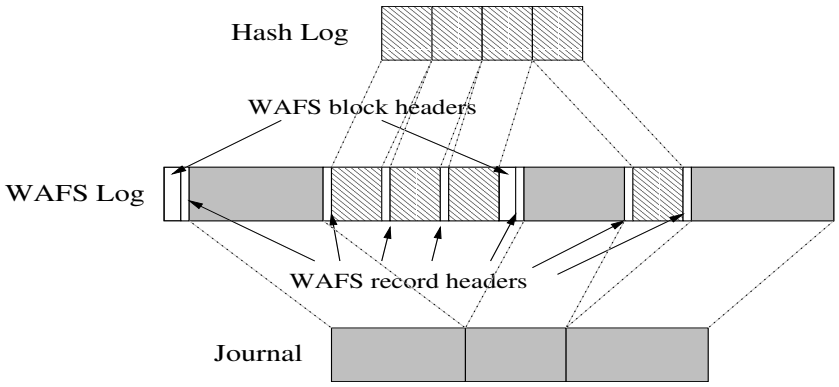

Figure 3: Multiplexing the WAFS. This figure shows two physical WAFS blocks containing four hash log records (diagonal-striped blocks) and 3 journal records (grey blocks). PFS uses the WAFS for its hash logging and journaling. The journaling and hash logging components write records during normal operation. Each component sees a distinct logical log, but the records are physically interleaved. Records are read only during recovery. WAFS does not export a block abstraction and hides the WAFS block header from its clients. The header contains an LSN needed during recovery and an authentication tag to verify WAFS block reads. The WAFS prepends a record header to every record write. This contains a client identifier used by the cursor read routines to ensure that the block map recovery code does not retrieve journal records and vice versa.

the block map once the I/O completion interrupt has been received. The *async map* is a small hash table containing all hashes describing block updates sent to storage, but for which PFS has not received I/O completion notification. Upon receiving this notification, the hash is removed from the async map and inserted into the block map, overwriting the previous value.

5.2 Checkpointing

For a 16GB partition with 8KB file system blocks, the block map will have 2M entries. If the hashes are MD-5, the block map will be 32MB. Clearly, it is undesirable to checkpoint the entire block map at once. The block I/O interrupt code accesses both the block map and the async map, so access to these data structures must be exclusively locked during the checkpoint. Locking out block interrupts in order to checkpoint tens of megabytes of data would undoubtedly result in dropped interrupts. In addition, system call latencies would be severely skewed and unpredictable. We solve this problem with partial checkpoints. The block map is broken up into distinct *chunks* approximately the size of the data portion of WAFS file system blocks. Each chunk is checkpointed independently and the chunks are selected in a round-robin fashion.

A chunk checkpoint is not complete until a small record known as the *async record* has been committed to the log following the chunk. The async record contains the contents of the async map for the file system blocks described by the chunk. These are the block I/Os in progress at the time of the checkpoint. Since there can be only one outstanding I/O on any particular block, the current state of the block is either described by the hash in the chunk or the hash in the async record. Once a chunk checkpoint completes, the log is free to reclaim earlier records for blocks described by the chunk.

For a given block, many hash records may be written after the chunk checkpoint. PFS must attempt to bound the number of potentially valid hashes, or *candidate hashes*, for a given block. Having many candidate hashes to test would slow down recovery and adversely affect security. One way to bound the number of candidate hashes per block would be to insert code in the I/O completion interrupt handler to write a special record to the hash log. This record would contain the hash value and block number like other records, but its type field would identify it as an I/O completion record. During recovery, on a read of such a record, the recovery code would deprecate all earlier hash values associated with the block. This solution would place a fairly tight bound on the number of candidate hashes per block, dependent on the flush rate of the buffer cache and the hash computation rate of hashd. The bound would be inversely proportional to the flush rate of the buffer cache and directly proportional to the buffer processing rate of hashd. Unfortunately, there is a fundamental architectural problem. Locks (e.g., the WAFS vnode lock) would have to be acquired during interrupt processing, which runs at a higher priority level than the top half of the kernel. This would introduce the potential for deadlock.

Our solution places the bounding responsibility within hashd. In order to bound the number of candidate hashes, every PFS buffer contains two LSNs. The first is the LSN immediately following the buffer's most recently written log record. The hash stored in the record preceding this LSN might not describe a committed PFS block. This LSN is known as the *buffer-end*. The second is the LSN immediately following the most recent hash describing a stable version of this buffer. This is known as the *buffer-begin.* For a given buffer, all of the hashes between buffer-begin and buffer-end have not had their corresponding data committed to disk. As hashd computes and logs hash records, buffers' buffer-end LSNs will move forward. When the PFS block is written to disk, the I/O completion interrupt will copy buffer-end into buffer-begin, moving it forward. At this point, the most recent hash in the log must be stable, due to the protection of the WAL protocol, and it must describe this particular write because the buffer is locked during physical I/O. This process is shown in Figure 4.

When hashd executes, it scans the buffer-begin values of all the PFS buffers, selecting the minimum from the buffers for which buffer-begin differs from buffer-end. This LSN is written to the log in a special record. This LSN has the property that if there are no hash records describing a particular block after this LSN, then the most recent hash record preceding the LSN describes this block. In Section 5.3 we will describe how this LSN, known as *logged*, is used in recovery.

The WAFS is a finite circular log and is responsible for managing this space. When the WAFS reaches a minimum free space threshold, it synchronously checkpoints its clients. The journal follows the methodology described in Seltzer et al. [19] and is beyond the scope of this paper. WAFS checkpoints the hash log by calling a PFS handler and providing a minumum acceptable LSN. The handler compares this against the LSN of the oldest live checkpoint chunk. If it is less, then PFS simply returns the LSN of the oldest checkpoint without any I/O. Otherwise, PFS checkpoints enough chunks for

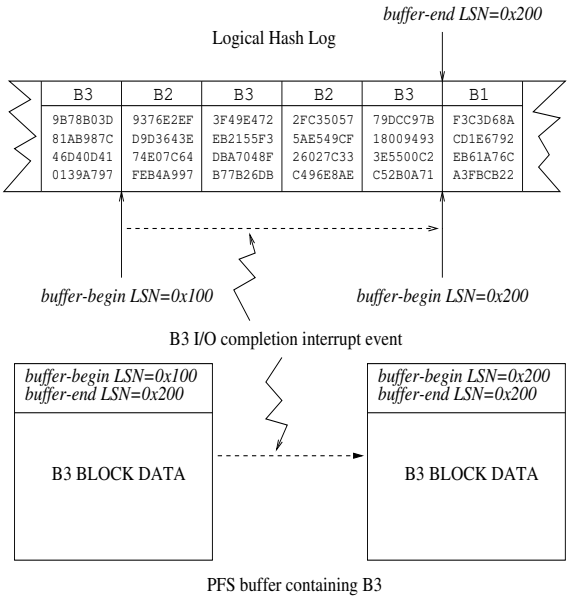

Figure 4: Updating the buffer-begin LSN. On reception of a block I/O completion interrupt, the buffer-end LSN is copied into the buffer-begin LSN. These two LSNs are used by hashd to compute the global logged LSN. If the buffer-begin and buffer-end LSNs are equal, the block contained within the buffer does not have any logged hashes that must be tested and resolved during recovery.

the oldest live LSN to move beyond the minimum acceptable LSN. This illustrates the fact that the WAFS space reclamation checkpoint must occur with enough free space to accomodate possible hash log checkpoints. Journal checkpointing will also require free space. Interestingly, the minimum free space is easier to bound for hash logging than for journaling. Heuristic bounds for journaling use the maximum number of open file descriptors and the maximum number of journal entries per system call [19]. The upper bound for hash logging is simply the size of the block map plus a small amount of space for an async record.

5.3 Recovery

The WAFS itself must be recovered before any PFS recovery can begin. The most recent checkpoint is found, and the log is then read sequentially until the end of the log is detected. The LSNs stored in the WAFS block headers are monotonically increasing, allowing the recovery code to find the most recent block. The authentication tag protects against incomplete writes.

PFS recovery has two phases: hash recovery and journal recovery. The hash recovery restores the block map to a consistent state. The journal recovery follows and restores the meta-data integrity. The journal recovery process is unmodified from the original implementation described by Seltzer et al [19].

The hash recovery code reconstructs the block map and verifies the contents of any blocks for which multiple hash values are potentially valid. The hash recovery code receives a log cursor handle from the WAFS that allows it to iterate through the log without receiving the journal records. The hash recovery code finds the oldest valid chunk checkpoint and rolls forward searching for the most recent record containing a logged LSN (see section 5.2 for a description of this special LSN). Once this is found, hash recovery returns to the start of the log for a second pass. Block map recovery requires little I/O up to the logged LSN. This is because every hash value either describes a committed block or is deprecated by a later record that does. Therefore, the hash values can be inserted directly into the block map with no I/O. Once recovery passes the logged LSN, every hash value must be checked for validity. The recovery code maintains a linked list of candidate hashes for every block. On reaching the log end, the hashes are resolved. Every block with multiple valid hashes is read in, the hash function is computed, and the resulting value is compared against all the potentially valid hashes. If it does not match any, recovery fails. Otherwise, the matching hash is inserted into the block map and all the others discarded. This illustrates the importance of bounding the number of candidate hashes per block as described in Section 5.2.

6 Performance Analysis

PFS and WAFS have been implemented in the FreeBSD-4.1 operating system. PFS is a set of changes to the Logging Fast File System (LFFS) [19], a journaling version of FFS [15]. PFS maintains the on-disk structure of FFS, and it uses the WAFS for both its journaling and hash logging as described in this paper.

For these experiments, PFS computes hashes using the MD-5 algorithm and WAFS computes its block authentication tags using the IETF RFC 2104 implementation of HMAC using MD-5 [13]. The HMAC secret key is stored within the kernel binary.

Our test system consists of a single 500Mhz Xeon Pentium III CPU with 512MB RAM and three 9GB 10,000 RPM Seagate Cheetah (ST39102LW) disks. The disks are connected to the host operating system via a single shared Adaptec AHA-2940UW Ultra SCSI card. The first disk contains the operating system and swap space, the second contains a 256MB WAFS partition, and the third contains an 8.5GB test partition.

We ran two macrobenchmarks and a microbenchmark suite. We compare PFS against LFFS. In our experiments, both PFS and LFFS do asynchronous journaling. This means that journal records are not committed to disk before the system call returns. WAL still maintains the ordering of updates so that the file system will be recoverable to a consistent state. However, it is possible that a create will return, the system will crash, and the file will not exist after recovery, violating the durability semantics that FFS has traditionally offered. We believe that these semantics grew out of convenience, rather than application demand. Originally, FFS used synchronous updates to order meta-data operations and ensure recoverability and consistency. Since the updates had to happen anyway, the semantics of durability came at no additional cost. Soft updates [8][16] and asynchronous journaling abandon these semantics. The difference between the performance of synchronous journaling and either asynchronous journaling or soft updates, whichever is better, is the cost of FFS system call durability. Seltzer et al. determined that the cost of durability far exceeds the cost of integrity [19]. The goal of our benchmarking experiments is to quantify the cost of the block-level integrity protection of PFS.

6.1 Microbenchmarks

The microbenchmarks are similar to those used in recent file system performance studies [18][19]. For file sizes ranging from 16KB to 1MB they create, write, read, then delete a directory hierarchy of files. The file system is unmounted between each operation phase, to ensure that each phase begins with a cold cache. This is useful for isolating the cost of individual operations, but is somewhat artificial. Much of the design of modern file

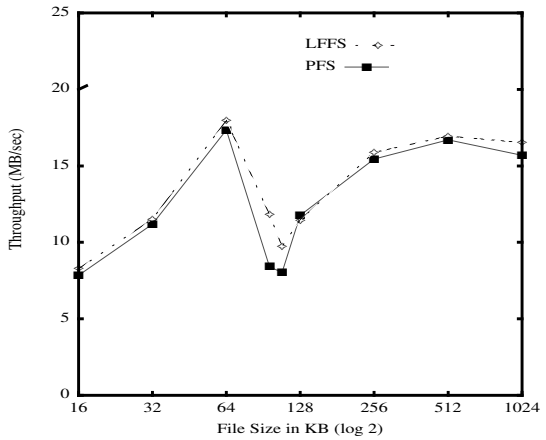

Figure 5: Throughput (MB/s) of creates.

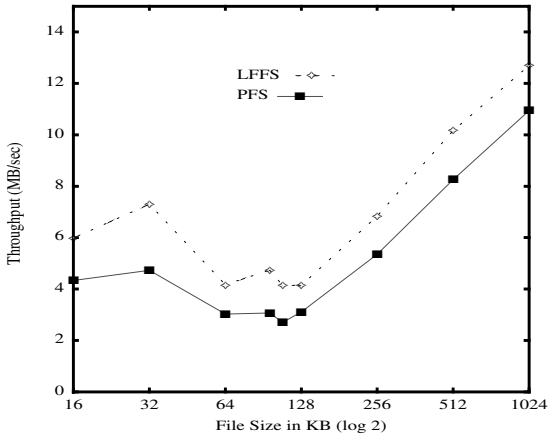

Figure 6: Throughput (MB/s) of writes.

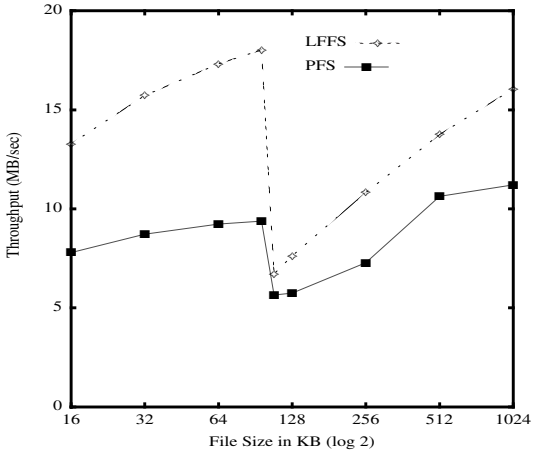

Figure 7: Throughput (MB/s) of reads.

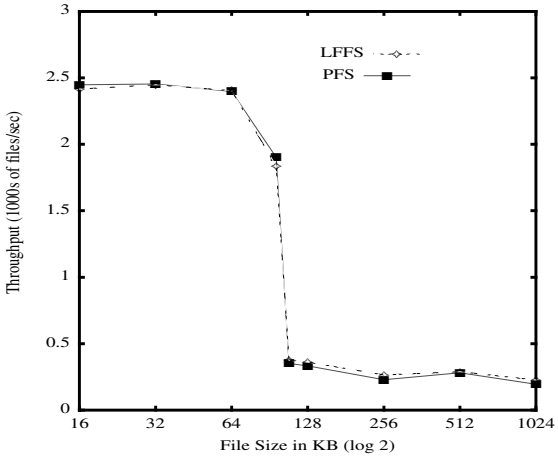

Figure 8: Throughput (thousands of files/s) of deletes.

Figures 5-8: Cold cache microbenchmarks. Figures 5 through 7 show the throughput of both PFS and LFFS expressed in MB per second. Figure 8 shows delete throughput expressed in thousands of files per second. The create and delete benchmarks are meta-data intensive. Note that the scale of the y-axis varies. Each test was run 5 times and standard deviations were small.

systems is predicated on a large cache with a high hit rate.

The hierarchy contains either 128MB of data or 512 files, whichever results in more data. No more than 50 files are allocated per directory to limit pathname lookup times.

The results of the microbenchmarks are shown in Figures 5 through 8. The performance of PFS and LFFS are similar on the create and delete tests. This shows how hash logging benefits from journaling. Journaling allows meta-data updates to be cached and updates combined. The performance of hash logging depends on the ability of the operating system to cache writes. This gives hashd time to generate hash log records and group these records together into full block writes, amortizing the cost of block writes.

On the read and write tests, LFFS outperforms PFS because PFS must compute hashes when blocks are written to and read from disk.

The largest disparity is on the read benchmark. This test has no data locality, but some meta-data locality due to the repeated lookup of directory components. For medium-sized files between 32KB and 96KB LFFS throughput is nearly double that of PFS. At 96KB, the read throughput collapses as the two systems must read the indirect block. For larger files these layout issues, which are independent of the hashing mechanism, play a more significant role and PFS performance improves relative to LFFS. The performance disparity is narrower on the write test. Here, the buffer cache is able to absorb many of the writes.

6.2 Macrobenchmarks

The microbenchmarks are useful for isolating the operations for which PFS and LFFS performance differs. However, it is difficult to use their results to predict application performance. The goal of our macrobenchmark experiments is to quantify the cost of block-level protection for realistic workloads.

FFS mounted asynchronously (FFS-async) does not make any attempt to ensure file system recoverability or protection. It is useful as an upper bound on performance because it shares layout format and algorithms with PFS and LFFS.

6.2.1 POSTMARK Benchmark

The PostMark benchmark was designed to model a combination of electronic mail, netnews, and e-commerce transactions, the type of load typically seen by Internet Service Providers (ISP) [11]. PostMark is meta-data intensive with many small files and high memory pressure. File sizes vary uniformly from 512 bytes to 16KB. Figure 9 shows the performance of the three systems. The throughput of PFS is 7.9% lower than LFFS and 9.1% slower than FFS-async. Although PFS performs well on meta-data operations, due to the fact that it is journaled and meta-data updates are cached, it does not perform comparatively well on small reads and writes.

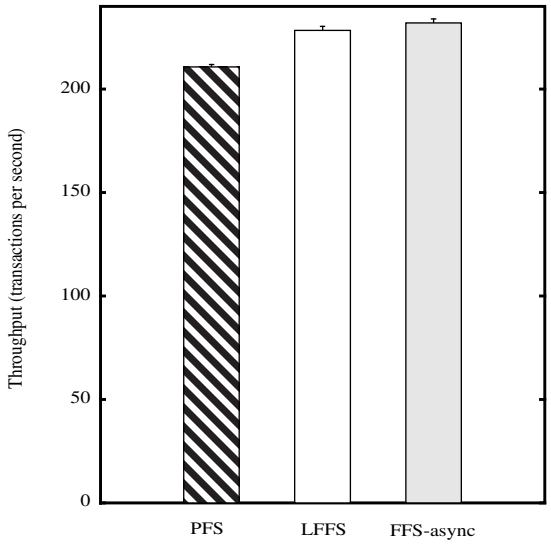

Figure 9: Throughput of PostMark macrobenchmark. Expressed in transactions per second.

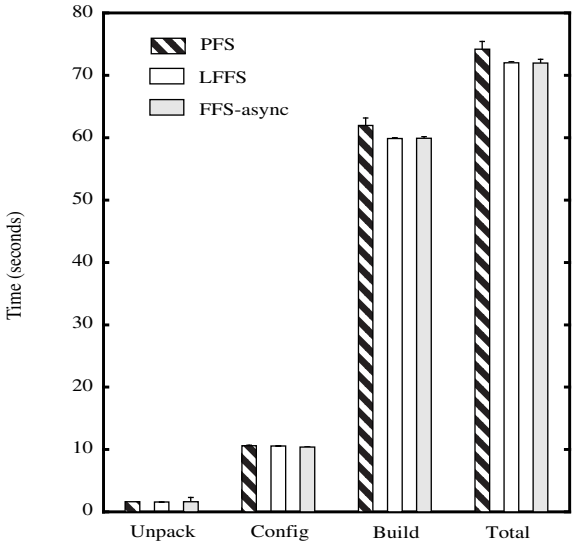

Figure 10: Time (s) taken for SSH-Build to complete. Unlike figures 5 to 9, here lower values indicate higher performance.

6.2.2 SSH-BUILD Benchmark

The SSH-Build benchmark unpacks, configures, and builds the secure shell (SSH) program, a medium sized software package [21]. It consists of three phases. The first, *unpack*, decompresses a tar archive and pulls the source files out of the archive. This phase is relatively short and characterized by meta-data operations. The second, *config*, runs small scripts and programs and generates small files. The third, *build*, compiles the source tree, reading and parsing the source files and generating object files, that are subsequently linked into the executable. Figure 10 shows the performance of the three systems across the three stages. The performance of the three systems on the first two phases is not statistically distinguishable. PFS takes 3.5% longer to complete the build phase. Being less meta-data intensive with larger files than the two earlier phases, this reconciles with our observations from the microbenchmarks.

7 Conclusions

We have presented an evolutionary file system architecture for strong block-level integrity. Our architecture changes neither the file system interface nor its on-disk layout and requires no special support from storage systems. This has been accomplished by modifying the file system to log collision-resistant hashes to a general operating system logging service. This service is shared with the file system journal. For efficiency, a background thread computes and logs hashes, removing the hash computation from the system call path and the log commit from the data I/O path.

We measured the performance of PFS using micro and macrobenchmarks. PFS performance is similar to that of the LFFS journaling file system under the meta-data intensive create and delete tests. Under the read and write tests, PFS incurs hashing costs as data is transmitted to and from storage. On the macrobenchmarks, the overhead of block-level integrity is only 3.5% on SSH-Build and 7.9% on PostMark. We believe this is a small price to pay for strong protection.

Acknowledgements

Thanks to David Molnar for bringing HMAC to our attention. Thanks to David Robinson for comments on an early version of this architecture.

References

[1] Bellare, M., Canetti, R., Krawczyk, H., "Message Authentication using Hash Functions - The HMAC Construction," *RSA Laboratories CryptoBytes*, Vol. 2, No. 1, Spring 1996.

[2] Clark, T., "Designing Storage Area Networks," Addison-Wesley, Reading, MA, 1999.

[3] Daniels, D., Haskin, R., Reinke, J., Sawdon, W. "Shared Logging Services for Fault-Tolerant Distributed Computing," Position Paper for Fourth ACM SIGOPS European Workshop, Bologna, Italy, Sept. 1990.

[4] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[5] Fu, K., Kaashoek, F., Mazieres, D., "Fast and secure distributed read-only file system," *Proceedings of OSDI 2000*, San Diego, CA, October 2000.

[6] Gibson, G. A., Van Meter, R., "Network Attached Storage Architecture", *Communications of the ACM*, 43(11), November 2000.

[7] Gray, J., Reuter, A., "Transaction Processing: Concepts and Techniques," Morgan-Kaufmann, 1993.

[8] Ganger, G. R., Patt, Y. N., "Metadata Update Performance in File Systems," *Proceedings of the First OSDI*, Monterey, CA, Nov. 1994.

[9] Haskin, R., Malachi, Y., Sawdon, W., Chan, G., "Recovery Management in QuickSilver," *ACM Transactions on Computer Systems*, 6(1), pp. 82-108, Feb. 1988.

[10] Internet Engineering Task Force, IP Storage (IPS) Working Group Charter, http://www.ietf.org/html.charters/ips-charter.html as of March 28, 2001.

[11] Katcher, J., "PostMark: A New File System Benchmark," Technical Report TR3022. Network Appliance Inc., Oct 1997.

[12] Kim, G. H., Spafford, E. H., "The design and implementation of Tripwire: A filesystem integrity checker". *In Proceedings of the 2cnd ACM Conference on Computer and Communications Security*, 1994.

[13] Krawczyk, H., Bellare, M., Canetti, R., "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Internet Engineering Task Force, Network Research Group, February 1997.

[14] Maheshwari, U., Vingralek, R., Shapiro, W., "How to Build a Trusted Database System on Untrusted Storage," *Proceedings of OSDI 2000*, San Diego, CA, October 2000.

[15] McKusick, M.K., Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX," *ACM Transactions on Computer Systems* 2(3), pp 181-197. Aug. 1984.

[16] McKusick, M.K., Ganger, G., "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proceedings of*

the 1999 Freenix track of the USENIX Technical Conference, pp. 1-17. Jun. 1999.

[17] Rivest, R., "The MD5 Message-Digest Algorithm," RFC 1321, Internet Engineering Task Force, Network Working Group, April 1992.

[18] Seltzer, M. I., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V. "File System Logging versus Clustering: A Performance Comparison," *Proceedings of the 1995 USENIX Technical Conference*, pp. 249-264. New Orleans, LA, Jan. 1995.

[19] Seltzer, M. I., Ganger, G. R., McKusick, M. K., Smith, K. A., Soules, C. A. N., Stein, C. A., "Journaling vs. Soft Updates: Asynchronous Meta-data Protection in File Systems," *USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[20] Stein, C. A., "The Write-Ahead File System: Integrating Kernel and Application Logging," Harvard CS Technical Report TR-02-2000, Cambridge, MA, Apr. 2000.

[21] Ylonen, T. "SSH—Secure Login Connections Over the Internet," *6th USENIX Security Symposium*, pp. 37-42. San Jose, CA, Jul. 1996.