Selected Project Reports, Spring 2005
Advanced OS & Distributed Systems (15-712)

edited by Garth A. Gibson and Hyang-Ah Kim

Jangwoo Kim[††], **Eriko Nurvitadhi**[††], **Eric Chung**[††]; **Alex Nizhner**[†], **Andrew Biggadike**[†], **Jad Chamcham**[†]; **Srinath Sridhar**[*], **Jeffrey Stylos**[*], **Noam Zeilberger**[*]; **Gregg Economou**[*], **Raja R. Sambasivan**[*], **Terrence Wong**[*]; **Elaine Shi**[*], **Yong Lu**[*], **Matt Reid**[††]; **Amber Palekar**[†], **Rahul Iyer**[†]

May 2005

CMU-CS-05-138

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

†Information Networking Institute
∗Department of Computer Science
††Department of Electrical and Computer Engineering

Abstract

This technical report contains six final project reports contributed by participants in CMU's Spring 2005 Advanced Operating Systems and Distributed Systems course (15-712) offered by professor Garth Gibson. This course examines the design and analysis of various aspects of operating systems and distributed systems through a series of background lectures, paper readings, and group projects. Projects were done in groups of two or three, required some kind of implementation and evalution pertaining to the classrom material, but with the topic of these projects left up to each group. Final reports were held to the standard of a systems conference paper submission; a standard well met by the majority of completed projects. Some of the projects will be extended for future submissions to major system conferences.

The reports that follow cover a broad range of topics. These reports present a characterization of synchronization behavior and overhead in commercial databases, and a hardware-based lock predictor based on the characterization; design and implementation of a partitioned protocol offload architecture that provides Direct Data Placement (DDP) functionality and better utilizes both the network interface and the host CPU; design and implementation of file indexing inside file systems for fast content searching support; comparison-based server verification techniques for stateful and semi-deterministic protocols such as NFSv4; data-plane protection techniques for link-state routing protocols such as OSPF, which is resilient to the existence of compromised routers; and performance comparison of in-band and out-of-band data access strategies in file systems.

While not all of these reports report definitely and positively, all are worth reading because they involve novelty in the systems explored and bring forth interesting research questions.

Contents

Optimistic Lock Speculation

Jangwoo Kim
jangwook@ece.cmu.edu

Eriko Nurvitadhi
enurvita@ece.cmu.edu

Eric Chung
echung@ece.cmu.edu

15-712 Professor Garth Gibson
School of Computer Science
Carnegie Mellon University

Abstract

Conservative locking of shared data or code regions can incur substantial overhead in shared-memory parallel programs. Optimistic concurrency techniques have been proposed to allow simultaneous access to shared regions while maintaining program correctness. Recently proposed hardware-based techniques are promising since they require no changes to an instruction set architecture for identification of critical regions. However, these techniques have not been evaluated with commercial workloads and assume easily identifiable critical section boundaries. Commercial applications such as online transaction processing are important due to their market dominance and may benefit through optimistic concurrency since they are multithreaded. Protection of critical sections in such applications may employ complex locking that could necessitate more sophisticated detection of critical regions for allowing hardware-based optimistic concurrency.

This paper addresses these unknowns by characterizing locking behavior in commercial databases (OLTP on DB2 and Oracle). We also quantify opportunity for optimistic concurrency and propose lock prediction hardware for enabling hardware-based optimistic concurrency. In our characterization, we found that (1) most locks are test&set variants; (2) synchronization overhead is a significant fraction of execution time (30%); (3) optimistic speculation using simple lock prediction can improve performance by 14% simply by removing the overhead of executing atomic instructions.

Keywords: Lock Characterization, Synchronization, Lock Prediction, Optimistic Concurrency

1. INTRODUCTION

Conservative locking of shared data or code regions in shared-memory parallel applications can incur a substantial overhead in execution time. To address this issue, researchers have revisited the idea of optimistic concurrency (in database transactions) [14] and allow simultaneous access to critical sections through speculation techniques in hardware.

Recently proposed techniques [2][3][4] have focused on providing hardware support for concurrent execution without changes to an instruction set architecture. This is accomplished by dynamically identifying lock boundaries and executing speculatively in a concurrent fashion when entering a shared region. The execution eventually commits if correctness is not compromised (e.g. no races detected). Otherwise, rollback is initiated.

These techniques are promising since they require no recompilation of existing binaries. However, many of these techniques have only been evaluated using open-sourced benchmarks and assume that shared region boundaries are protected by conservative locks with simple constructs that are easily identifiable dynamically (e.g. by detecting atomic load-store pairs for lock acquire and release).

Due to the aforementioned reasons, the applicability of these techniques for commercial applications (e.g. databases) would rely on the assumption that shared regions are protected by dynamically identifiable locks. Commercial applications are an area of importance due to their market dominance and potentially could benefit the most through concurrent execution due to their multithreaded characteristics.

Furthermore, these commercial applications do not use open-sourced lock libraries such as the simple test&set synchronization primitives assumed in lock speculation hardware proposed by Rajwar et al [2][3].

In this paper, we contribute a characterization of synchronization behavior and overhead in OLTP on DB2 and Oracle. We also propose a hardware-based lock predictor based on the insights gained from the characterization study. This lock predictor could potentially benefit hardware-based optimistic concurrency techniques that require annotated critical section boundaries [5][6] by avoiding the need for binary rewriting (annotation of critical region boundaries) or modification of an instruction set architecture.

More specifically, this paper seeks to: (1) statically characterize synchronization techniques used in operating systems and databases; (2) dynamically characterize locking behavior in commercial databases via full-system simulation; (3) characterize opportunity for optimistic concurrency by speculating on locks; (4) develop a hardware lock predictor for supporting optimistic lock concurrency.

The rest of the paper is organized as follows. Section 2 provides background on optimistic concurrency. Section 3 discusses various lock constructs that have been proposed in literature. Section 4 and 5 present our static and dynamic characterization of locks. Section 6 elaborates on the proposed

lock predictor hardware. Section 7 and 8 provides implications for future work and concluding remarks.

2. BACKGROUND

2.1. Optimistic Lock Speculation

We refer to optimistic lock speculation as a hardware technique for dynamically identifying a critical section and executing within it without acquiring the locks needed. This dynamic execution can be thought of as a database transaction that commits only if atomicity and isolation properties can be guaranteed. Specifically, no two processors executing in a critical section should have their accesses appear non-atomic to one another. Both should have executed in the appearance of some serial order. By allowing such speculation, overall concurrency can increase when multiple threads execute the critical section simultaneously but operate on different sets of data. Bypassing of lock acquires and releases can also reduce execution time overhead, which can be costly in a distributed shared memory multiprocessor.

Figure 1 illustrates the process of optimistic lock speculation. Part a shows the normal lock execution without any speculation, where the lock acquire and release overhead is incurred. Part b shows the lock speculation when a race does not happen. In this case, the acquire and release are bypassed and their overheads are eliminated. In part c, the case where a data race occurs is shown. To maintain correctness when a data race is detected violating processors must rollback and restart. Note that if there is no data race (e.g. the store by CPU2 was to Y instead of X), simultaneous access to the critical regions as depicted in part c will not result in rollback and therefore, overall concurrency improves.

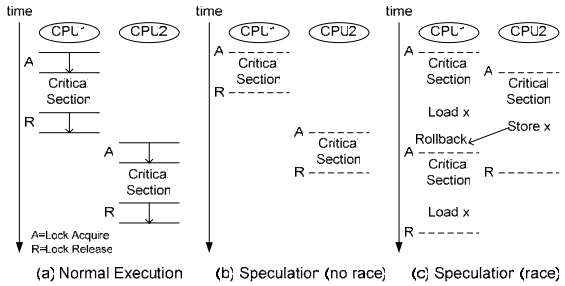


Figure 1 – Optimistic Lock Speculation

2.2. Techniques for Optimistic Concurrency

One of the earliest forms of eliminating lock maintenance appeared in Optimistic Methods for Concurrency Control [14] which proposed to maximize the throughput of database transactions by allowing concurrent accesses to shared data between processors and providing a validation phase in order to ensure correctness. Although proposed in the context of transactions for databases, the key idea borrowed from this work is to optimistically enable a processor to buffer speculative data and release it (i.e. becoming visible to other processors) into the memory hierarchy if a transaction commits.

The first instance of this idea applied to hardware appeared when Herlihy and Moss [1] introduced the Transactional Memory (TM) architecture that allowed custom read-modify-write operations to be defined by programmers to achieve a lock-free environment. TM was implemented as an extension to the cache-coherence protocols in multiprocessor systems and by providing a small, fully-associative transactional cache to buffer speculative data in critical sections.

More recently, this idea was revived when Rajwar and Goodman [2] observed that most of the time concurrent execution in lock-protected regions do not actually result in conflicts. They proposed Speculative Lock Elision (SLE) which predicts unnecessary synchronization and speculatively permits concurrent execution in critical regions. Misspeculation is detected using existing cache coherency mechanisms and recovery is accomplished by rolling back to the buffered register and memory state. In this case, the depth of SLE speculation is limited by the size of the store buffer, since speculative writes in a critical section are unreleased to the memory system until commit.

As an extension to this work, Rajwar and Goodman [3] proposed Transactional Lock Removal (TLR), which focuses on resolving synchronization conflicts. While original SLE would rollback in the presence of atomicity violations (e.g. without guaranteeing forward progress on lock transactions), SLE augmented with TLR would resolve the conflict by using timestamps to decide the conflict winner and defer response to synchronization messages such that delayed accesses to the shared data is achieved and rollback is avoided. However, since TLR assumes SLE as the underlying infrastructure, the SLE limitation for the depth of speculation exists.

Martinez and Torrellas [4] proposed Speculative Synchronization (SS), which utilizes Thread-Level Speculation (TLS) techniques to conduct speculations with respect to synchronization. With SS, a speculative thread is spawned for each speculation on a synchronization point (e.g. locks, barriers flags). One safe thread is kept at all times to ensure forward progress. SS utilizes the local cache hierarchy to buffer speculative data. Thus, the depth of speculation in SS is affected by the number of speculative threads that can be spawned and the size of the local cache(s).

Hammond et al. [5] proposed a new shared memory model, Transactional Memory Coherence and Consistency (TCC), at which the basic unit of parallel work is a transaction (an atomic set of instructions). This new model trades off inter-processor bandwidth with more tolerance to latency and simplicity. The supporting hardware relies on rollback and recovery mechanisms to deal with correctness violations. The depth of speculation in this scheme is dictated by the rollback capability of the supporting hardware.

Ananian et al. [6] similarly proposed Unbounded Transactional Memory (UTM), which extends the TM approach to allow for a large speculative window. This is accomplished by allowing speculative data to spill into the L1 cache and a special region in main memory, similar to the technique proposed by Gniady and Falsafi to enlarge the

speculative history buffer space in L2 [7]. While UTM provides a deeper speculation window in comparison to previous proposals, the way the speculative data is spilled into main memory induces a substantial overhead, at which it reduces performance up to a factor of 14 for one of the benchmarks used in the UTM evaluation. To deal with storing speculative data in the cache, an extra speculative bit is added for each cache line.

2.3. Characterizing Optimistic Concurrency

Despite the volume of research on optimistic concurrency for transactional execution of critical sections, no work has characterized how these concurrency schemes would perform on existing commercial workloads such as On-Line Transaction Processing (OLTP) or Decision Support Systems (DSS). Previous research in lock speculation has mostly been evaluated in the context of scientific workloads such as the SPLASH and SPLASH-2 benchmark suites [11], which are not representative of commercial workload behaviors [13].

A recent characterization of the effects on memory-level parallelism provided by Chou et al. [15] showed that for three transaction-based workloads (database, SPECjbb2000, SPECweb99), a primary limiter in memory-level parallelism was generated by the presence of serializing or atomic instructions (e.g. Load-store, Compare-and-swap, Membar), which usually form the anatomies of locks, barriers, condition variables, etc. When a serializing instruction appears in a processor's pipeline, all instructions are forced to drain before the serializing instruction is permitted to issue. For SPECjbb2000, serializing instructions on average appear in 0.6% of all instructions (1 serializing instruction per 166 instructions) which is a severe limiter in memory-level parallelism and therefore performance.

Singhal et al. [18] also showed that in an analysis of locking behavior in IBM's DB2 relational database system that locking contention is fairly infrequent, which suggests that there is opportunity for lock speculation since lock conflicts are rare (and therefore data conflicts are infrequent).

3. LOCK CONSTRUCTS

This section describes several lock constructs that utilize a single lock variable. First, a simple test&set construct and its variants are discussed. Then, more sophisticated queuing locks are presented based on survey work by Michael L. Scott who originally proposed queue-based approaches to synchronization [19] [20]. These queuing locks are beneficial in reducing lock overhead during contention and providing fairness.

3.1. Test&Set Locking

In a test&set lock, a lock is represented by a shared Boolean variable indicating whether the lock is held. A processor attempting to acquire the lock performs a 'test' by reading the lock variable. If the variable returns false, it means that the lock is currently free. The processor then proceeds to 'Set' the lock by writing true to the variable. If another processor later tries to acquire the lock, its 'test' operation will return true, indicating that lock is already held. In this case, the processor can wait by looping and testing the lock variable (i.e. spinning) until it reads the value of false indicating that the lock has been freed and that it can proceed with the 'Set' operation. Releasing the lock is done simply by writing the value of false to the lock variable. The test&set operations have to be done atomically to ensure that the tested value has not been updated by another processor when the set is performed. Thus, the test&set is typically combined as a single atomic instruction (e.g. LDSTUB in SPARC).

The disadvantages of a simple test&set lock are that it does not provide fairness and there is a substantial overhead due to contention during spin. Fairness is not enforced when there are multiple processors contending for the lock; the one that performs the set the earliest after the lock is released will win the lock. Thus, a processor may be too late each time in getting the lock after the previous lock-owner releases. To provide fairness, the lock should be given to the next requestor based on the time each of the lock contenders perform the lock request. Queue-based locks that address this issue are described in the next subsections.

Spinning on a shared variable incurs substantial overhead during lock contention due to the atomicity of test&set instructions. Since this instruction is atomic, the test has to be followed by a set each time, even if the test turns out to be false. This incurs overhead for each spin event. In cache coherent multiprocessor systems, this set operation can continuously bounce a lock variable from cache to cache in the presence of contention. The test&test&set construct avoids this issue by employing an additional test independent (i.e. with a conventional load instruction) of the atomic test&set instruction. The test&set is executed only if the first test is true.

Another cause of overhead during contention is the burst of succeeding lock acquires after a lock release in the presence of multiple contenders. This is because each contender attempts to obtain the lock right after it is released. A backoff technique can be used to reduce such burstiness. The idea is to have a contender wait (backoff) before attempting to obtain the lock. Each contender backs off with a different time, so lock acquiring is more distributed.

3.2. Ticket-based Locking

A ticket-based lock utilizes two unique addresses that processors modify and observe. The first address location, which we call the ticket counter, is used by processors trying to acquire a lock. When a processor requests a lock, it performs a fetch-and-increment on the ticket counter, which gives the processor a ticket number and increments the ticket counter atomically. The processor then spins on a second memory address, which we call the ticket release, and compares its own ticket value with the ticket release.

As can be seen, each processor acquires a unique ticket number (provided that there is no wrap-around). The processor that has a matching ticket number with the ticket release is permitted to enter the critical section. When the processor leaves the critical section, it commits an ordinary store to the ticket release that is the value of its own ticket plus one. This enables the subsequent ticket owner to acquire the lock.

This mechanism relieves pressure on the lock variable and can reduce hot spots in the interconnection network. However, all of the processor still spin on a single local address.

3.3. Array-based Locking

An improvement to the previous technique is to keep the ticket numbers but use the ticket number to index into an array of boolean variables. All elements of the array are initialized to false save the first entry. When a processor acquires the ticket number, it will check against the index into the array using the ticket number (just like in the previous with a ticket release). When a processor releases a lock, it is responsible for freeing the subsequent ticket owner who is spinning on a separate boolean address. This technique is advantageous over the first since it distributes contention for the lock over an array of lock variables, which could reduce hot spots in an interconnection network. However, the only disadvantage is that the array has to be defined statically according to the number of possible locks initialized in the system. Both the ticket-based and array-based locks also ensure fairness in the system; it is guaranteed that no processor will suffer from livelock or starvation. The only disadvantage is the overhead of issuing an expensive fetch-and-op atomic instruction.

3.4. List-based Locking

List-based locking is designed to have the same properties as array-based synchronization methods but enable dynamic allocation of local variables a processor can spin on demand. By default, a global lock structure has its pointer value initialized to a null value. When a processor tries to acquire the lock, it will first dynamically allocate a special data structure called the q-node ("Queue Node"), which contains a local boolean variable and a local pointer. First, the processor will initialize its local pointer to the global lock variable and then perform a fetch-and-store on the lock variable, replacing it with a pointer to its own dynamic heap structure. As a consequence, any subsequent processors wanting to acquire the lock will repeat the aforementioned procedure on the global lock variable, chaining up in a linked list. The rules for acquiring a lock are as follows. When a processor detects that it has a null pointer (for example, it was the first to observe the global pointer after it was initialized to null), it has access to the critical section. If the lock pointer is not null, it will initialize its local boolean variable to false and spin on it. Embedded within the global lock is also a "next" pointer. Each processor that fails to acquire the lock will set this "next" pointer to itself. As a consequence, the lock owner who finishes with the critical section is responsible for freeing up the next processor on the linked list.

The common theme to all of these queuing lock techniques is to enable contending processors to spin on unique memory addresses.

4. STATIC ANALYSIS OF LOCKS

4.1. Methodology

The static analysis of locks was accomplished by looking at source codes or snippets of code presented in literature for database software and operating systems. Since commercial databases are closed-source, we chose to investigate an open source experimental database library called SHORE [21]. For an operating system, we chose to look at snippets of synchronization code for Linux as provided in the source code and descriptions from kernel development books [16][26]. To provide further insights, a qualitative survey of synchronization constructs in Solaris was also performed. The following subsections present the results of the analysis.

4.2. SHORE Storage Manager

The SHORE Storage Manager [21] is a set of libraries that can be used for building database software. SHORE has been widely used in research community for building experimental prototype software, such as in project Paradise [23], Predator [24], and Dimsum [25]. The wide use of SHORE makes it a good candidate for investigation for further understanding of how synchronization in typical experimental database prototypes is done.

```
/*  Acquire the mutex. Block and wait for up to timeout milliseconds
 *  if some other thread is holding the mutex. */
w_rc_t smutex_t::acquire(int4_t timeout) {
    sthread_t* self = sthread_t::me();
    w_rc_t ret;
    if (! holder) {
        /* No holder. Grab it. */
        holder = self;
    } else {
            ....
        /*  Some thread holding this mutex. Block and wait. */
        ret = sthread_t::block(timeout, &waiters, name(), this);
            ....
    }
    return ret;
}
```

```
/*  Release the mutex. If there are waiters, then wake up one. */
smutex_t::release()
{
    ....
    holder = waiters.pop();
    if (holder) {
        W_COERCE( holder->unblock() );
    }
}
```

Note: The block() method will block the current thread and puts it on 'waiters' list. The acquire() method has a variant that accepts no timeout value and blocks forever until awakens.

Figure 2 – Handling Mutual Exclusion Variables in SHORE's Thread Synchronization

SHORE-based software runs as a collection of SHORE threads. The fundamental synchronization primitives are implemented via synchronization features of these SHORE threads. A SHORE thread provides two techniques for synchronization: mutual exclusion (*mutex*) and condition variable (*cond*). The mutex synchronization uses two methods, acquire() and release(). The snippets of code relevant for these methods are shown in Figure 2. Note that SHORE assumes a uniprocessor environment, thus spinning is not used when waiting for synchronization variable since the waiter will always be put on a list. This also seems to be the reason that the 'holder' variable in Figure 4 is not protected since there

can only be one reader/writer to it due to the uniprocessor assumption.

Figure 3 shows snippets of methods used in dealing with condition variables. Mutex primitives in figure 4 is used in the implementation of condition variables. The wait() method is used when a thread is waiting for a particular condition variable. This method releases the mutex it holds and blocks waiting for the condition variable to trigger. When it does, the thread unblocks and acquires the mutex again, and wait()completes. The signal() method is used to wake up a waiter that is on top of the list when the waited condition variable becomes true. The broadcast() method has similar function, except that it wakes up all waiters instead of just one. Note that SHORE also provides a database-lock-like locking mechanism called a latch, which allows more than one holder. The implementation of a latch also uses mutex primitives (i.e. acquire(), release()) and will not be described here for brevity.

```
/*  Wait for a condition. Current thread release mutex and
 * wait up to timeout milliseconds for the condition to fire.
 * When the thread wakes up, it re-acquires the mutex
 * before returning. */
w_rc_t scond_t::wait(smutex_t& m, int4_t timeout)
{
    w_rc_t rc;
    …
    m.release();
    rc = sthread_t::block(timeout, &_waiters, name(), this);
    W_COERCE(m.acquire());
    …
    return rc;
}
```

```
/*  Wake up one waiter of the condition variable. */
void scond_t::signal()
{
    sthread_t* t;
    t = _waiters.pop();
    if (t)  {
       W_COERCE( t->unblock() );
    }
}
```

```
/*  Wake up all waiters of the condition variable. */
void scond_t::broadcast()
{
    sthread_t* t;
    while ((t = _waiters.pop()) != 0)  {
       W_COERCE( t->unblock() );
    }
}
```

Note: refer to Figure 1 for description on block(), release(), and acquire() methods

Figure 3 – Handling Condition Variables in SHORE's Thread Synchronization

While SHORE thread synchronization does not use spin locks, SHORE does have a spin lock capability for providing non-blocking I/O. Disk accesses are handled by a "diskrw" process, which deals with reading and writing to a disk. Such process will be forked for each of the disk used. The communication between the "diskrw" process and the server (software that uses the SHORE library) in UNIX is done by means of a shared-memory queue and a pipe. The spin lock capability is needed for synchronization on the shared-memory queue. An implementation of the spin lock for Solaris2 is shown in Figure 4. The acquire() is a test&set with 'ldstub' instruction, which fetches the current content of the lock and replaces the contents with 'lock held', or xFF. The release() is a store of 'lock not held' (x00) to the lock variable.

The bottom line here is that the synchronization primitives boil down to conventional test-and-set mechanisms. This is an interesting observation since we expected that in contemporary systems, there would be larger use of more sophisticated locking mechanisms. Investigating SHORE tells us that the potential of simple test-and-set lock prediction is high.

```
class spinlock_t {
   ….
   void acquire()     { while (tsl(&lock, 1)); }
   ….
   void release()     { tsl_release(&lock); }
   ….
};
```

```
// Note: tsl implementation for Solaris2
PROC(tsl)
      retl
      ldstub     [%o0],%o0     /* in delay slot */

PROC(tsl_release)
      retl
      stb   %g0,[%o0]
```

Figure 4 – Spin Lock for share-memory queue synchronization in SHORE

```
Write lock:
       // set highest-order bit of rwlp->lock
       // (make it negative)
       1: lock; btsl $31, rwlp
            // if old value was 1, lock was busy, go to 2 to spin
          jc 2
            // no write lock, check for readers (if rwlp > 0)
          testl $0x7fffffff, rwlp
            // no readers, go ahead w/ read
          je 3
            // readers exist, release write lock and spin
          lock; btrl $31, rwlp
       // spin until lock becomes 0
       2: cmp $0, rwlp
          jne  2
          jmp 1
       3:
Write unlock:
      // clear bit 31 of rwlp (make it positive)
       lock; btrl $31, rwlp

Note: negative (i.e. sign bit) rwlp indicates write lock is held, other
bits of rwlp indicate that reader(s) exists.
```

Figure 5 – An Implementation of a Write Lock in Linux.

3.3. Linux and Solaris

Linux kernel synchronization supports kernel semaphore, spin (e.g. regular) locks, and read-write locks [16][22]. The semaphore uses a typical wait and signal model and is implemented using atomic decrement and increment. The regular and read locks are implemented using test&test&set

construct, and write locks are implemented using double test&test&set to check for both readers and writers. An example of a write lock implementation is shown in figure 5.

Solaris 2 uses condition variables, read-write locks, and adaptive mutexes [17][26]. Adaptive mutexes are configurable locks that are used for critical data item in Solaris 2, which are typically less than hundreds of instructions. They spin when waiting on locks held by a running thread and block while waiting on non-running thread.

5. DYNAMIC ANALYSIS OF LOCKS

5.1. Methodology

Our dynamic analysis was accomplished by observing the run-time behavior of locks in pthreads microbenchmarks and commercial workloads. The SimFlex timing model, which is built on top of the Virtutech Simics full-system simulator [27][28], is used to simulate our Pthreads microbenchmark and an OLTP workload (i.e. TPC-C) running on IBM's DB2 and Oracle. In investigating lock constructs used by the workloads under study, run-time traces were collected and analyzed. Specifically, we searched for instruction sequence patterns that resembled lock constructs presented in section 3. Characterization of locks was accomplished by adding instrumentation code to the simulator to collect statistics on lock events of interest (e.g. lock acquire, release, number of addresses touched in critical region). Table 1 shows our simulator configuration. Table 2 shows the configuration of the OLTP workloads used in this study.

System	16 way DSM
CPU	4-Ghz uSparcIII ISA 8way-OOO-CPUs
Core	256 ROB, LSQ, STB
L1	L1 cache: 2-way 64KB
L2	L2 cache: 8-way 8MB
Main memory	Memory: 3GB, 16bank for each module, 60 ns access per bank

Table 1 – System Configuration in SimFlex.

IBM DB2	100 Warehouses (10GB) striped over 32 disk partitions
	64 clients, 450MB buffer pool space
Oracle	100 Warehouses (10GB), 16 clients, 1.4GB SGA

Table 2 – Commercial workload parameters

5.2. Pthread Microbenchmark

As a first step towards understanding how locks behave at the assembly level, we wrote a series of microbenchmarks that exercised conventional locking with varying degrees of contention and number of threads. The microbenchmark source code is annotated to have the simulator insert breakpoints during trace generation that indicate the boundaries of lock acquires and releases. By analyzing the traces, we observed how locks behave in Pthread applications. Figure 6 illustrates the assembly.

```
pthread_mutex_lock:
      // Register %o1 = 0xff (setting the lock to held)
      ldstub [lock address], %o1
      // %o1 is now the old lock value, 0x00 if free
      // orcc sets a condition variable indicating
      // if lock was held
     orcc %g0, %o1, %g0
      // The branch instruction tests the condition code
      be, address

pthread_mutex_unlock:
      // Register %o5 = 0x0 before, %o5 = 0xff after
      swap [lock address], %o5
```

Figure 6 – Pthread Lock in Assembly

It was surprising to discover that Pthread locks are nothing but simple test-and-set locks. It was our expectation that Pthread locks would at least employ a test&test&set algorithm in order to reduce system traffic and added latency by allowing processes to spin on cached lock values. From these results, it is likely that lock-based applications built using Pthreads are likely to have predictable locks.

It is also interesting to observe that the unlock procedure is implemented using a swap instruction. A swap instruction can impact performance in a conventional Out-of-Order processor that implements a relaxed memory consistency model such as TSO, RMO, or RC since the pipeline is forced to flush at any instance of an atomic instruction. Forcing the pipeline to drain at every lock release can have consequences for performance. We concluded that the lock library writers used swap as a release mechanism in order to consolidate two instructions (STORE and MEMBAR) into one. Since MEMBARs are placed optionally to guarantee memory ordering in an RMO or TSO model for a SPARC system, the swap instruction conservatively guarantees appropriate memory ordering for any consistency model.

5.3. Synchronization overhead in DB2 and Oracle

Figures 7 and 8 show the breakdown of execution time for DB2 and Oracle running an OLTP workload separated into system and user mode. The synchronization overheads are the categories highlighted by the boxes. Overall, the synchronization overheads account for a significant fraction (30%) of the execution time. Furthermore, the only case where there is a significant lock spinning overhead is in the DB2 system mode execution. This is perhaps due to the kernel locks that DB2 exposes during its run. Excluding the spinning overhead, the rest of the synchronization overheads can be eliminated by the optimistic lock speculation because lock acquires and releases can be avoided or value-predicted (i.e. speculate ahead while waiting for the lock value to return). Thus, optimistic lock speculation can potentially yield a 14% speedup for DB2 and 13% speedup for Oracle, simply by eliminating the overhead of atomic instructions used in lock acquires and releases.

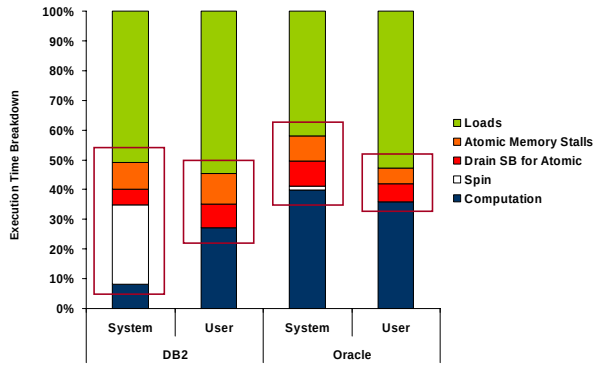


Figure 7. Synchronization Overhead for OLTP running on DB2 and Oracle.

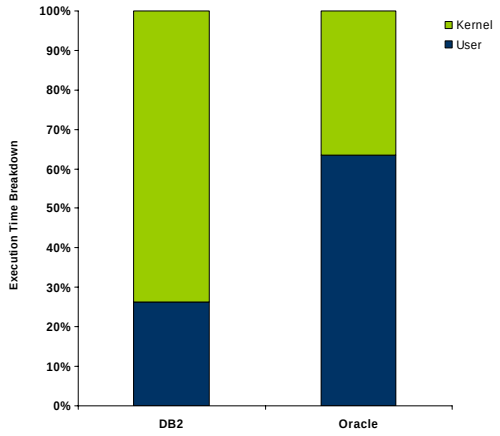


Figure 8. System vs. User time for DB2 and Oracle.

Additional opportunity for further improvement from the fraction of lock spins that do not actually incur data races is open for further study.

5.4. Anatomy of Locks in DB2 and Oracle

This section describes the dominant locking behavior we observed in OLTP on DB2 and Oracle when looking at dynamic traces in the presence of an accurate timing model.

Table 3 shows that in DB2 (Pattern 1), the majority of spin was generated by a single lock address (possibly an OS scheduler) which on average took each processor 100,000 cycles to resolve. The time to do work in the critical section was only 10,000 cycles on average by comparison. The second pattern in DB2 mostly consisted of accesses to various lock addresses with little contention. In Oracle, there was practically no contention for locks with accesses to many different addresses. In all cases, the bulk of locks were made out of ldstub-swap or ldstub-store pairs, implying the dominance of test&set variant locking.

	DB2 Pattern 1	DB2 Pattern 2	Oracle Pattern
Acquire time (cycles)	~100k	~10k	~10k
Release time (cycles)	~10k	~20k	~20k
Critical section (cycles)	~10k	~100k	~100k
Comments	**1 address, highly contended**	**Various addresses, little contention**	**Many addresses, little contention**
	ldstub, swap (75-80%) >> CAS (20-25%)		

Table 3 – Characterization of locking in DB2 and Oracle

This section's analysis allows us to conclude that lock prediction for test&test&set variants is necessary to successfully remove the overhead of atomic instructions and to increase the overall amount of concurrency in the system. In the next section, we discuss lock prediction.

6. LOCK PREDICTION

In this section, we discuss existing techniques for lock prediction and show that they are only able to predict specific classes of synchronization. Finally, we propose a generalized lock prediction mechanism that dynamically identifies critical sections that are protected by *any arbitary locking construct*.

6.1. Silent-Store Lock Prediction

Rajwar et al. showed in [3] that lock prediction for test-and-set locking can be achieved through the detection of silent-store pairs. An observation was made that all lock acquire and release pairs are usually made up of a single store to a lock address followed by a "restoring" operation to the same address. For example, when writing a lock to be held, a processor will write "held" to the lock and "unheld" after it finishes in the critical section. Therefore, a pair of stores to an addxress that effectively keeps the value unchanged can be called a "silent-store pair". By ignoring these stores, program correctness is not affected as long as the lock address values are never used and a silent-store pair actually happened. This invariant guarantees that ignoring two stores to the same address (that is not a lock) will not affect program correctness. To achieve this, a load must be executed on the "silent-store" address and checked later in order to guarantee the invariant.

However, this method is only applicable to simple test&set locks. If a test&test&set lock is used (which is common in the commercial workloads we evaluated), the silent-store pair detection does not eliminate the overhead of the "test" phase, which is simply an ordinary load. Executing this ordinary load can take hundreds of cycles in a distributed shared memory before the remaining test&set can execute.

Furthermore, a lock predictor's purpose is to dynamically identify the beginning and the end of a critical section for the purposes of optimistic lock speculation. Figure 9 shows why SLE may not work in the presence of complicated locking constructs.

In this ticket lock example, the notion of a lock is defined at a higher semantic level by the programmer. The actual

locking primitives (ticket_lock_acquire, ticket_lock_release) are used to form protected access to a ticket counter, which are likely built out of some variant of test&set. In this case, SLE would only elide the process of acquiring and incrementing the ticket. However, each processor will still spin when a node's number is not the same as the now_serving ticket. In

```
Lock_acquire()
    ticket_lock_acquire()
        my_ticket = ticket_issued;
        ticket_issued++;
    ticket_lock_release()

Entering a critical section
while(my_ticket != now_serving) ; //idle loop until my ticket is valid
CRITICAL SECTION

Lock_release()
    now_serving++;
```

Figure 9. Example why SLE does not accurately predict a critical section.

this situation, SLE has failed since processors are executing the critical section in a completely serial manner.

The fundamental problem is that critical sections are not always associated with a corresponding set of atomic primitives. In the previous two cases, SLE is unable to predict the critical sections that test&test&set and ticket locks protect. What is needed is a predictor capable of detecting *entry* and *exit* points to a critical section and not necessarily the associated atomic primitives.

Another disadvantage to this approach is the lack of guarantees on forward progress. Since all processors are speculating, there is no guarantee that all processors will make forward progress in the face of repeated violations in a critical section. To solve this problem, timestamps were used in [] to allow transactions to make forward progress in a serial order.

6.2. Last-value Lock Prediction

A non-silent-store approach for lock prediction is possible through the use of *last-value prediction* on atomic primitives. Two key advantages to this approach is the unnecessary detection of silent-store pairs and guarantees on forward progress. Last-value prediction has been proposed in literature for eliminating data-dependencies in a conventional pipeline [29].

The idea here is to keep a table of last-value predictions for atomic primitives only. For example, an atomic instruction "load-store" that stores the value X to address A and returns the value Y would have a map entry (A, Y) kept on each update to A. The key idea is that when a load-store instruction appears in the pipeline, the hardware switches into lock speculation mode (buffering updates to memory) and proceeds to issue the load-store instruction. In the shadow of the time needed for the load-store instruction to complete (1000s of cycles), the processor proceeds to execute any subsequent instructions. When the load-store instruction returns with the load value, the prediction is checked against the returned value. If the prediction was correct, the processor commits all of its speculative state; otherwise, it re-executes the load-store instruction non-speculatively.

This approach is simple since it only relies on a single value-prediction made on some address when an atomic instruction executes. Furthermore, it guarantees that one processor will make forward progress in the presence of contention since all load-stores are still being executed by all processors. This contrasts to the SLE approach, which does not issue the atomic instruction (e.g. through elision).

However, in the absence of contention, load-value prediction on atomic primitives degenerates to SLE in its ability to predict critical sections. As shown earlier, predicting atomic instructions does not necessarily correlate to speculating into a critical section. Therefore, load-value prediction can only do as well as SLE provided that there is no lock contention.

Load-value prediction would perform worse than SLE in the presence of lock contention. High lock contention (such as the one observed in DB2's kernel lock) would force misspeculation even if there was no real data contention. In this case, SLE can still perform better if a lock's granularity were large enough such that two processors would not be touching the same critical addresses at the same time.

This approach does have some good properties. First, no notion of a lock needs to be present to execute this correctly. It works very well if there are only test&set locks (e.g. a program written in Pthreads). Secondly, no matching "silent-store" pairs need to be detected. Finally, forward progress guarantees can be made since all processors are still issuing all their atomic instructions. However, like SLE, there is still no way to guarantee prediction of a "true" critical section.

6.3. Value-Based Lock Prediction with Spin Detection

This approach is designed to address the shortcomings of the previous two proposals. All forms of locking are centered on some kind of "spinning" on a global (e.g. test&set) or local (e.g. queue-based, ticket-based) variable. This type of spinning can be manifested through repeated execution of atomic instructions (e.g. load-stores) or sequences of loads to some address to spin on. This address may or may not necessarily correspond to a lock variable.

If we assume that all critical sections are preceded by some form of spin code, there is a possible way to predict the presence of a critical section.

In the presence of spin, there can be sequences of either atomic instructions (e.g., load-stores) or loads. If the sequences are made up of atomic instructions, the value-based lock prediction approach (section 6.2) can easily handle this. The more difficult scenario is how to handle a sequence of ordinary loads (e.g. how to distinguish this from regular program execution). A hardware predictor would identify a sequence of loads being executed as a candidate "spin variable" if those loads were being repeatedly sent to the same address and possibly being invalidated (if some other processor changes the value of the lock). Once a candidate

"spin variable" is ascertained, its address and value prediction can be placed in a lock prediction table for future use. What actual value to use will be described later. Once a candidate lock has been chosen and if a future load reappears executing on a previously recorded address and value, a value prediction is made on that load. Presumably, this value prediction would permit the processor to bypass the spin variable and execute within the critical section.

However, speculation can only proceed and finish based on two conditions: 1) if there is no data contention in the critical section, and 2) the load value prediction *eventually* matches the load request.

Condition 1 is the same condition for all lock prediction approaches. Condition 2 is interesting because of the "eventual" outcome. Consider the ticket-based lock example. In the conventional situation, processors would spin by comparing their ticket values to the globally shared ticket until their values matched. This ensures a serial execution of all processors waiting on the spin variable since only one processor can have the matching ticket value at any given time. The key observation is that "eventually" all processors will have a matching ticket value since the globally shared ticket will eventually reach them. Our approach attempts to bypass this by having each processor value-predict past the load that forces them to spin. Specifically, each processor predicts that their load value matches the globally shared ticket. In the absence of data contention, each processor's value prediction will *eventually* become true, albeit possibly some time down the future.

However, it is obvious that only one processor would ever receive a confirmation that their value prediction was correct (e.g. there can only be one true ticket owner). Other processors instead of rolling back and re-executing non-speculatively when they receive non-matching loads, can *ignore* the result returned to them through the load value prediction and *re-issue* the load in hopes that some time in the future, their predictions will match. This of course, also increases the window of vulnerability since true data contention can force multiple processors to rollback. However, to our knowledge, this is the only technique that can reliably detect the presence of a critical section and permit optimistic lock speculation in the face of any kind of locking mechanism. Through similar reasoning, one can show that this approach works for test&test&set variants or array-/queue-based locks, which can possibly generate load sequences in a spin. Both of the previous approaches could not address this type of locking construct.

So far, we have not mentioned how to make accurate value predictions. Ascertaining what value to place in the table requires special care. For example in ticket-based locking, the ticket values that force a processor to spin may be different on each instance of a processor's attempt to access the critical section. For example in Figure 9, this means that the my_ticket value can be different on each instance of acquiring access to the critical section. One way to solve this problem is to realize that the ticket value being compared against is likely to be sitting in a processor's cache already (e.g. the my_ticket variable which is kept in the cache). To create accurate value predictions, dynamic run-time analysis of branch and load patterns can be used to identify candidate cache block values that would be placed in the lock prediction table.

Finally, we discuss the approach for detecting critical sections in the absence of spin. In the ticket- or queue-based approaches, lock prediction is unnecessary in the absence of lock contention. For example, a processor trying to compare a ticket lock would acquire it immediately since it is the only processor accessing the critical section. This leaves behind the case of test&test&set in the absence of lock contention. One way to bypass the critical section would be to continuously monitor any atomic instructions in the dynamic instruction stream. Any addresses and values received while executing atomic instructions are stored in a map table with address value pairs. On subsequent loads (e.g. the test portion of test&test&set), a value prediction can be made. This approach essentially builds upon the technique proposed in section 6.2.

7. IMPLICATIONS FOR FUTURE WORK

Characterizing synchronization behavior in commercial workloads has led us to believe that lock prediction is highly feasible and can potentially yield substantial improvements in performance with no changes to existing software. In light of this, we have proposed a new technique for lock prediction that addresses the shortcomings of previous approaches. In the future, we would like evaluate this new approach by implementing a model in our full-system simulator framework and running it for scientific and commercial workloads.

Having a lock predictor also opens the door to new opportunities such as enhancements for Streaming [] applications. The ability to dynamically identify critical sections permits hardware-based streaming of data in a critical section to next-in-line processors, which could effectively remove all coherence misses in commercial workloads (which constitute nearly 40% of execution time in our timing model).

8. CONCLUSION

In this paper, we presented a characterization of synchronization overhead and locking behavior in commercial workloads. We discovered that on average, 30% of execution time in a modern superscalar processor is spent executing on synchronization. We also discovered that the majority of locks in OLTP on DB2 and Oracle consisted of test&test&set variants and determined that they were highly amenable to lock prediction. We show that straight-forward lock prediction can generate a speedup of at least 14% by removing the overhead of non-contended locks. Finally, we analyzed existing lock prediction approaches and addressed their shortcomings by proposing a Value-based lock predictor with spin detection.

9. REFERENCES

[1] M. Herlihy and J. Eliot B. Moss, "Transactional Memory: architectural support for lock-free data structure," *ISCA*, 1993.

[2] R. Rajwar and J. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *MICRO*, 2001.

[3] R. Rajwar and J. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs," *ASPLOS*, 2002.
[4] J. F. Martinez and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," *ASPLOS*, 2002.
[5] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, Kunle Olukotun, "Transactional Memory Coherence and Consistency," *ISCA*, 2004.
[6] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie, "Unbounded Transactional Memory," To Appear, *HPCA*, 2005.
[7] C. Gniady and B. Falsafi, "Speculative Sequential Consistency with Little Custom Storage," *PACT*, 2002.
[8] V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM: An Execution-Driven Simulator for ILP-based Shared-Memory Multiprocessors and Uniprocessors," *In Third Workshop on Computer Architecture Education*, 1997.
[9] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20-24, March 1995.
[10] M. Galluzzi, V. Puente, A. Cristal, R. Beivide, J. Gregorio, and M. Valero, "A First Glance at Kilo-instruction Based Multiprocessors," *CF*, 2004.
[11] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", *ISCA*, 1995.
[12] J. Martinez, J. Renau, M. C. Huang, M. Prvulovic, J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", *MICRO*, 2002.
[13] P. Ranganathan, K. Gharachorloo, S. Adve, L. Barroso, "Performance of database workloads on shared-memory systems with out-of-order processors", *ASPLOS*, 1998.
[14] H. T. Kung, J. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 6, No. 2, 1981.
[15] Y. Chou, B. Fahs, S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism", *ISCA*, 2004.
[16] D. P. Bovet and M. Cesati, "Understanding the Linux Kernel," O'Reilly, 2001.
[17] A. Silberschatz and P. B. Galvin, "Operating System Concepts," 5th Ed., Addison-Wesley, 1998.
[18] V. Singhal, A. J. Smith, "Analysis of locking behavior in three real database systems," The VLDB Journal, 6: 40-52, 1997.
[19] A. Kagi, D. Burger, J. Goodman, "Efficient synchronization: let them eat QOLB", *ISCA*, 1997.
[20] J. M. Mellor-Crummey, M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February, 1991.
[21] SHORE Project Home Page. URL: http://www.cs.wisc.edu/shore/
[22] R. Love, "Linux Kernel Development", *Pearson Education*, January, 2005.
[23] Paradise Project Home Page. URL: http://www.cs.wisc.edu/paradise/
[24] Predator Project Home Page. URL: http://www.cs.cornell.edu/database/predator/
[25] Dimsum Project Home Page. URL: http://www.cs.umd.edu/projects/dimsum/
[26] Sun Multithreaded Programming Guide. URL: http://docs.sun.com/app/docs
[27] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, A. G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Performance Evaluation Review*, 31(4):31-35, April 2004.
[28] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. "Simics: A full system simulation platform", *IEEE Computer*, 35(2):50-58, February 2002.
[29] M. H. Lipasti, C. B. Wilkerson, J. P. Shen. "Value Locality and Load Value Prediction", *ASPLOS*, 1996.

15-712 Spring 2005 Project Final Report

Transport Protocols and Direct Data Placement: Partitioning Functionality

Alex Nizhner Andrew Biggadike Jad Chamcham

May 11, 2005

Abstract

Transport protocol offload moves all protocol processing from the host CPU down to an intelligent network interface. The main advantage of this architecture is a reduction in host CPU load, since the network interface handles most of the protocol processing tasks traditionally performed by the operating system on the host CPU. However, much existing research shows that the benefit from this reduced CPU load is minimal. Furthermore, network interfaces have their own performance limitations, which makes it difficult for them to keep up with traffic on high-bandwidth network connections. What meager benefits that have been derived from existing protocol offload implementations is attributed almost entirely to the architecture's ability to reduce memory copies through Direct Data Placement (DDP) functionality.

This situation calls for a new protocol offload architecture that is able to provide DDP functionality, yet relieves the reliance on the processing power of the network interface. In this work, we present a *partitioned* protocol offload architecture in which separate portions of the protocol processing are performed on the network interface and the host CPU. These two components work together in order to provide complete protocol processing functionality. We provide a proof-of-concept implementation of this architecture and evaluate its effectiveness. Our results indicate that a partitioned offload architecture is able to maintain throughput in comparison to fully offloaded implementations, but significantly increases RPC response time.

1 Introduction

In the past decade, end-system memory copies and other data touching operations have been identified as the primary bottlenecks in high-bandwidth bulk data transfer [8, 10, 17], leading to the recent standardization of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) protocols and hardware interfaces [4]. Direct Data Placement is recognized by many researchers as the only justification for transport protocol offload: it has been shown that transport offload in itself provides meager benefits and may even backfire, since network interface hardware usually lags behind common general-purpose CPU's in terms of performance [20,25,26]. As it currently stands, complete transport protocol offload has been an unwritten rule guiding the design of DDP-capable network interfaces, and RDMA NIC's typically implement a full protocol stack beneath the direct data placement logic.

We believe that such designs leave room for improvement, since direct data placement functionality does not eliminate the fundamental problem of resource limitations in network interfaces. In this paper, we propose a new method for achieving direct data placement that partitions transport protocol processing between the host system and the network interface. The aim of this new architecture is to reduce the amount of work that must be performed on the network interface, while still achieving direct data placement functionality for host applications. An additional benefit of this approach is a reduction in the amount of resources needed on the network interface, which would create room for other features such as application-specific optimizations. We have developed a proof-of-concept implementa-

tion that partitions UDP/IP processing between the host kernel and a programmable network processor in order to determine the effectiveness of this architecture. Our evaluations indicate that, while the partitioned architecture can maintain throughput in comparison to a fully offloaded architecture, it significantly increases response time. Additionally, since the functionality that logically remains on the network interface is the most significant portion of protocol processing, our implementation did not achieve resource savings as intended.

The rest of this paper is organized as follows. Section 2 describes the related work in this area. This includes both research aimed at determining the shortcomings of transport protocol offload as well as existing DDP solutions. We then provide a detailed description of our system architecture and outline the method by which transport protocol processing is partitioned in Section 3. The methodology and results of our evaluation of the proposed architecture are presented in Section 4. Finally, we conclude and describe future directions for this work in Section 5.

2 Related Work

2.1 Transport Protocol Offload

Transport protocol offload engines[1] implement the transport protocol and the layers beneath within the network interface subsystem, in contrast to traditional software implementations on host CPU's. Proponents of transport offload have argued that such solutions reduce the load placed on host CPU's by protocol processing tasks, which can be substantial for gigabit and faster networks. However, TCP offload engines have repeatedly fallen short of expectations; Mogul [20] discusses several reasons for their failure.

The most prominent of these reasons are fundamental performance limitations of network interface hardware, which typically lags a year or more on the Moore's Law curve behind high-speed host CPU's. As a result, such computationally-limited offload engines can actually reduce performance. Sarkar *et al.* [25] demonstrate this phenomenon in the context of an iSCSI workload; the analysis presented by Shivam and Chase [26] shows that the benefits of protocol offload are bounded by the the disparity in processing power between offload hardware and the host CPU.

Another reason is the structure of the protocol processing workload. Transport offload engines operate under the assumption that protocol processing is inherently computationally expensive, which over the years has been shown to be largely untrue. The landmark paper by Clark *et al.* [11] shows that the TCP fast path requires comparatively few instructions (excluding operating system overhead) and that overall TCP overhead is dominated by copy and checksum operations. Recent work [8, 10, 15] confirms this result; it is now widely believed that protocol processing in itself is not a major source of overhead for bulk transfer workloads, and that the cause of the end-system bottleneck lies in the per-byte costs.

In light of the above, Mogul suggests that transport protocol offload is effective only as an enabler of DDP-capable network interfaces, "since to rely on the host OS stack would defeat the purpose" [20]; Shivam and Chase formalize this notion in terms of structural improvements to the offload architecture that eliminate some of the overheads "rather than merely shift them to the NIC" [26]. It is interesting to note that Mogul mentions simple NIC extensions (*e.g.,* hardware checksumming and copy reduction support) that can enable very efficient transport implementations, albeit not capable of true zero-copy, but stresses the need for full transport offload beneath DDP logic. We believe that the partitioning of protocol processing on which such devices rely—namely, outboard buffering and checksumming [17,29] are applicable to DDP-capable network interfaces as well.

2.2 Existing DDP Hardware

Direct data placement hardware available today typically falls in two domains. One can be roughly categorized as Memory-to-Memory interconnects for high-performance distributed applications, and uses the RDMA protocol [4]. The other is IP block storage, in which solutions are offered in the form

[1] In this context, the offloaded transport protocol is typically TCP or another reliable transport. Since we are focusing on the UDP protocol, we will restrict our attention to features these protocols have in common.

iSCSI Host Bus Adapters (HBA's); often a single network interface can support both classes of protocols. Though both types of applications assume a reliable transport layer beneath the DDP logic (*e.g.,* [12, 13]), these NIC's do not necessarily feature a fully-offloaded transport; indeed, some designs partition protocol processing functionality in ways similar to our proposal.

We first mention some of the full-offload solutions on the market. The Chelsio T110-CX Protocol Engine [9] offers support for both iSCSI and RDMA, and features nearly complete transport protocol offload, including TCP congestion control, connection setup and teardown, IP path MTU discovery—in addition, of course, to the complete data path. The LeWiz Communications Magic2020 Multi-Port HBA [18] is similar, and offloads absolutely all aspects of transport protocol processing. These solutions are extreme; they lie at the opposite end of the spectrum from software-based transport implementations.

More similar to our work are solutions that offload only the protocol processing fast path, typically based on the Microsoft Chimney offload architecture[2]. The Broadcom BCM5706 Ethernet Controller [5] provides RDMA and iSCSI functionality, and offloads the TCP fast path including checksumming and segmentation; the slow-path protocol processing tasks execute in a host software driver. The NetEffect NE01 iWARP Ethernet Channel Adapter [21] operates similarly, as do the Alacritech Accelerator products [2, 3]. The main difference between these solutions and our proposal is that we advocate partitioning the protocol fast path as well—*i.e.,* we envision non-data-intensive fast-path operations executing on the host processor, with the network interface handling only those operations for which host CPU architectures are poorly suited.

3 Architecture

In this section we summarize the architecture of the complete system. The description given here is mostly normative: the only design issues and tradeoffs we discuss relate to the placement of transport protocol processing functionality. In the interest of brevity, we avoid discussing design decisions pertaining to other system components, communication interfaces, and overall programming model; instead, we treat those features as immutable and simply describe their structure.

The remainder of this section consists of two parts. Section 3.1 serves to provide architectural background and context for our work, and focuses on intelligent data placement proper. In Section 3.2, we describe our variations on the basic architecture in support of transport protocol functionality partitioning, and justify those variations as appropriate.

3.1 Baseline Architecture and Programming Model

Our project is being carried in the context of intelligent data placement (IDP) research [16], from which we borrow the basic architectural framework. Intelligent data placement can be thought of as a generalization of RDMA [4] *hardware*—as opposed to the RDMA protocol suite. A network with intelligent data placement support does not prescribe a particular wire protocol, and in particular does not require the exchange of buffer references among connection endpoints in order to achieve direct data placement, as is the case with, *e.g.,* RDMA-based distributed filesystems [6, 14]. Instead, IDP-capable networks strive to support non-RDMA-aware peers using standard wire protocols such as NFS [7,27,28]. This requires that an IDP NIC parse the conventional upper-level wire protocol in order to infer destinations in host memory locally, which in turn requires the NIC to have knowledge of the layout of application data structures and their locations in physical memory.

These requirements are realized by partitioning an application on the IDP-capable host into components executing in two distinct domains: the host processor and what is referred to in [16] as the *data engine*. The components executing on the data engine (in our case, the embedded processor on the NIC) are first-class "delegates" of the application: they encapsulate knowledge of one or more application data structures and associated wire protocols, and cooperate with the application in manipulating those data structures. We refer to these components as the application's *data proxies*. A data

[2]The Microsoft Chimney API is not scheduled to come out until mid-2005; no reference is available at this time.

proxy is primarily responsible for orchestrating the direct placement of bulk data into the application's address space without host CPU involvement. Additionally, it can perform application-specific data-intensive tasks that do not necessarily require the data to traverse the I/O bus, *e.g.,* computing simple digests; indeed, some data proxies may perform no DDP at all.

There are two types of communication that take place between a data proxy and its parent application: streaming bulk data transfers and the exchange of control, synchronization, and application-logic-related information. These are decoupled in the spirit of Thekkath [30], though on a much tighter scale, with the I/O bus replacing the network. All synchronization and control transfer is in the form of message passing; we do not consider shared address space architectures at the present. The content of messages exchanged in this fashion is entirely application-specific—the system provides only a generic message queue abstraction.

3.1.1 Target Applications

Although a large cross-section of data-intensive distributed applications can benefit from intelligent data placement hardware, RPC-based applications constitute our primary target. The XDR-encoded [28] ONC RPC [27] wire protocol is fairly straightforward to parse; moreover, the NFS protocol family [7] is RPC-based. This project therefore focuses on RPC server applications, which lend themselves to a natural data proxy partitioning.

A typical RPC server can be thought of a dispatch loop which accepts framed procedure invocations from the underlying transport, extracts procedure IDs and unmarshalls the XDR-encoded parameters, invokes the corresponding procedure, marshals the result, and finally transmits the response to the client. We cast this process into the IDP framework as follows. The RPC server's data proxy assumes responsibility for unmarshalling procedure arguments, separating those that require direct placement from the rest and depositing them into pre-posted application buffers (*e.g.,* NFS `write` file data is placed into pre-registered buffer cache entries), and signaling the host server process with the RPC transaction and procedure IDs, inline arguments, and references to application buffers into which bulk arguments had placed. Additionally, the proxy maintains transient state for the RPC transaction—in particular, it pre-allocates NIC memory (the *staging buffer*) for the RPC response, into which the host server can deposit bulk data for transmission. Upon receipt of a message, the host server simply extracts the necessary information and dispatches the application handler; when the procedure completes, the server signals the proxy with the location of bulk return data, the remaining results and the return status. The proxy completes the transaction by marshalling the results and transmitting the response back to the client.

In this paper, we focus on a simple RPC server we call `NFS-lite` that mimics the wire protocol and data path of a full-featured NFS server. The `NFS-lite` protocol supports the following RPC procedures:

- `void write(file, offset, length, data<>);`
- `data<> read(file, offset, length);`

where `file` is a unique file identifier similar to the NFS file handle. In response to a `write` call, an `NFS-lite` server deposits the opaque file data from the RPC argument buffer into a data structure reminiscent of the Linux page cache, wherein each file is represented in memory as a collection of discontiguous page-sized blocks. The SunRPC implementation performs this function by means of one or more `memcpy` operations. The IDP implementation performs it in two steps: first, the data proxy queries the host server for the physical addresses of page cache entries corresponding to the given `file`, `length`, and `offset`; then, it initiates a scatter/gather DMA of the data payload into those buffers and notifies the host server of completion. Note that the IDP implementation is thus completely copy-free. (The implementation of the `read` procedure is symmetric in both cases.)

3.1.2 Hardware Platform and Software Module Decomposition

Our prototype IDP-capable network interface is the ENP-2611 PCI board from Radisys [24], controlled by the Intel® IXP2400 network processor [1]. The IXP2400 contains 8 RISC-like 8-way

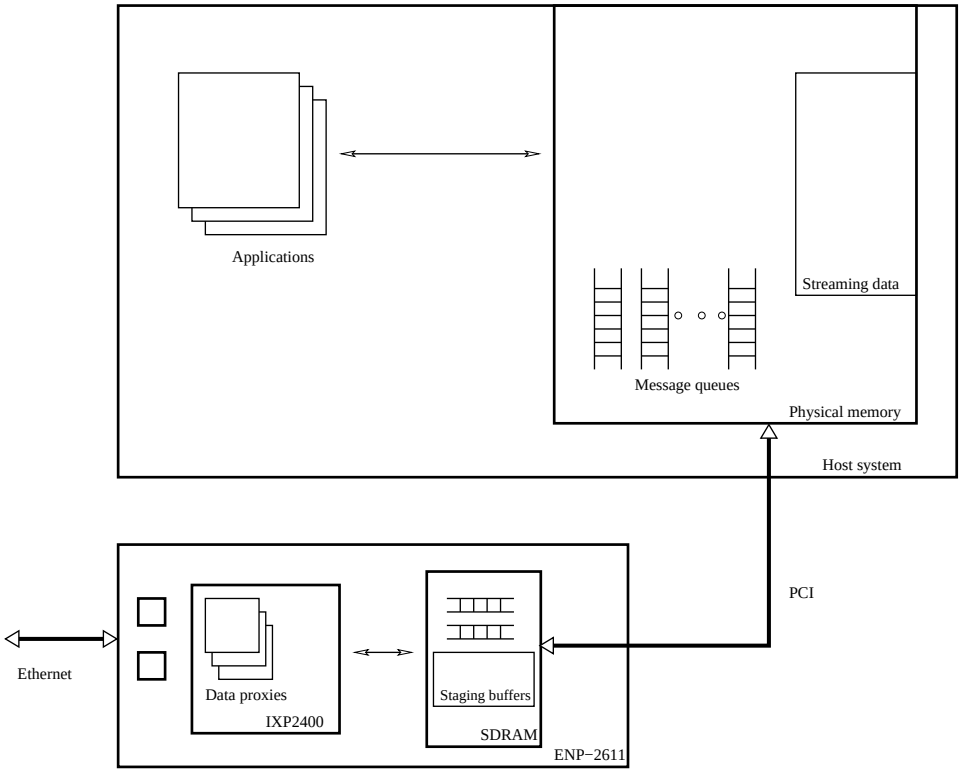


Figure 1: **Hardware Platform and Basic Architecture.** Data proxies execute on the IXP2400 microengines, staging I/O to the host or the link from SDRAM-based buffers. Remote memory is never read by either the applications or their proxies; all data are written remotely and read locally. Control and synchronization traffic is in the form of short messages and is kept separate from streaming bulk data.

multithreaded CPU cores known as microengines, well-suited to perform memory-intensive operations, and an Intel® XScale control processor. The microengines typically communicate using hardware-assisted rings in the 16K on-chip scratchpad memory. The IXP2400 also contains a PCI controller which allows external access to ENP-2611 memory resources, and provides three DMA channels for access to host memory. The ENP-2611 features 256 MB of DDR SDRAM, 8 MB of QDR II SRAM, and three Gigabit Ethernet ports controlled by dual PCM Sierra PM3386 MAC devices. The board connects to the host's PCI subsystem via an Intel® 21555 nontransparent PCI/PCI bridge.

Figure 1 shows how the conceptual IDP architecture outlined above maps to the ENP-2611 platform. Unsurprisingly, the IXP2400 microengine become the data engines on which the data proxies execute; the XScale control processor does not participate in the data path and is used for initialization only. The ENP-2611 SDRAM, normally used for streaming packet data, contains the staging buffers. Due to the peculiarities of the PCI bus, the proxy-to-application messaging infrastructure is partitioned across the system's memory resources: queues carrying messages from data proxies reside in host memory, whereas those carrying messages from host applications are placed in ENP-2611 SDRAM, making all cross-PCI communication take place in the form of writes.[3]

The software infrastructure supporting this system is partitioned as follows. Besides the data proxies, the IXP2400 dedicates additional microengines to (1) Ethernet packet reception and buffering, (2) Ethernet packet transmission, and (3) IXP2400 DMA engine management. These modules (subsequently referred to as *Ethernet Receive*, *Ethernet Transmit*, and *DMA Manager* each occupy a

[3]The queues read by data proxies can also be placed in ENP-2611 SRAM in order to reduce latency.

single microengine. The *Ethernet Receive* module, in addition to interacting the IXP2400 link interface for packet reception, is responsible for demultiplexing incoming application-level frames and assigning them to contiguous staging buffer storage; this is accomplished using a light-weight packet filter mechanism [19]. The *DMA Manager* module is responsible for placing bulk data into host-memory using the IXP2400 DMA engines in response to data proxy requests. It also has the capability of placing a data proxy's message into the appropriate application queue upon the completion of a DMA transfer, for the purposes of synchronizing the host's access to these data. Of particular interest is the *UDP/IP* module; it is responsible for all UDP/IP processing, may occupy multiple microengines, and the transport protocol processing partitioning we have implemented takes place here. This decomposition of IXP2400 firmware and its organization into a software data path is illustrated in Figure 2-a.

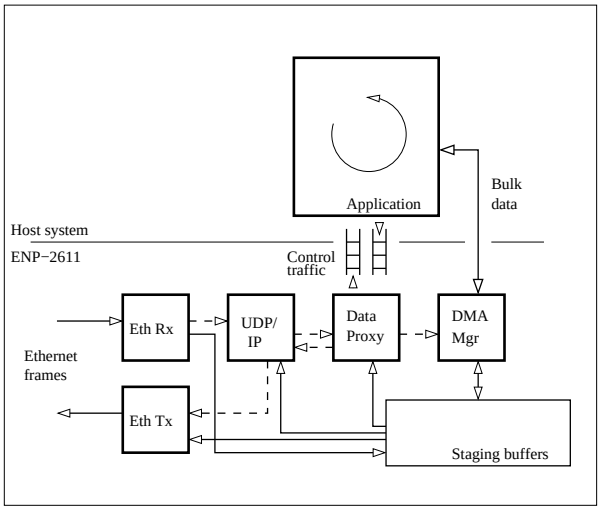
These details are abstracted from host applications behind a *messaging framework* and a light-weight *resource management infrastructure*. The latter provides host applications with protected user- or kernel-space access to ENP-2611 memory resources and facilities for the allocation of DMA buffers in host memory. The messaging framework builds on the resource management infrastructure to provide a bare-bones message delivery service with polling-based notification.

3.2 Transport Protocol Placement

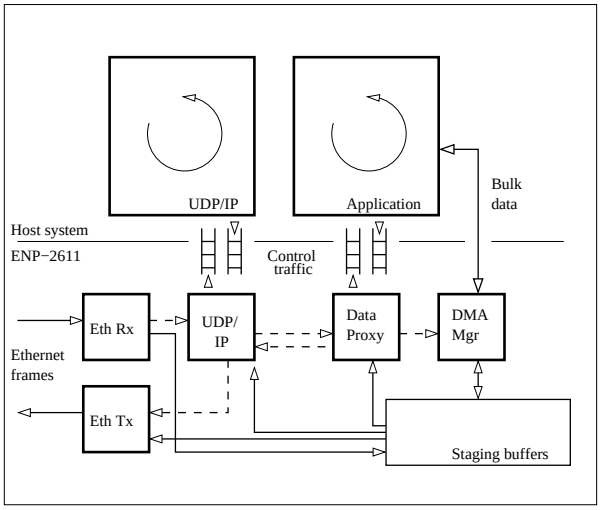
3.2.1 Protocol Processing Requirements

Our target workload consists of NFS traffic, which contains RPC commands transferred over UDP in several IP fragments. This composition of protocols requires a various levels of protocol processing within our system.

At the network layer, we must perform IP header validation and fragment reassembly in accordance with the IP specification [23]. Header validation requires ensuring the checksum is correct and that the destination address applies to the network interface. Fragment reassembly requires maintaining data structures to determine when all portions of the packet have arrived, as well as keeping a timer



(a) Full Offload



(b) Minimal Offload

Figure 2: **Data Path Comparison.** (a) shows the components along the data path for a fully offloaded architecture and (b) shows the components required for the minimal offloaded architecture. Solid lines indicate bulk data flow and dashed lines indicate control messages. The vertical queues indicate control traffic between the ENP-2611 and the host system over PCI.

to ensure fragments of uncompleted packets can be freed after a timeout period.

At the transport layer, we must perform UDP header and payload validation according to the UDP specification [22]. The UDP checksum must be computed across the actual UDP header, its pseudo-header, and the entire UDP payload. Additionally, it is necessary to forward the payload to the correct application based on the destination port.

At the application layer, we must inspect the RPC command as specified in the RPC specification [27]. Once the RPC command is known, it is possible to perform direct data placement into preregistered host buffers.

3.2.2 Placement of Functionality

The protocol processing functionality can be located in various components within the system. A traditional network stack is implemented entirely on the host system within the operating system kernel. This approach is appealing for its straightforward design, but usually requires excessive memory copies, thus decreasing performance. An extreme alternative to this approach is to have complete protocol offload onto the network interface, which places the necessary intelligence to perform protocol processing and data placement directly on the network interface's network processors. Our architecture focuses on utilizing an approach that falls between these two, partitioning the protocol processing so that portions are performed on both the host and network interface. The aim of this partitioning is to minimize the protocol offload to the network interface, while still achieving the benefits of direct data placement that comes with the fully offloaded approach. Architectural diagrams showing each of the components involved for the full and minimal offload approaches are shown in Figure 2.

The full offload approach is shown in Figure 2-a. As shown in the diagram, there is a separate component for each of the required protocols. The *Ethernet Receive* module receives the data frame on the network link and provides it to the *UDP/IP* module; symmetrically, the *Ethernet Transmit* module places data received from the *UDP/IP* module on to the network link. The *UDP/IP* module handles all checksum calculation and verification, header validation, and fragment creation and reassembly. The *Data Proxy* module on the network interface determines the RPC command, marshalls and unmarshalls arguments, and communicates with the host application via PCI control messages to determine buffer addresses reading and placing data. A separate *DMA Manager* module will then handle the bulk data transfer to host memory. Note that the only processing done on the host system is the determination of the buffer addresses.

The minimal offload approach is shown in Figure 2-b. The *Ethernet Receive*, *Ethernet Transmit*, *Data Proxy*, and *DMA Manager* modules have the same functionality in both approaches. The key difference in this architecture is that the transport protocol processing located in the *UDP/IP* module has been partitioned between a microengine on the network interface and a module running within the host kernel. Data intensive operations including calculating and verifying IP header and UDP payload checksums are performed on the network interface. The host *UDP/IP* module handles all other operations, namely the computational tasks of fragment creation and reassembly[4] and determining the correct application based on the IP addresses and UDP ports. Together, these two components perform all of the UDP and IP processing necessary. A single message is sent over the PCI bus from the network interface to the host module that contains the necessary header fields required for the host to complete its processing tasks. When done, the host sends a message back to the network interface so that it can drop the packet or forward it to the correct location as appropriate. This message exchange adds a single round-trip over the PCI bus to the protocol processing procedure. We conjecture that the reduction in *UDP/IP* functionality that is performed on the network interface would enable merging the *Ethernet Received* and *UDP/IP* microengines, thus saving resources and decreasing the reliance on the processing power of the network interface.

[4]Due to the decreased importance of IP fragmentation on today's networks, the proof-of-concept implementation did not implement this feature. We include it in our architecture description for completeness.

4 Evaluation

Below, we perform a comparative evaluation of the IDP implementation of the `NFS-lite` server with a partially-offloaded UDP stack against the Sun RPC implementation, and an IDP implementation with a fully-offloaded stack. Our chief metrics of interest include (a) the base response time on an unloaded system, (b) the maximum achievable application-level data throughput, and (c) ENP-2611 buffer space usage. We report results for the following two classes of experiments:

- *Single-client Null RPCs.* The client performs a large number of consecutive null RPC calls.
- *Multi-client MTU-sized `NFS-lite` writes.* The client machines each run a number of independent threads generating consecutive MTU-size `NFS-lite` write calls.

We present no results for `NFS-lite` read performance due to the fairly dismal PCI read bandwidth of which our experimental system is capable.

4.1 Experimental Setup

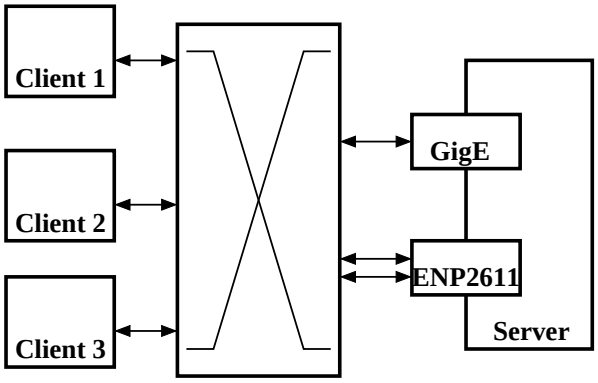


Figure 3: **Experimental Network Topology.** The server machine hosts the ENP-2611 and a traditional Gigabit Ethernet NIC. The clients connect to server NICs via a Gigabit Ethernet switch.

The experimental setup is shown in Figure 3. The server machine is an Intel® Pentium IV clocked at 1.5 GHz, with a 256 KB L2 cache, a 100 MHz front-side bus, and 256 MB of ECC RDRAM clocked at 400 MHz; the server runs SuSE Linux 9.0 with a 2.4.21 kernel. The server hosts both the ENP-2611 for IDP traffic and a D-link DGE550-SX Gigabit Ethernet fiber adapter for Sun RPC traffic. For measurements of `NFS-lite` performance over Sun RPC and IDP with a fully-offloaded stack, the `NFS-lite` server is the highest-priority runnable process in the system. For partially-offloaded-stack IDP experiments, the `NFS-lite` server and the host-based UDP stack kernel thread are the highest-priority runnable processes.

Table 1: **Null RPC Response Time.**

Sun RPC	118 μs
IDP (full)	68 μs
IDP (partial)	1000 μs

The server's PCI bus is 32 bits wide, clocked at 33 MHz. We have measured a maximum sustained bandwidth of 680 Mbps for PCI writes from the ENP-2611 to the host PCI subsystem, and a short-message roundtrip time from the host to the ENP-2611 of 3 μs.

We use three client machines featuring the same hardware configuration as the server for generating `NFS-lite` traffic. The clients were observed to saturate their transmit paths at roughly 300 Mbps application-level throughput. The `NFS-lite` client implementation is Sun RPC-based, and is the same regardless of the server implementation. The clients connect to the server's NICs via a Gigabit Ethernet copper switch; we use Milan fiber media converters to connect D-link and ENP-2611 optical ports to the copper ports on the switch.

4.2 Null RPC Response Time

We measure the null RPC response time for the three `NFS-lite` server implementations in order to gauge the base amount of time an RPC call spends in the system. This latency directly affects the number of independent clients required to saturate the system: the synchronous nature of RPC requires an independent client for each outstanding request. In turn, the number of simultaneous outstanding requests directly affects packet buffer space usage, as we demonstrate below.

Null RPC response times for the three `NFS-lite` server implementations are summarized in Table 1. We observe a factor of 15 increase in the base RPC response time for the implementation with a partially-offloaded stack compared to that with a fully-offloaded stack. We attribute this increase primarily to the peculiarities of the Linux scheduler—more specifically, scheduling granularity and context switch overhead.

4.3 `NFS-lite` Write Throughput

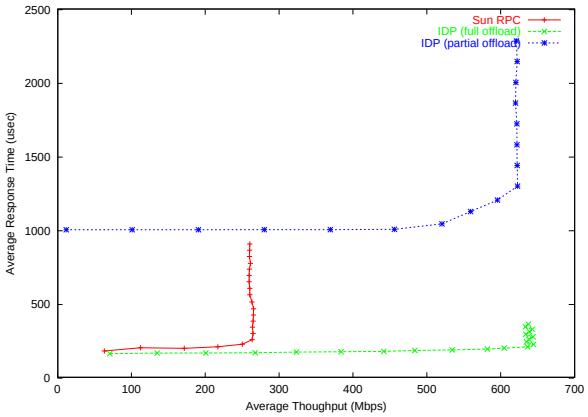
In this experiment, we evaluate the maximum achievable write throughput for the three `NFS-lite` server implementations. Figure 4 shows the observed throughput scaling properties.

We observe that the Sun RPC implementation saturates at 263 Mbps application-level throughput, being fundamentally limited by the host CPU at this data rate (CPU utilization numbers are not shown). In contrast, both IDP implementations saturate the PCI bus—the fully offloaded transport implementation at 644 Mbps, and the partially-offloaded implementation at 622 Mbps. (The figure of 680 Mbps sustained PCI write bandwidth quoted earlier refers to the *raw* bandwidth, and does not take into account control and synchronization traffic; the partially-offloaded implementation adds a control message round-trip to the data path.)

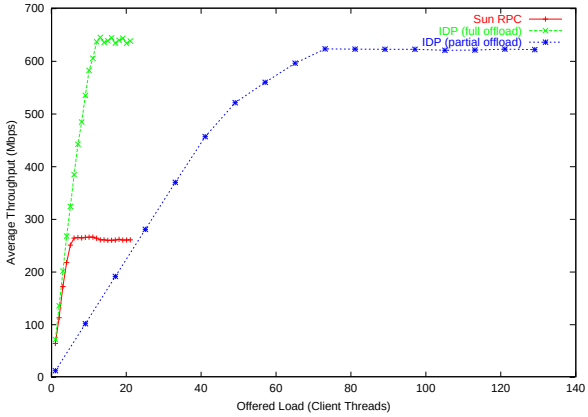
Quite noteworthy is the base RPC response time for the three implementations, and consequently the number of client threads required to achieve peak throughput. The partially-offloaded implementation requires 73 client threads in order to saturate the PCI bus, for a response time of 1295 μs at a peak throughput of 622 Mbps—both a factor of 5 greater than 13 client threads and a 221 μs response time at 644 Mbps write throughput for the fully-offloaded implementation, as shown in Table 2. In the next section, we examine the effect on this phenomenon on packet buffer space usage.

4.4 Resource Usage

We now consider the effect of partitioning the UDP stack on ENP-2611 buffer space utilization. Figure 5 shows buffer space usage for the two IDP im-



(a) Throughput Saturation: Response Time Growth



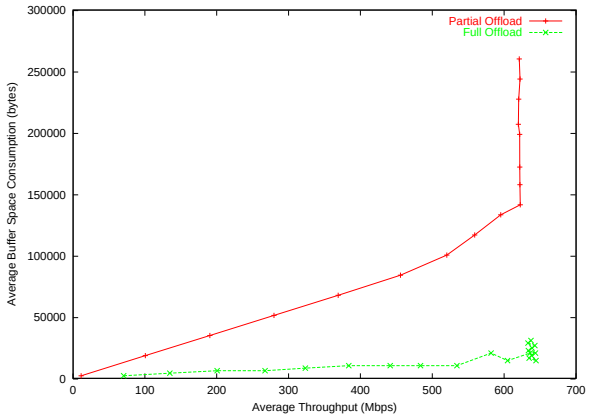
(b) Throughput Saturation: Offered Load

Figure 4: `NFS-lite` **Throughput Scaling.** Figure (a) shows the throughput saturation point for the three implementations in terms of the `write` response time; Figure (b) presents the same data in terms of the number of independent client threads generating `write` calls. The Sun RPC implementation saturates the CPU at 263 Mbps; both IDP implementations saturate the PCI bus at 644 and 622 Mbps, respectively. The IDP implementation with a partially-offloaded transport requires many more client threads to achieve the same level of throughput as the fully-offloaded-transport implementation.

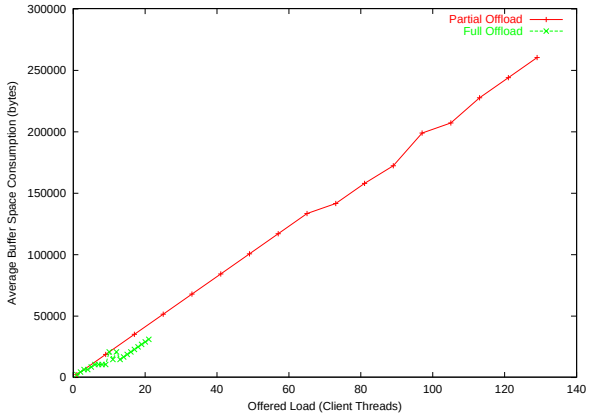
Table 2: **Maximum Write Throughput.**

Impl.	T-put	Resp. Time	Clients
Sun RPC	263 Mbps	255 μs	6
IDP (full)	644 Mbps	221 μs	13
IDP (part)	622 Mbps	1295 μs	73

plementations of `NFS-lite`. We observe that buffer space usage is directly proportional to the number of simultaneously-active independent clients, as shown in Figure 5b. A direct consequence of this fact, illustrated in Figure 5a, is that the partially-offloaded-stack implementation requires 5 times the buffer space of the fully-offloaded version at the same level of application throughput.



(a) Buffer Space Usage: Average Throughput



(b) Buffer Space Usage: Offered Load

Figure 5: `NFS-lite` **ENP-2611 Buffer Space Usage.** Each outstanding request in the system corresponds to a 2048-byte packet buffer; total buffer space usage is thus determined by the number of simultaneous outstanding requests, or client threads, as shown in (b). Figure (a) shows buffer space usage for a given level of application data throughput.

4.5 Discussion

We see that both IDP implementations of `NFS-lite` maintain comparable peak throughput over twice that of the SunRPC implementation. The throughput observed for the IDP implementation with a partially-offloaded UDP stack is marginally lower than that of the fully-offloaded implementation; this is due to the fact that the partially-offloaded implementation requires more PCI traffic in support of protocol processing, and this additional traffic consumes a small fraction of the raw PCI bandwidth. This difference in throughput among the IDP implementations, albeit small, leads us to believe that PCI is not the right interconnect for IDP applications—as the complexity of IDP applications grows, the amount of control traffic (perhaps in the form of shared memory accesses) can be expected to grow as well, and should not adversely affect application data rates.

On the other hand, we see that our implementation of the partitioned UDP stack significantly increases the base RPC response time, and that buffer space usage in the ENP-2611 increases proportionally. We believe that the UDP stack in the partially-offloaded solution should not be implemented as a regularly-scheduled thread polling for frame arrivals, since we hold Linux scheduling granularity and context switch overhead to be the primary reasons for the increased latency. Instead, some combination of polling and interrupts might serve better, or a single-process implementation in which the UDP stack is combined with the application.

We are unable to demonstrate either performance improvements or resource savings we expected to achieve with a partially-offloaded stack implementation. The main reason for this is the minimal nature of UDP: the UDP payload checksum is the only computationally-expensive element of UDP processing. We expected to be able to implement the ENP-2611-resident portion of the UDP stack in the partially-offloaded version in fewer instructions than the fully-offloaded implementation, and perhaps to even combine it with the link receive and transmit logic. We have found, however, that UDP checksum computation is the most expensive element of the UDP pipeline not only in terms of running time, but also instruction counts; since the UDP checksum must be computed by the NIC in both implementations, significant code space savings are not possible

without hardware support. Whereas it is trivial to compute the incremental UDP checksum in hardware for arriving packets (in fact, the IXP2400 MSF subsystem has this feature), adding this logic to the transmit path is not easy, since the UDP checksum resides in the UDP header. We believe that a PCI bus interface with a capability to compute an incremental 16-bit ones-complement checksum would allow a truly light-weight NIC-resident UDP stack implementation.

5 Conclusions and Future Work

In this paper, we have shown that a network interface need not implement a full transport protocol stack in order to effectively perform Direct Data Placement. Our implementation of a prototype NIC supporting Intelligent Data Placement with a partially-offloaded UDP stack achieves throughput comparable to the throughput achieved by a similar implementation with a fully-offloaded UDP stack. On the other hand, our implementation exhibits much longer RPC response times, and consequently uses much more packet buffer memory than the fully-offloaded implementation—a phenomenon we attribute to Linux task scheduling granularity and context switch overhead.

In the future, we would like to experiment with alternative implementations that avoid that overhead—e.g., combining the UDP stack with the RPC server process, or using a combination of polling and interrupts in the UDP pipeline. Additionally, we would like to examine a more elaborate transport protocol than UDP and attempt to show the computational resource savings we listed as a goal of this work.

Finally, we intend to construct a full-featured NFS server implementation within the IDP framework in order to be able to perform a more meaningful end-to-end performance evaluation with a traditional filesystem benchmark suite. Going forward, we would also like to investigate application classes other than RPC servers.

Acknowledgements

The authors thank Professor Garth Gibson and his fearless teaching assistant Hyang-Ah Kim for a wonderful semester. Thanks also to Larry Huston and Peter Steenkiste for the resources and many useful discussions.

References

[1] Intel Network Processors. `http://developer.intel.com/design/network/products/npfamily/index.htm`.

[2] Alacritech. Alacritech Accelerator Product Data Sheet. `www.alacritech.com`.

[3] Alacritech. Alacritech SES1001 iSCSI Accelerator Product Data Sheet. `www.alacritech.com`.

[4] Bailey, S., and Talpey, T. The Architecture of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) on Internet Protocols. Internet Draft *draft-ietf-rddp-arch-06*, 2004.

[5] Broadcom. BCM5706 Product Brief. `www.broadcom.com`.

[6] Callaghan, B., Lingutla-Raj, T., Chiu, A., Staubach, P., and Asad, O. NFS over RDMA. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence* (2003), ACM Press, pp. 196–208.

[7] Callaghan, B., Pawlowski, B., and Staubach, P. NFS Version 3 Protocol Specification. RFC 1813, June 1995.

[8] Chase, J., Gallatin, A., and Yocum, K. End-System Optimizations for High-Speed TCP. *IEEE Communications Magazine 39*, 4 (2001), 68–74.

[9] Chelsio. T110-CX 10GbE Protocol Engine Product Data Sheet. `www.chelsio.com`.

[10] Chu, H. K. J. Zero-Copy TCP in Solaris. In *USENIX Annual Technical Conference* (1996), pp. 253–264.

[11] Clark, D., Jacobson, V., Romkey, J., and Salwen, H. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine 27*, 6 (June 1989).

[12] DAT Collaborative. kDAPL API Specification Version 1.2. `www.datcollaborative.org/kdapl.html`, 2003.

[13] DAT Collaborative. uDAPL API Specification Version 1.2. `www.datcollaborative.org/udapl.html`, 2004.

[14] DeBergalis, M., Corbett, P., Kleiman, S., Lent, A., Noveck, D., Talpey, T., and Wittle, M. The Direct Access File System. In *Proceedings of the USENIX FAST '03 Conference on File and Storage Technologies* (San Francisco, CA, April 2003), USENIX Association.

[15] Foong, A. P., Huff, T. R., Hum, H. H., Patwardhan, J. P., and Regnier, G. J. TCP Performance Re-Visited. In *ISPASS-2003 International Symposium on Performance Analysis of Systems and Software* (Austin, Texas, United States, Mar. 2003).

[16] Huston, L. Application-Controlled Data Placement. In *1st Workshop on Building Block Engine Architectures for Computers and Networks (BEACON 2004)* (Oct. 2004).

[17] Kleinpaste, K., Steenkiste, P., and Zill, B. Software Support for Outboard Buffering and Checksumming. In *SIGCOMM* (1995), pp. 87–98.

[18] LeWiz Communications. Magic2020 Multi-Port HBA Product Data Sheet. `www.lewiz.com`.

[19] Mogul, J., Rashid, R., and Accetta, M. The Packet Filter: an Efficient Mechanism for User-Level Network Code. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles* (1987), ACM Press, pp. 39–51.

[20] Mogul, J. C. TCP offload is a dumb idea whose time has come. In *HotOS IX: The 9th Workshop on Hot Topics in Operating Systems* (Lihue, Hawaii, May 2003).

[21] NetEffect. NE01 Product Data Sheet. `www.neteffect.com`.

[22] Postel, J. User datagram protocol. RFC 768, Aug. 1980.

[23] Postel, J. Internet protocol. RFC 791, Sept. 1981.

[24] Radisys. ENP-2611 Product Data Sheet. `www.radisys.com`.

[25] Sarkar, P., Uttamchandani, S., and Voruganti, K. Storage over IP: When Does Hardware Support Help? In *Proceedings of the USENIX FAST '03 Conference on File and Storage Technologies* (San Francisco, CA, April 2003), USENIX Association.

[26] Shivam, P., and Chase, J. S. On the Elusive Benefits of Protocol Offload. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence* (2003), ACM Press, pp. 179–184.

[27] Srinivasan, R. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Aug. 1995.

[28] Srinivasan, R. XDR: External Data Representation Standard. RFC 1832, Aug. 1995.

[29] Steenkiste, P. A Systematic Approach to Host Interface Design for High-Speed Networks. *IEEE Computer 27*, 3 (1994), 47–57.

[30] Thekkath, C. A., Levy, H. M., and Lazowska, E. D. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, 1994), pp. 2–11.

A Searchable-by-Content File System

Srinath Sridhar, Jeffrey Stylos and Noam Zeilberger

May 11, 2005

Abstract

Indexed searching of desktop documents has recently become popularized by applications from Google, AOL, Yahoo!, MSN and others. However, each of these is an application separate from the file system. In our project, we explored the performance tradeoffs and other issues encountered when implementing file indexing *inside the file system*. We developed a novel virtual-directory interface to allow application and user access to the index. In addition, we found that by deferring indexing slightly, we could coalesce file indexing tasks to reduce total work while still getting good performance.

1 Introduction

Searching has always been one of the fundamental problems of computer science, and from their beginnings computer systems were designed to support and when possible automate these searches. This support ranges from the simple-minded (e.g., text editors that allow searching for keywords) to the extremely powerful (e.g., relational database query languages). But for the mundane task of performing searches on users' files, available tools still leave much to be desired. For the most part, searches are limited to queries on a directory namespace—e.g., Unix shell wildcard patterns are useful for matching against small numbers of files, and GNU locate for finding files in the directory hierarchy. Searches on file *data* are much more difficult. While from a logical point of view, grep combined with the Unix pipe mechanisms provides a nearly universal solution to content-based searches, realistically it can be used only for searching within small numbers of (small-sized) files, because of its inherent linear complexity.

However, fast searches on file content are possible if an indexing structure is maintained. Recently, programs that build such indexing structures to allow fast searches have attracted a lot of attention. Popular examples are Google Desktop Search and X1 Desktop Search.

For our project, we designed and implemented a *file system* for efficient data-indexed searches. When text files are written, the file system automatically

indexes the files and the file system provides a virtual directory structure for users and applications to access the index.

There are many advantages to having a data-indexing file system rather than specialized programs that sit on top of it, and one of the motivations for our project was to explore the design space. In particular, we were interested in issues such as ensuring consistency of the index with respect to the actual data in the file system, performance tradeoffs, and cross-application index sharing.

An important feature of our system is that we update the index incrementally, almost immediately after a file modification. Since our indexing scheme is built into the file-system it is easy to be notified when a file is modified. One important practical result of such a dynamic scheme is that the index is always consistent and up-to-date. Though the usefulness is obvious, it is hard to measure the significance of having a current index.

Also difficult to measure are the benefits that come from having a centralized index. As described in Section 2.1, many current applications build and keep track of partial indexes of some portion of a user's data. By providing an indexing service from within the file system, applications do not have to keep track of indexes themselves (which involves tracking when files are moved, deleted, renamed, created, their contents changed, and other complexities), simplifying the applications and allowing more new applications to take advantage of content indexing for speed and functionality enhancements.

Finally, there are many possible performance gains from being able to include the index in the file system. By updating the index quickly after files are written, we can ensure that the indexed file will already be in cache, while indexing at a later time will likely result in a cache miss. By delay indexing, we can coalesce multiple writes to the same file into a single indexing operation. By blocking searches while there is still data queued for indexing, we can ensure consistency.

The next section explores work related to our project, Sections 3 and 4 describe the design and implementation of our file system, Section 5 evaluates the performance of different versions of our file system, and the concluding sections offer ideas for future work and lessons to be drawn from our experience.

2 Related Work

The prior work related to our project generally falls within the scope of two distinct domains. On the one hand there has been a recent spate in applications that index file content for fast searches, but none of these have been implemented as file systems. On the other hand, there is a long history of "semantic" file systems, which associate with files more semantically-relevant metadata, i.e. descriptions like "source code", or "picture of a dog". While such file systems do not attempt to index and search the entire byte contents of files, they do in a sense maintain abstractions of those contents.

2.1 Applications that index and search file content

In the past six months several free software applications have popularized the indexing of files for fast content-based searching. Google Desktop, released in October of 2004, indexes the content of certain types of files in certain folders and provides a webpage front-end similar to its web search engine. Since then, MSN, Yahoo! and AOL have all released or announced similar new index-based file search tools. These offerings represent a growing of interest in a market that was previously left to smaller companies such as Copernic. Beagle [1] is an open-source project similar to Google desktop.

In addition to complete file system indexers, there has been a trend seen in several new applications do their own file-monitoring and indexing of file "metadata". For example, Apple's iTunes and Google's Picasa build and maintain databases of attributes associated with a user's music or pictures, respectively, and use these for faster browsing or indexing of (specific types of) files. The application-specific creation of indices, even if only for such attributes, offers motivation for a system-wide indexing service integrated into the file system or operating system, available to any application that needs it.

All of these tools differ from our project by being separate from the file system and operating system. This has ramifications in: when the indexing is done; which applications can use the index; and how incremental the indexing can be.

By integrating an indexer into the file system, we support immediate indexing of files when they are changed or created. This provide the benefits of a completely consistent index, as well as potential performance gains. By indexing the files while they are still in cache, we avoid additional disk accesses in the indexing process. In addition, by seeing exactly what data in a file has changed, and how, a file-system based indexer has more possibilities for incrementally indexing a file. And of course by providing a standardized file-system indexing API, we allow new applications to leverage the index for database-like UIs without the overhead of creating their own index.

2.2 Semantic file systems

Sheldon et al. introduced the concept of "semantic file systems" in [2]. Their basic idea was to replace the ubiquitous directory-based hierarchical approach to file system organization with something more database-like, using key-attribute pairs. This approach has been further developed in various projects, such as the HAC file system [3] and DBFS [4], and is rumored to be a feature of Microsoft's "Longhorn" release.[1] However, one of the difficulties in designing such semantic file systems is coming up with appropriate attributes. In [2], these were supplied by the user upon file creation—an approach with obvious deficiencies not only in

[1] see `http://msdn.microsoft.com/data/default.aspx?pull=/msdnmag/issues/04/01/WinFS/default.aspx`

terms of scalability, but because when a user creates a file they may not yet know what attributes they will want to use to locate it in the future. The problem of assigning attributes to files *automatically* has been studied in [8]. Finally, in [6], the authors explore the implications of a semantic file system that *does* have full content-based indexing. Interestingly, in exploring the design space they independently come up with many of the same ideas we had. However, their paper is only an ideas paper, with no implementation and very little theoretical work.

3 Design and Implementation

3.1 Basic architecture

The Searchable-by-Content File System (SCFS) was written with the FUSE toolkit [9] for user-level file systems in Linux, which is in turn based on the Virtual File System layer (VFS). VFS is sufficiently abstract to enable writing file systems without a physical presence on disk. For example procfs exports a directory /proc that is used to get and set information about the running system. It is also possible to write a virtual file system that simply serves as a mount point for a different location on the file system, translating requests relative to the mount point into requests relative to that location (cf. BSD's nullfs [7]). This is the way SCFS is structured: it wraps a layer of content-indexing around an already existent location on the file system. When files are created or changed through the SCFS mount point, the index is updated accordingly. SCFS also exports a virtual directory "/index" as an interface for performing keyword queries; the virtual index directory will be described in Section 4.

The index itself is maintained as a Berkeley DB, mapping keywords to lists of files containing those keywords. The basic procedure for indexing a file involves parsing it to extract keywords (alphanumeric strings) and then inserting the appropriate mappings into the DB. However there are a number of finer issues involving indexing: What should be indexed? When should indexing be performed? How is index consistency maintained after destructive updates of files?

3.2 Deciding what to index

An issue that naturally arises for content-indexing file systems is that indexing clearly must be selective. It would almost always be useless to index temporary files, for example, or to attempt to directly index the binary content of graphics or music files. The indexing scheme must also be able to distinguish between normal files and devices or directories (e.g. indexing /dev/zero would be a bit (actually infinitely many repetitions of a bit) counterproductive). Sometimes,

the file system organization can provide hints as to whether (and how) a file should be indexed—on Windows, for instance, file name extensions can be used to distinguish executables—which should not be indexed—from text files and also more structured text files (e.g. HTML or Word documents) that could be indexed given a suitable parser. In Unix it does not make much sense to use file name extensions, though. So instead, SCFS uses XFS extended attributes (cf. attr(5)) as flags for whether to index a file. Setting the "index" extended attribute to 1 for a file causes it (and all future versions of it) to be indexed, and setting the attribute for a directory causes all future files created in that directory to be indexed. Another possibility would have been to adopt Beagle's approach of letting the presence of a file ".noindex" indicate that files in a directory should *not* be indexed.

3.3 Deciding when to index

Whenever the file system is modified, such as after a write or a delete, the index must be updated to reflect the change. This update can either occur immediately after performing the corresponding system call (i.e. `write`, `unlink`, etc.) and before returning the system call's result, which we call *synchronous* behavior, or it could be queued for execution by a separate thread, possibly after a delay, which we call *asynchronous* behavior. While synchronous indexing has the virtue of always ensuring consistency of the index, it is ultimately untenable because of the mismatch between file sizes and the granularity of updates. The operation of copying a one megabyte file typically results in 256 writes of four kilobyte blocks, rather than a single one megabyte write. Synchronous indexing would therefore result in the file being reindexed 256 times, each time reading and parsing the entire file. This is a huge amount of wasted effort, since the index presumably will not be queried until the final write is completed.

On the other hand, it is easy to set up asynchronous indexing so that the queue is not flushed immediately (only every second, in our implementation). By then making the queue idempotent (i.e. enqueueing a file multiple times results in only one indexing operation), we create a "slack process" in the sense of Hauser et al. [5] that eliminates all of the unnecessary work performed by the synchronous indexing scheme. This asynchronous slack process is illustrated in Figure 1.

3.4 Destructive updates

Writes to a file that overwrite other data require that the old associations in the index somehow become invalidated. There are at least two unsatisfactory solutions to this problem: perform a traversal of the entire index looking for keyword entries containing the given file (horrible time complexity); or maintain an inverse mapping from files to keywords (horrible space complexity). The approach taken in SCFS is based on the idea that rather than physically remov-

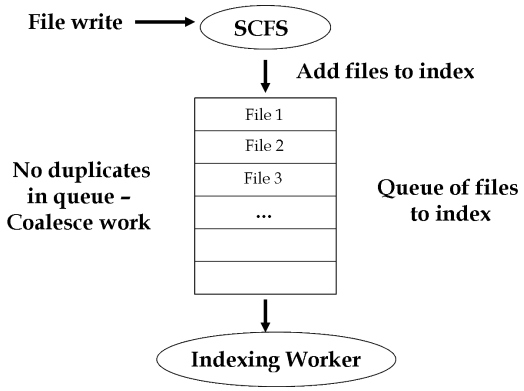


Figure 1: Asynchronous indexing with an idempotent queue.

ing old mappings from the database, it is possible to *logically* invalidate them. The index is in fact not simply a mapping from keywords to lists of files, but from keywords to lists of *versions* of files. A destructive update creates a new version of a file, and the file is parsed to find the keyword mappings to this new version. Nothing needs to be removed from the index database, and in our prototype implementation it actually grows monotonically; pruning the database of defunct mappings is a simple garbage collection problem, which we leave to future work.

3.5 Non-destructive updates

In the case of non-destructive updates, such as appends, since no keywords are being removed from the file it is not necessary to invalidate old mappings, and it is only necessary to add new mappings corresponding to the new keywords resulting from the append. This can almost be performed by just parsing the buffer sent into the appending write (i.e. without any disk reads) but there is a complication: a keyword can begin before the start of an append. To deal with this case, we must read as many bytes before the start of the append as the maximum keyword length. However, this is still a major improvement important for dealing with appends to large files, such as Berkeley mailboxes.

4 The Virtual Index Directory

We take advantage of being inside the file system to make searches possible by regular directory and file reads inside a special virtual directory called "index". The files viewable in this directory correspond to the keywords in the database. So, for example, after indexing one file named "mydoc.txt" with one line that reads:

```
"Hello," said the hippopotamus.
```

The index directory would look like:

```
/mnt/index$ ls
hippopotamus
hello
said
the
```

These files are virtual—i.e. they only exist as an abstraction exported by SCFS—but since they are files you can apply all sorts of cute Unix tricks to them, like tab completion. So, for example, by typing "`hi<tab>`" the word is completed in this case to hippopotamus. This is functionality we get for free from Unix.

To list the files that include a phrase, one can simply read the virtual file:

```
/mnt/index$ cat hippopotamus
/scfs/tmp/mydoc.txt
```

Though not yet implemented, multiple keyword searches could be provided with special operators. For example, the file "hello,hippopotamus" would contain the names of files containing both keywords.

This mechanism fits well into the small, pluggable architecture of Unix, and provides simple access for users, script writers and application developers. It could also be used as the basis for a more standard web-based or other GUI interface.

5 Evaluation

For small files, or small appends to large files, indexing is very fast and the overhead a user experiences when updating files is neglible. However, there are real-world scenarios when a large amount of data needs to be indexed in a short time—for example, untar'ing a large source code package.

Our tests do not try to build artificial files to index. We simply use popular packages that contain a lot of text files and examine how our indexing schemes perform when untar'ing them. Our test packages vary in size and this helps to evaluate the performance as the data size increases.

Our metrics for performance evaluation was the total time spent indexing, buffer cache performance, and the overhead of performing indexing with a cold cache. Timing is the most important factor as far as the users of the system are concerned. Buffer cache performance was monitored using the `sar` profiling tool.

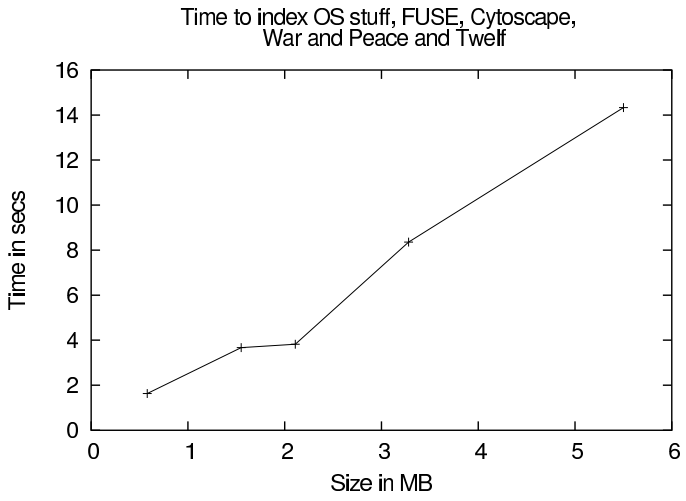


Figure 2: Plot showing running times for indexing OS_STUFF (0.58MB), FUSE (1.55MB), Cytoscape (2.11 MB), War and Peace (3.28 MB), and Twelf (5.5 MB)

The cold cache overhead gives a measure of the advantage of performing indexing dynamically with a short delay after file updates, since the cache is always warm. We timed SCFS with the three different variants of indexing (synchronous, asynchronous, and asynchronous indexing with an idempotent queue) which were described in section 3.3.

5.1 Results

As mentioned earlier, we tested our program by untar'ing publicly available packages. The packages that we tested include the source code for FUSE, Twelf and Cytoscape, the book 'War and Peace' and the well-researched readings and project files of the CMU operating systems course 15-712 (`OS_STUFF`). We will present the comparisons of the variants of SCFS for `OS_STUFF`, since it is considerably smaller than the others. We also present the running times for the above files using the asynchronous, coalescing variant.

The timing results shown below are just for the indexing functions. A clock is started when we begin the indexing routine and it is stopped when the routine completes. We also maintain a cumulative integer of the number of microseconds spent in indexing. The cache results shown below correspond to the number of physical disk accesses performed. This does not include the read on the zip file performed during the untar'ing operation since it is common for all the tests.

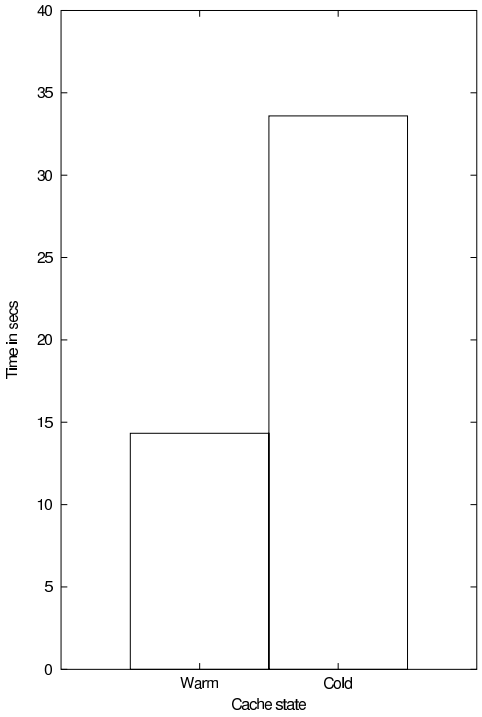


Figure 3: Plot showing running times for indexing Twelf source code with a warm and cold buffer cache.

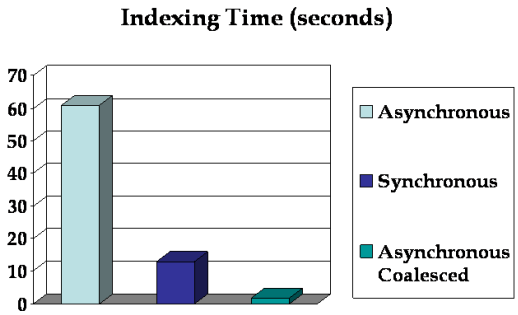


Figure 4: Plot showing running times for indexing `OS_STUFF`.

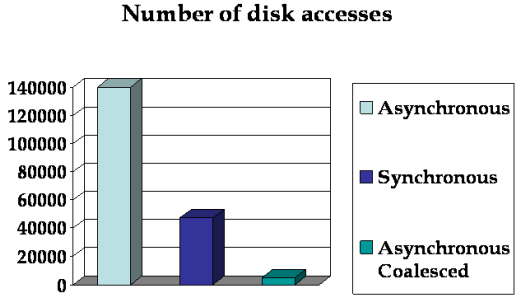


Figure 5: Plot showing the number of pages swapped out to disk while indexing.

In Figure 2, we compare the performance of the indexing scheme as the size of the packages increase. It is hard to extrapolate too much information from the plot since the packages contain different types of files. It is however clear that the indexing scheme is practical and the running time does not increase dramatically with the file sizes.

In Figure 3, we compare the performance of the indexing schemes when we have a warm cache versus a cold cache. As we verified with `sar`, untar'ing always results in a warm cache, since the writes swap the files and subsequent indexing just read them out of the buffer cache. Hence to obtain cold cache numbers, we untar'd the files with indexing turned off, manually polluted the cache, and then directly indexed the untar'd directory. The difference in the running times provides an estimate of the time spent in fetching blocks from disk.

In Figure 4, we have timing information for the three variants of SCFS. It is clear that coalescing work with an idempotent queue greatly helps in improving the performance of the indexing scheme. Most of the improvement is due to the fact that large files in the packages are written out in blocks which results in indexing the same file multiple times if coalescing is not used.

In Figure 5 we compare the number of pages swapped out for the different variants. This plot matches well with the timing results. Note these are pages swapped out, not swapped in—as mentioned above, every variant has perfect buffer cache performance when reading in files to index.

6 Future Work

One of the possibilities created by an indexing file system that we did not have time to explore was that of a *persistent* indexing file system. Logging file systems like the Elephant Filesystem have the ability to remember all versions

of a file. By combining this with automatic and immediate content indexing in the file system, a persistent indexing file system would allow one's entire file system history to be quickly searched. By indexing inside the file system, we can track every write that happens and ensure that every version of a file will be indexed. External indexing applications would not be able to guarantee this and might miss intermediate versions of a file that occurred between indexing passes. Our implementation of the index database was built to support persistence by keeping track of multiple version of each file and not removing old entries from the file system. However, we have not yet implemented the other code needed to have a persistent file system. Also, there are interesting user interface issues that need to be explored of how to display results of such searches. If a file included a keyword for some but not all of its lifetime, the search results display could identify each different version of the file as a different match, each consecutive period of time when the file did or didn't match the keyword, or employ some more graphical timeline to represent the history of that file or the entire file system. A persistent indexing file system would offer new functionality and open up interesting new user interface issues of how to best provide this functionality.

Automatic indexing of text is one step away from the idea of a file system as storing arbitrary bits and closer toward having semantic knowledge of its contents. With knowledge of other structured file formats such as HTML and PDF documents, we could support searching over more types of data and also provide smarter searches that weight matches differently (e.g. if a match occurs in the title).

7 Conclusions

Indexing the content of local file searching allows for very fast file searches. We built a file system that automatically indexes files' contents, providing several advantages. First, the file system is a central location for different applications to share the same index, removing the need for individual applications to create and maintain their own index. Second, by being in a file system we can offer access to the index by way of a novel virtual directory structure that takes advantage of existing Unix capabilities and lets applications or users easily make use of the index for searching. Third, by indexing in the file system we can index files shortly after they are written to insure the consistency of the index. Fourth, the greater flexibility to decide when and how to index allows us to make informed performance tradeoffs that let us balance cache performance, file-update coalescing and system responsiveness.

Our evaluation showed that some amount of file-update coalescing was necessary to avoid frequent indexing caused by file writes being automatically broken up into many individual 4KB writes. We think that content indexing will continue to be a valuable service, have demonstrated that placing it inside the file system offers several advantages, and hope that our experience in implementing and

testing it will be valuable to those working on content indexing and file systems in the future.

Acknowledgments. We would like to thank Garth Gibson and Hyang-Ah Kim for helpful discussions.

The source code for SCFS is available at:

`http://www.cs.cmu.edu/~noam/scfs/`

References

[1] Beagle, 2005. http://www.gnome.org/projects/beagle/.

[2] Mark A. Sheldon David K. Gifford, Pierre Jouvelot and James W. O'Toole Jr. Semantic file systems. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, October 1991.

[3] Burra Gopal. Integrating content-based access mechanisms with hierarchical file systems. *Proceedings of the Third Symposium on Operating Systems Design*, 1999.

[4] Onne Gorter. Database file system: An alternative to hierarchy based file systems. Technical report, University of Twente, August 2004.

[5] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: a case study. pages 94–105, December 1993.

[6] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. In *The 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, 2003.

[7] J. S. Pendry and M. K. McKusick. Union mounts in 4.4bsd-lite. In *USENIX Conf. Proc.*, pages 25–33, January 1995.

[8] Craig A. N. Soules and Gregory R. Ganger. Why can't I find my files? new methods for automating attribute assignment. In *The 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[9] Miklos Szeredi. Filesystems in userspace (fuse), 2005. http://fuse.sourceforge.net/.

Comparison-Based Server Verification for Stateful and Semi-Deterministic Protocols

Gregg Economou, Raja R. Sambasivan, Terrence Wong
15-712: Advanced Operating Systems & Distributed Systems
Carnegie Mellon University

Comparison-based server verification has been shown to be an useful tool for debugging servers. However, current implementations only allow for debugging of servers that use stateless and deterministic protocols. Given the current push towards complex stateful and semi-deterministic protocols, it is interesting to see if we can realize the benefits of comparison-based verification for these more complicated protocols. In this paper, we detail the implementation of a comparison-based verification tool known as the Tee for NFS Version 4 (NFSv4), which is both stateful and semi-deterministic.

1 Introduction

Debugging servers is tough [12]. Although the client-server interface is usually documented in a specification, there are often vague or unspecified aspects. Isolating specification interpretation flaws in request processing and in responses can be a painful activity. Worse, a server that works with one type of client may not work with another, and testing with all possible clients is not easy.

The most common testing practices are RPC-level test suites and benchmarking with one or more clients. With enough effort, one can construct a suite of tests that exercises each RPC in a variety of cases and verify that each response conforms to what the specification dictates. This is a very useful approach, but it is time-consuming to develop and difficult to perfect in the face of specification vagueness. Popular benchmark programs such as SPEC SFS [10] for NFS servers, are often used to stress test servers and verify that they work for the clients used in the benchmark runs.

Another interesting method of testing servers is *Comparison-Based Server Verification* as introduced in [12] and shown in Figure 1. In this method, clients send requests to a *Tee* that relays the requests to both a *Reference Server* and *System-Under-Test (SUT)*. Responses from both servers are captured by the Tee and differences flagged. If the reference server is chosen based on the belief that it works correctly, it can be used as a "gold standard" against which the SUT's correctness can be evaluated.

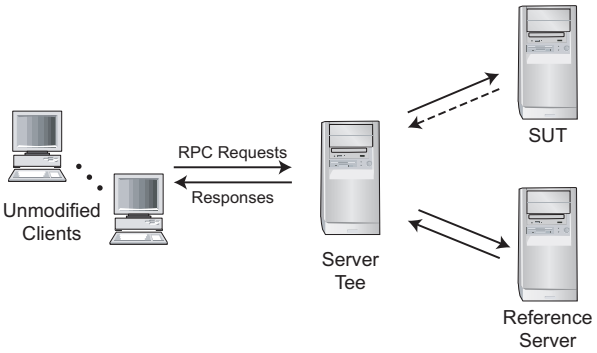


Figure 1: *The server Tee is interposed between unmodified clients and the unmodified reference server, relaying requests and responses between them. The Tee also sends the same requests to the system-under-test and compares the responses to those from the reference server.*

Comparison-based server verification yields two distinct advantages to regular testing. First, it ensures *bug-compatibility*. Second, it allows for live testing. These advantages are described in further detail below.

Bug Compatibility: A given server implementation may become so widely used that it introduces a de facto standard. Any new servers wishing to compete with this popular implementation must emulate all of its nuances regardless of whether they are discussed in the original protocol specification. Comparison-based server verification allows verification of such "bug compatibility" by comparing responses from both the popular server and new server.

Live Testing: Benchmarks such as SPEC SFS work by stress testing servers with high intensity static workloads over a short period of time. If the server does not crash and returns results correctly over this period, it is deemed to be correct. This testing approach has two critical flaws: (1) The testing period is over a period of hours or days at best. Hence, these benchmarks will not reveal bugs related to system degradation that occurs over very longer periods. (2) The workload is static and hence may not "tickle" a bug which will be exposed by some real workload that the SUT will see when de-

ployed. Comparison-based server verification addresses these shortcomings by allowing a new server to be added quickly in an existing live environment as a SUT without risk of failure affecting the client. Client data is safely stored on the proven reference server and clients are oblivious to the existence of the SUT. Both clients and the reference server continue working normally while SUT responses to the live workload requests are captured and compared with reference server responses.

Though comparison-based server verification does provide clear advantages to regular testing, current implementations are limited to testing servers that implement stateless and deterministic protocols such as NFSv3 [4]. However, new emerging client/server protocols are both stateful and semi-deterministic[1]. Statefulness allows the server to make better use of client side caching and locks. Since the server keeps track of which clients are accessing various files, it can delegate complete control of unshared files to the clients that are exclusively accessing them. Lock management is also made more robust by requiring statefulness. Semi-determinism allows various server implementations the ability to choose when they should issue such delegations and locks. This allows various servers to optimize for the workload they expect to see.

Given the current push towards more complex stateful and semi-deterministic protocols, it is compelling to see if it is possible to realize the benefits of comparison-based server verification for these more complicated protocols. In attempting to do so, we find that there are two major issues that must be addressed. The first is a problem of state-keeping at the Tee. The second is a problem of performing comparison verifications for a semi-deterministic protocol in which two servers are allowed to respond differently to the same request.

The problem of state-keeping arises because a Tee for a stateful protocol is simply not privy to certain state information that it needs about clients and servers in order to perform its function. An example of such state is the file currently being accessed by each client. A Tee for a stateless protocol would see such information over the network because clients would have to retransmit such state with every request. A Tee for a stateful protocol does not see this information because it is hidden within the server and never broadcast over the network. Hence, a Tee for a stateful protocol must somehow scan the client request stream and via some heuristics, opportunistically recreate the necessary state.

The problem of semi-determinism arises because with a stateful protocol, it is not valid to assume that replies from two servers for the same request will be identical. A semi-deterministic protocol allows two servers that see the same request stream to end up in different valid states and generate different valid replies. Hence, the question that is posed here is whether comparison-based verification is possible for a semi-deterministic protocol and if so, what accommodations must be made.

In this paper, we describe the creation of a comparison-based verification tool called the NFSv4 Tee for the NFS version 4 protocol [11] [8] which is a semi-deterministic and stateful. The main contribution of this paper is a description and analysis of how we handle the problem of state-keeping at the Tee. In the future works section, we describe how we accommodate non-determinism within the context of comparison based server verification.

The rest of this paper is organized as follows. Section 3 describes the components of a generic Tee along with detailed information about the NFSv3 protocol, previous work in creating a Tee for NFSv3, and the NFSv4 protocol. Section 2 describes other work similar to comparison based server verification. Section 4 describes how we handle the problem of keeping state at the NFSv4 Tee. Section 5 describes the basic architecture of the NFSv4 Tee. Section 6 evaluates our architecture and the data structures needed to keep state. Section 7 describes work that is left to be done. This includes handling the semi-deterministic features of NFSv4 such as locks, delegations, and callbacks. Finally we conclude with Section 8.

2 Related Work

We believe that the concept of comparison-based verification as a means by which to debug servers is new. However, there has been a large amount of previous research done in the field of debugging techniques. Additionally, the Tee itself is an application of proxy-clustering which is a technique that has been used for many other applications.

Ballista [7] is a tool developed at Carnegie Mellon University to evaluate the robustness of a system. Ballista aims to comprehensively test SUTs by generating an exhaustive sequence of inputs and monitoring the resulting output. In this sense, Ballista is much more comprehensive than a Tee that performs comparison-based verification. Where the Tee only aims to guarantee that the SUT is as robust as the reference server, Ballista aims to guarantee that the SUT is completely devoid of any failures. Ballista's drawback lies in the fact that it is hard to create a set of completely comprehensive inputs. A Tee avoids this problem by assuming that the live workload on which testing is performed will sufficiently encom-

[1] A semi-deterministic server is a server that is free to choose between one of several valid replies and states given some input

pass most of the comprehensive input set.

Version Manager [2] is a tool that facilitates software upgrades by employing a cluster proxy to redirect incoming requests to both the old version and the new version of the software. In this way, availability and compatibility are both preserved during and after the upgrade process.

Cuckoo [6], Slice [3], Z-Force NAS Aggregation switches [13], and Rainfinity Rainstorage devices [9] all use the concept of proxy-clustering in order to perform their function. Cuckoo uses proxy-clustering to affect a form of load balancing among monolithic NFS servers. Slice uses proxy clustering to reroute client requests based on request type. Proxy clustering is used in slice to reroute metadata operations, small I/Os, and large I/Os to different servers that are optimized for each workload. Z-Force and NAS aggregation switches and Rainfinity Rainstorage devices use proxy clustering to virtualize independent NFS servers.

3 Background

3.1 Major Components of a Tee

Any implementation of comparison-based server verification will have four common components: relay, duplication, comparison, and synchronization.

Relay: The relay is the component that handles communication between the client and the reference server. When it receives a client request, it forwards it to the reference server, waits for the reply, and sends the reply back to the client.

Depending on the RPC protocol, the relay may have to modify some fields. Because it is effectively funneling multiple client requests into a single relay-to-reference server connection, the relay must be able to uniquely identify each response so that it can be sent to the correct client.

Duplication: The duplication module makes a copy of each original client request, modifies it as necessary, and forwards it to the SUT. The duplication module may have to modify any requests with fields that are specific to the reference server so that they have the proper effect when set to the SUT.

Comparison: The comparison module matches the reference server's reply to the associated SUT reply. Then, using its knowledge of the protocol, the comparison module parses the responses and compares the fields. For some fields such as the file data, the fields are bitwise compared when looking for discrepancies. For some fields like timestamps, however, special rules may be employed to ensure that they are similar but not necessarily identical. These fields are called loosely comparable fields. Finally, some fields contain data that is specific to the server and are simply not comparable.

Synchronization: In order for a comparison on requests to an object to be meaningful, the object in question must be in the same state on both the reference server and the SUT. The synchronization of objects is beyond the scope of this paper.

3.2 NFS Version 3 Protocol

The Network File System version 3 protocol implements a simple network filesystem. Clients mount the remote filesystem and use it as if it were local by issuing a command at a time through RPCs. NFSv3 supports UNIX like filesystem semantics by offering an interface with commands such as open, read, write, delete, and readdir.

NFSv3 is a stateless protocol, meaning the servers retain no knowledge of their clients. As a result, server implementations can be very simple; they merely translate the request RPC to a request on the local filesystem. The client, then, must provide with each request the complete set of information required to allow the server to perform the operation. This information may include a unique and persistent file identifier called a filehandle, or a parent directory path followed by a file name.

In addition to being stateless, the NFSv3 protocol is highly deterministic; only when a client has sent a request does it expect a single, consistent, and reasonably quick reply.

3.3 NFS Version 3 Tee

The NFSv3 Tee is designed to do debugging while in a live environment. To facilitate this, the code is divided into two major components called the relay and the plugin (Figure 2). The relay component is the same as the one described above. It waits for the client request and forwards it to the reference server. It then waits for the reply and sends it back to the client. The plugin, which communicates with the relay via shared memory, contains the rest of the Tee components: duplication, comparison, and synchronization.

The separation of the relay from the rest of the system allows for live testing. Any failures on the part of the SUT or the plugin are isolated and therefore do not affect the client's ability to communicate with the reference server. The separation also allows the plugin and SUT to be added, removed, or restarted at any time.

The stateless nature of the NFSv3 protocol influences the design of the plugin. In a stateless protocol, the server remembers nothing about the clients and therefore the

client must provide all necessary state with each request. In NFSv3, an operation on a file requires the client to include a unique and persistent file identifier called a filehandle. Since the Tee sees all client requests, any metadata can easily be stored by keying it on this filehandle.

Because the NFSv3 protocol is stateless and deterministic, the implementation of comparisons for the Tee is fairly straightforward.

3.4 NFS Version 4 Protocol

The Network File System version 4 protocol aims to improve NFSv3 performance over high latency low bandwidth networks such as the Internet. A new operation, called a compound request, improves Internet performance by reducing the number of low level RPCs required to complete a high level task. Several NFS requests are combined into a single RPC packet. A consequence of this is that the server must now store per-client state. Consider, for example, a compound request that contains a file open and a file write. The file open, in NFSv3, returns a filehandle which allows the client to identify the file when it issues the write command. However, because the write is packaged into the compound along with the open, the client is unable to include the filehandle because it does not know it yet. This is just one reason the server stores the notion of the *current filehandle*[2].

The current filehandle automatically changes based on the request stream so that the server correctly operates on the file that the client expects.

In addition to compound requests, NFSv4 includes some semi-deterministic features that the servers may implement. Because of these features, an NFSv4 Tee implementation will be more complicated than a NFSv3 Tee implementation and comparisons will not be as simple.

Byte Level Locking: This mutual exclusion feature is now built into the protocol whereas it was an external in NFSv3.

Delegations: A server issues a delegation to a client to signify that the client has exclusive access to the file. The client may cache the file locally and use it without checking for updates at the server. When a different client wishes to access the object, the server sends a callback to the client with the delegation to revoke it and force the data to be written back to the server. To accomplish this, the server must store more state; the location of the client holding each delegation.

Volatile Filehandles: In NFSv3, filehandles were unique and persistent file identifiers. In NFSv4, the designers did not want to burden servers with the need to track persistent and unique identifiers in the event that the backing store did not easily support such a thing. As a result, NFSv4 has different types of filehandles: the familiar persistent filehandles and also filehandles that expire. For filehandles that expire, the client must be prepared to deal with the fact that the use of a filehandle may return an error message indicating that the filehandle is stale.

[2]The current filehandle is the filehandle that the client is currently accessing. The server stores this state for each client so that the client does not have to include it in successive requests for the same object.

4 Storing State

The NFSv4 Tee must store state for each client and file on both the reference server and SUT. This is because NFS server replies to certain requests contain opaque field values that must be saved by clients and returned to the server on subsequent calls. The opaque field values returned by the reference server are stored by the clients and hence do not need to be stored by the Tee. However, because the Tee does not forward SUT replies to clients, the Tee must save these opaque field values itself and insert them into new SUT calls as appropriate. Table 1 shows all of the opaque fields that can be returned by NFSv4 servers.

Figure 3 demonstrates the need for keeping state at the Tee. In step one, a `OPEN` call is issued by a client for file A_{ref}, the version of file A stored on the reference server. This call is captured by the Tee and immediately forwarded to the reference server. In step two, the Tee duplicates the `OPEN` call and sends this duplicated call to the SUT. The `OPEN` call sent to the SUT instructs the SUT to open file A_{sut}, the version of file A stored on the SUT. In step three, the reference server responds with its reply to the `OPEN`. The reply contains an opaque $stateid4_{ref}$ field value. In step four, the SUT responds with its reply to the `OPEN`. This reply contains a $stateid4_{sut}$. field value that need not be the same as $stateid4_{ref}$. Subsequent calls for file A_{ref} from must contain $stateid4_{ref}$, whereas the duplicated version of these calls sent to the SUT must contain $stateid4_{sut}$. The Tee must hence store $stateid4_{sut}$ and replace $stateid4_{ref}$ with $stateid4_{ref}$ in calls it duplicates to the SUT for file A.

The scenario shown above shows that the Tee must store a mapping between files on the reference server and the corresponding opaque field values that must be inserted in calls that reference the same file on the SUT. We only need to save the opaque field values returned by the SUT as the clients handle the opaque field values sent by the reference server.

Notice that the scenario presented above is a simplification of the state keeping problem. The opaque field values are unique to files *and* clients. Hence, the Tee must

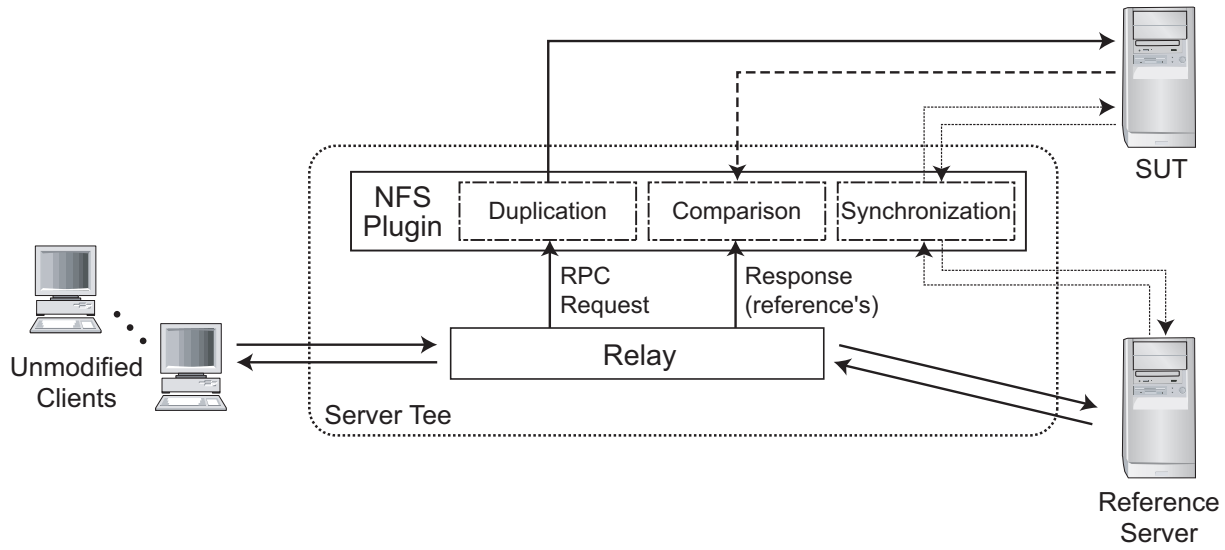


Figure 2: **Software architecture of an NFS Tee.** To minimize potential impact on clients, we separate the relaying functionality from the other three primary Tee functions (which contain the vast majority of the code).

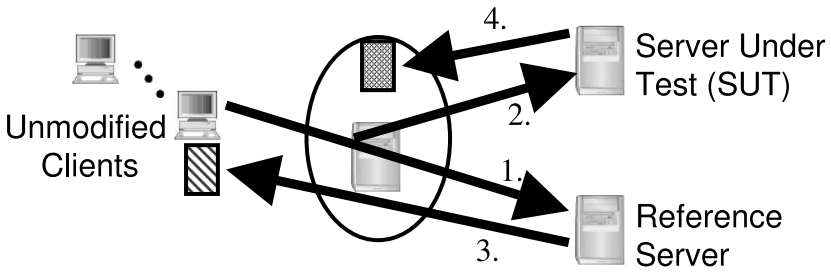


Figure 3: *A case for storing state at the Tee.*

store the opaque field values for each file on the reference server and for each client that accesses these fields.

Section 4.1 describes how we keep track of state in both the NFSv3 and NFSv4 Tee along with why keeping track of state in the NFSv4 Tee is so much harder than doing the same in the NFSv3 Tee. Section 4.2 details describes the architecture of the namespace tree, our statekeeping solution for the NFSv4 Tee.

4.1 Tracking state in NFSv3 and NFSv4

In stateless protocols like NFSv3, state keeping in the Tee is trivial. Each operation on a file contains a unique and persistent identifier for the file call the filehandle. The NFSv3 tee simply keeps a mapping of reference server filehandle to SUT opaque field values.

State keeping in NFSv4 is not so easy. No persistent or unique identifier of a file is ever sent in NFSv4 calls and replies. Instead, NFSv4 servers keep track of the current file each client is accessing in a *current_fh*$_{client}$ variable. The NFS operations shown in Table 2 implicitly change *current_fh*$_{client}$ on the server. All other NFS operations are assumed to operate on the filehandle stored in *current_fh*$_{client}$.

Our solution to the problem of keeping track of state in the NFSv4 Tee involves mirroring the reference server namespace in a tree (we call this tree the *Namespace Tree*. This namespace tree contains nodes that correspond to directories and files on the reference server. Each node contains the opaque field values for the corresponding file on the SUT. This solution presents many problems that are not present when using filehandles for the mapping. Most importantly is the fact that we must "reconstruct" a copy of the reference server namespace solely from viewing NFSv4 calls and replies.

Mapping Data	
Types	*Purpose*
stateid4	Used to share state between clients & servers
nfs_fh4	Uniquely identifies a file at a server
verifier4	Identifies lock state held by client for a file
nfs_cookie4	Identifies the last read directory entry

Table 1: **Set of opaque field types that are unique to NFS Servers** All of these types correspond to values that are returned by a NFS server to clients as opqaque. Servers expect these opaque values to be passed back to the server during certain NFS requests.

NFSv4 Requests that may implicitly modify the current filehandle			
CREATE	LOOKUP	LOOKUPP	OPEN
OPENATTR	PUTROOTFH	RENAME	RESTOREFH

Table 2: **NFSv4 Requests that may change the current filehandle on a server without any notification.**

4.2 Namespace Tree Implementation

Our implementation of the namespace tree is based around a copy of the reference server's namespace. Each node in the namespace contains the required mapping information to convert a reference server call to a SUT call. A second tree mirrors the SUT namespace and each node is a pointer to the corresponding object in the reference server namespace tree. The need for the two trees arises from the fact that requests to the two servers are sent asynchronously; the namespaces for the two trees may diverge for a short period of time.

Calls which refer to files by name, such as LOOKUPP or OPEN, provide information about the file namespace on a server and are used to construct this tree. Another set of calls, GETFH and SETFH, refer to objects by filehandle rather than file name. A hash table is used to translate a filehandle to the appropriate file name in the namespace tree.

5 NFSv4 Tee Architecture

The NFSv4 Tee is broken up into two major components: the Relay and the Plugin. The relay is solely responsible for the communication path between the reference server and clients and is capable of functioning even in the face of plugin and SUT failure. The Plugin handles communications with the Plugin and contains the duplication, and comparison modules.

In the following sections, we describe the architecture of both the relay and plugin in more detail. We discuss the relay in detail in Section 5.1 and the plugin in Section 5.2.

5.1 NFSv4 Tee Relay Architecture

The relay architecture for the NFSv4 Tee relay is simple and is mostly unchanged from the NFSv3 Tee relay. Requests from clients are captured, made available over shared memory to the plugin, and forwarded to the reference server. Correspondingly, replies from the reference server are also captures, mode available over shared memory to the plugin, and forwarded back to the appropriate client. A more detailed description of the relay can be found in [12].

5.2 NFSv4 Tee Plugin Architecture

Figure 4 shows the basic architecture of the NFSv4 plugin. Individual boxes in the diagram correspond to various plugin modules. To maximize performance, modules are asynchronous with regards to each other. Communications between modules is performed via IPC and concurrency is handled by the Netscape State Threads Package [1]. Below, we list a description of each module listed in the figure. Please note that since we have not yet implemented support for the semi-deterministic features of NFSv4 (such as callbacks, delegations, and locks) that Figure 4 and module descriptions listed below do not take into account these features. The architecture diagram presented also does not currently show the modules needed for synchronization.

SUT Forwarder: The SUT forwarder serves as the duplication module for the NFSv4 Plugin. It receives client requests over shared memory from the relay. For each request it receives, it looks up the current reference server file that the client which sent the request is currently operating on. This is the file the request currently being processed by the SUT forwarder is operating on. The SUT forwarder looks up the current reference server file

in the reference server namespace tree to retrieve the associated server specific fields for the corresponding file on the SUT. The SUT forwarder modifies the request by inserting in the server specific fields for the SUT and sends the modified request to the *RPC Send* module and *SUT Reply Matcher* module. It also sends the original unmodified calls that it receives from shared memory to the *Plugin Reference Input Matcher*.

RPC Send Module: This module simply sends the RPCs it receives to the SUT.

Plugin Reference Input Matcher: The sole function of this module is synchronization. Though we do not need the reply for a request sent to the reference server before relaying that request to the SUT, we do need to wait for the reference server reply before modifying the reference server namespace tree. Hence, this module simply waits for reference server calls sent to it by the *SUT Forwarder* and reference server replies sent to it over shared memory by the relay. When it receives a matching call and reply, it concatenates both into the same data structure and passes this along to the *Reference Server Namespace modifier*.

Reference Server Namespace Modifier: The namespace modifier receives matched calls and replies from the *Plugin Reference Input Matcher*. If the result of a `RENAME`, or a `DELETE` is success, then it modifies the reference server namespace tree accordingly. A `RENAME` corresponds to moving a subtree of the reference server namespace tree from one node to another note. A delete corresponds to removing the corresponding node for that file from the tree. Because it is possible for us not to have heard a reply from the SUT for the same request yet, that modifications to the reference server namespace tree might make this tree inconsistent with the namespace of the SUT. This is the primary reason why we keep both a reference server namespace tree and a SUT namespace tree. After potential modification of the reference server namespace tree, the *Reference Server Namespace Modifier* passes the matched reference server calls and replies to the *SUT Reply Matcher*.

Receive RPCs Module: This module simply receives RPCs from the SUT and forwards these replies to the *SUT Reply Matcher*.

SUT Reply Matcher: The function of the SUT reply matcher is similar to that of the *Plugin Reference Input Matcher*. This module receives matched reference server calls and replies from the *Namespace Modifier* and SUT calls from the *SUT Forwarder*. It waits for SUT replies from the *Receive RPCs* module and matches SUT calls and replies with the corresponding matched reference server calls replies. It also saves the server specific data sent to it by the SUT by inserting into the SUT namespace tree using the SUT current file variable. Finally, it sends this newly matched data to both the *Comparator* and *SUT Namespace Modifier*.

SUT Namespace Modifier: The SUT namespace modifier performs the same function as the *Reference Server Namespace Modifier* except for the SUT namespace tree.

Comparator: This module compares corresponding replies from the SUT and reference server and flags differences.

6 Preliminary Evaluation

6.1 Performance impact

We use PostMark to measure the impact that theTee has on a client. Because the relay runs as a seperate process from the plugin, it is the only part of the Tee that affects the client. It is sufficient therefore to run PostMark on the Tee without the plugin portion activated (We assume that we are using a multiprocessor system in which the operation of the plugin has a negligable impact on the performance of the relay).

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [5]. It creates a large number of small randomly-sized files (between 512 B and 9.77 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append.

The experiments are done by running a variable number of clients on a single machine mounting an NFSv4 server over a gigabit ethernet link. We compare the throughput of the clients when the clients directly mount the NFS server and when the clients mount the NFS server through the Tee. The machines used in this experiment are all Pentium 4 class machines at 2.66ghz with 512MB of RAM.

Figure 5 shows that using the Tee reduces client throughput when compared to a direct NFS mount. Performance hardly scales with increased client concurrency and the through-tee performance stays around 75-80 percent of the direct mount case.

The NFSv4 Tee's performance hit of 20-25 percent when compared to the direct NFS mount is an improvement over the NFSv3 Tee's performance hit of up to 60 percent. Curiously however, the NFSv4 server's overall performance is around 15-30 percent of the NFSv3 server's. This reduction in performance may be a result of the infancy of the NFSv4 Linux code and may explain the reason that PostMark performance does not scale with the

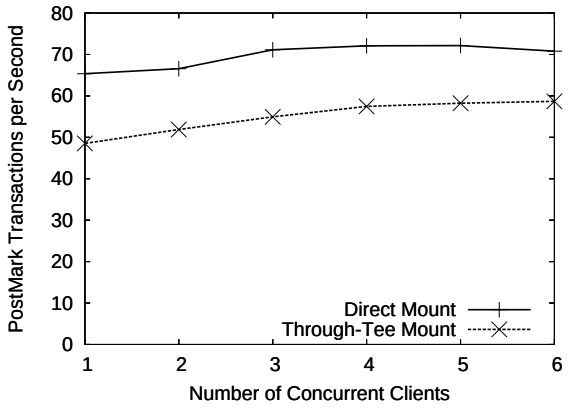


Figure 5: Performance with and without the Tee.

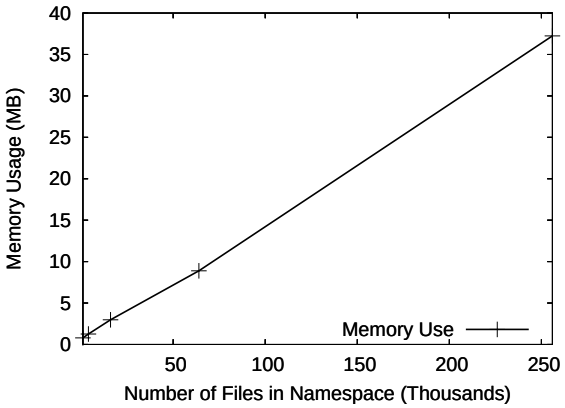


Figure 6: Memory use of the namespace tree.

number of clients.

6.2 Namespace tree performance

We ran workloads of various sizes to determine the performance of the namespace tree. Five different live filesystems were used including a small /var filesystem of a typical bsd system, a typical / filesystem, a /usr filesystem, a large source tree, and a whole workstation including diverse installations and lots of deep directories and several hundred thousand files. These experiments were done on a Pentium-3 1ghz machine with 512 MB of RAM. File lookups or opens were done on all files in the filesystems in order to populate the namespace tree, and getfh operations were subsequently done to lookup entries in the namespace tree as well as exercise the filehandle hash tables. Figure 7 shows that the time it takes to lookup every object in the system scales reasonably as well. The plateau is an artifact of a particular workload; real filesystems were used and each had its own structure. Particularly unbalanced directory trees would increase the average lookup time for paths going through them. This behavior is comparable to a real filesystem which has the same issue as it stores a tree structure imposed on it externally.

For workloads of reasonable size, the namespace table is more than capable of storing the mapping data that is needed for each file. With 250k files in the system, the table required 37 megabytes of memory and supported lookup operations at a rate of over 17,500 per second.

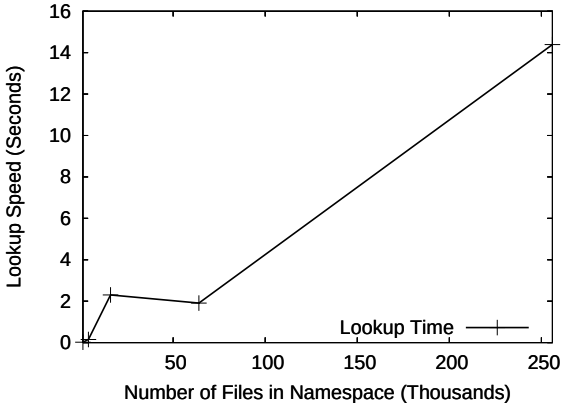


Figure 7: Performance of the namespace tree.

7 Future Work

The complexity of the NFSv4 protocol when compared to NFSv3 prevented us from implementing much more than the low level infrastructure and the namespace tree. Our NFSv4 Tee design, however, was not done without consideration for some of the other NFSv4 changes.

7.1 Delegations and Callbacks

In NFSv3, the reference server and the SUT, from the Tee's perspective, were mostly deterministic. That is, when the Tee sent a client request to both servers, it could reasonably expect similar replies. In NFSv4 however, the server is given much more freedom to choose options. It is up to the server, for example, to determine the circumstances under which to issue a delegation. Differing implementations at the reference server and the SUT could easily cause problems at the Tee's comparison module.

Consider what happens if one server issues a delegation for a file but the other server does not. There are two combinations in which this can happen.

Reference delegation, no SUT delegation: If the reference server alone issues a delegation to a client, it will take advantage of it and use the file locally. Neither the

reference server nor the SUT will see writes, which does not cause any problems. When the delegation ends, the reference server alone sends a callback to the client and the states of the two servers again converge.

SUT delegation, no reference delegation: If the SUT alone issues a delegation, the client will never receive it since the SUT response ends in the Tee. As a result, the client may continue to issue writes to the network which the duplication module will forward to the SUT, possibly causing it to become confused. When the delegation ends, the SUT will issue a callback which the client again will not receive. The server states again converge.

It would be trivial to require that the two servers being compared implement similar policies regarding delegations, but that limits the number of server combinations that are testworthy. We would like to compare any two servers without forcing them to be implemented in any particular way. Fortunately, detection of delegations should be straightforward: the client issues a request after one server has returned a delegation but before the other server has issued a delegation. Unfortunately, once the Tee detects the problem, it can do nothing more. Comparing requests on this object while there is only one delegation outstanding may give us may false positives. Although the Tee can still perform comparisons on other objects, this object does not provide any useful comparisons until their delegation states converge again.

7.2 Locks

Because the Tee merely acts as an intermediary, it simply sends client lock requests to both servers. If an unauthorized client attempts to access the locked data, both servers should return the same errors. Comparisons for the locking of and for locked objects should be straightforward.

7.3 Volatile Filehandles

Volatile filehandles introduce a difficulty similar to that caused by delegations and callbacks. One server may generate a persistent filehandle while the other server does not. A future client request may result in one server returning a stale filehandle error while the other server executes the request. When we see that one server has returned a stale filehandle error when the other has not, we must update the Tee's namespace tree (this is easy because we have separate trees representing each of the two servers). Depending on which server returned the error, which server executed the request, and the idempotency of the request, the Tee may have to resynchronize the object state before comparisons are allowed to continue.

8 Conclusion

From [12] it is clear that comparison-based server verification offers many advantages to current server debugging techniques when debugging stateless and deterministic servers. Some of these advantages include the ability for live-testing and bug-compatibility.

In this paper, we have shown that complex stateful and semi-deterministic protocols can also enjoy the benefits offered by comparison-based verification. We have also identified the major problems associated with creating a comparison-based verification tool for these complex protocols.

References

[1] State Threads for Internet Applications, 2003. http://state-threads.sourceforge.net/docs/st.html.

[2] M. Agrawal, S. Nath, and S. Seshan. Upgrading Distributed Applications Using the Version Manager, March 2005.

[3] D. Anderson and J. Chase. Failure-atomic file access in an interposed network storage system. *IEEE International Symposium on High-Performance Distributed Computing* (Pittsburgh, PA, 1–4 August 2000), pages 157–164. IEEE Computer Society, 2000.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. RFC 1813 - NFS Version 3 Protocol Specification. RFC 1813, June 1995.

[5] J. Katcher. *PostMark: a new file system benchmark.* Technical report TR3022. Network Appliance, October 1997.

[6] A. J. Klosterman and G. Ganger. *Cuckoo: layered clustering for NFS.* Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[7] P. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. *International Symposium on Fault-Tolerant Compter Systems* (Madison, WI, 15–18 June 1999), 1999.

[8] B. Pawlowski, S. Shepler, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS Version 4 Protocol. Appears in the proceedings of the 2nd Intl. System Administration and Networking (SANE) Conference, May 2000.

[9] Rainfinity Inc. www.rainfinity.com.

[10] SPEC SFS97_R1 V3.0 benchmark, Standard Performance Evaluation Corporation, August 2004. http://specbench.org/sfs97r1/.

[11] S.Shelpler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC 3530 - Network File System (NFS) version 4 Protocol. RFC 3530, April 2003.

[12] Y.-L. Tan, T. Wong, J. D. Strunk, and G. R. Ganger. Comparison-based File Server Verification. Appears in proceedings of the 2005 Usenix Technical Conference, General Track, April 2005.

[13] Z-Force Inc. www.zforce.com.

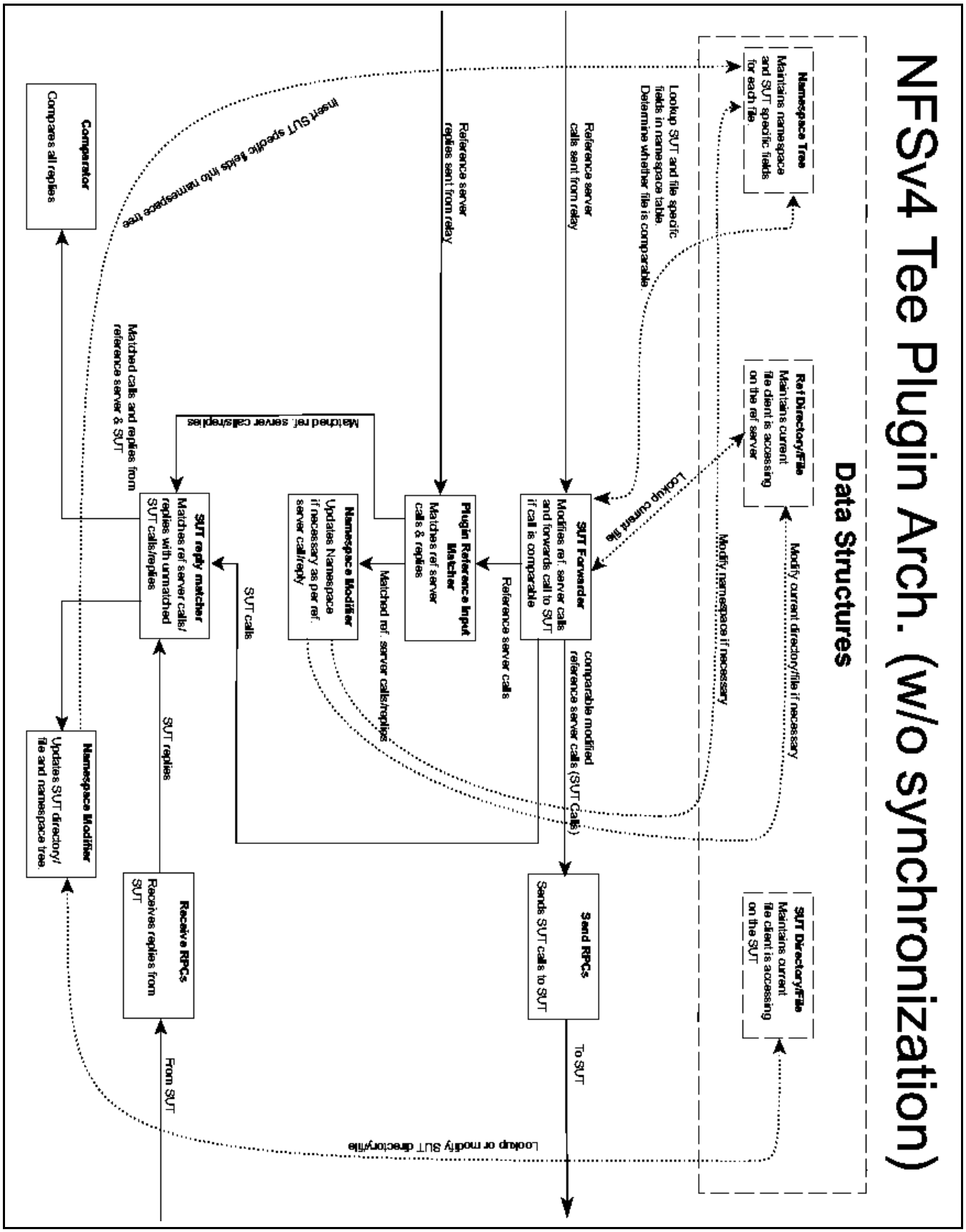


Figure 4: **The NFSv4 Tee Plugin Architecture** This diagram shows all of the components of the NFSv4 Tee Plugin

Recovering from Intrusions in the OSPF Data-plane

Elaine Shi Yong Lu Matt Reid

Abstract

In this paper, we propose CONS-ROUTE, a data-plane intrusion recovery mechanism for securing the OSPF routing protocol. CONS-ROUTE allows routers to perform intrusion detection in a distributed manner. The intrusion detection outcome can be used globally to reevaluate routing decisions in a way that is resilient to the slandering attack, where a malicious router claims that a legitimate router is misbehaving. We evaluate CONS-ROUTE through simulation and compare it with several simple OSPF data plane resilience techniques.

1 Introduction

1.1 The OSPF Routing Protocol

OSPF is one of the predominant link state routing protocols in today's Internet. The invention of link state routing protocols was partly to overcome the limitations of distance vector routing protocols. A link state routing protocol works roughly as follows: Upon initialization or network topology changes, a router generates a Link-State Advertisement (LSA) to reflect the new link states on that router. The link-state advertisement will be broadcast to the entire network of routers through flooding. As a result, each router in the network possesses complete knowledge of the entire network's topology. Then each router uses a shortest path algorithm, e.g., Dijkstra, to calculate a Shortest Path Tree to all destinations. The destinations, the associated cost and the next hop to reach those destinations will form the IP routing table.

Unfortunately, OSPF was designed for a trusted environment, and there is no guarantee as to its robustness in the presence of compromised routers. Recently, researchers have systematically studied the potential vulnerabilities of the OSPF routing protocol [14, 7]. In addition, various defense mechanisms have been proposed. In Section 4, we shall survey related work in the space.

We consider any routing protocol to consist of a *control plane* and a *data plane*. **Control plane** The control plane is where routing information is exchanged in order for the routers to set up the topology information needed to calculate routes. In the case of OSPF, the control plane is where the LSAs are broadcast.

Data plane By contrast, the data plane refers to the actual packet forwarding. When a router receives a packet for forwarding, it indexes into its routing table based on the packet's destination address and looks up the packet's next hop. The packet will then be forwarded to the corresponding outgoing interface.

Though this paper studies OSPF as a subject, the algorithms proposed in this paper works in general for other link state routing protocols (e.g., IS-IS) as well.

1.2 Systematic Framework for Securing Routing

Three types of techniques are commonly used to secure a routing protocol, *prevention, detection and recovery*, and *resilience*.

Prevention. The prevention approach seeks to harden the network protocol itself, typically through means of cryptography, to render attacks computationally infeasible.

Detection and recovery. Detection involves monitoring of the real-time behavior of protocol participants. Once malicious behavior is detected, we resort to recovery techniques to eliminate malicious participants, and restore network order and functionality that may have been impaired.

Resilience. The resilience technique seeks to maintain a certain level of availability even in the face of attacks. Here a desirable property is graceful performance degradation in the presence of compromised network participants, i.e., the communication availability of the network should degrade no faster than a rate approximately proportional to the percentage of compromised participants. Examples in this category include redundancy mechanisms, such as multipath routing.

1.3 A Data-plane Recovery Technique for Securing OSPF

In this paper, we focus on data plane security. We propose a data-plane intrusion recovery technique for the OSPF protocol.

In the data plane, a severe attack is the *blackhole/greyhole* attack. The blackhole attack is where a malicious router drops all packets they are supposed to forward. On the other hand, the greyhole attack is a selective forwarding attack where the malicious router drops packets of its choice.

Previous work has proposed Secure Traceroute [9], an intrusion detection technique for detecting malicious routers that drop packets. Secure Traceroute allows an endhost to set a complaint bit in the packet header when its packets are being dropped. When a router sees the complaint bit, it can initiate a Secure Traceroute request to a downstream router. The Secure Traceroute packet is cryptographically

secured such that it is indistinguishable from a regular packet. Secure Traceroute allows a router to pinpoint where the misbehaving router is located downstream.

Our goal is to provide an automated recovery mechanism after intrusions have been detected. The basic idea is to pick an alternative route to circumvent the malicious routers when the default route fails. The following properties are desired in the reroute algorithm:

Globally consistent reroute decision. While each router performs intrusion detection independently, the final rerouting decision has to be made in a globally consistent manner. In other words, when an alternative path becomes effective for a packet, all routers in the network must be immediately informed of the decision. Global consistency ensures loop-free routing.

Resilience to the slandering attack. Intrusion recovery mechanisms must be carefully designed to avoid the slandering attack. When intrusion detection is performed in a distributed manner, we often need to communicate each router's local intrusion detection outcome to other routes in the network in order for the detection outcome to be globally useful.

Ideally, if a well-behaved router detects a misbehaving neighbor, it should announce its decision to the rest of the world so all other routers can preclude the malicious router in the route evaluation, and in the case of OSPF, the shortest path tree computation. Unfortunately, this approach is susceptible to the *slandering attack*, where a malicious router claims that a well-behaved router is malicious.

A simple mechanism to defend against the slandering attack is to perform majority voting. Unfortunately, majority voting is usually expensive, and it fails when the majority of voters are malicious.

Efficiency When misbehavior is detected, routers need to reevaluate routing decisions. The router reevaluation algorithm needs to be efficient. In particular, we would like its running time to be a low-degree polynomial.

The algorithm we propose is based on the idea of distrusting node pairs. When one router blames another, it may be a well-founded blame, or it may well be a malicious router slandering a good one. It is not always possible for a third router to judge which of the two is a well-behaved. Therefore, our algorithm declares these two routers as distrusting routers, and the path selection algorithm tries to avoid going through a pair of distrusting routers simultaneously.

This strategy gives us the following properties:

- Resilience to slandering attacks: If a malcious router slanders many good routers, it will effectively detach itself from the network, hurting the attacker itself.

- Cheating routers eventually get eliminated: Though there is a chance that a newly selected route contains the malicious router of a distrusting pair, if the malicious router continues to drop packets, other good routers will

then be able to detect it, and in time, the malicious router accumulates many complaints and effectively detaches itself from the network.

- If any good path exists between two nodes, there is a high chance of finding a working path even if the majority of routers are malicious.

2 Problem Formulation and Algorithm

2.1 Intuition

We consider a distributed intrusion detection framework. In the routing example, routers collaborate when delivering a packet from the source to the destination. A router is able to detect whether a downstream collaborating router is misbehaving and dropping packets. Each router performs intrusion detection independently.

Ideally, if a well-behaved router detects a misbehaving neighbor, it should announce its decision to the rest of the world so all other routers can preclude the malicious router in the route evaluation, and in the case of OSPF, the shortest path tree computation. Unfortunately, this approach is susceptible to the *slandering attack*, where a malicious router claims that a well-behaved router is malicious.

To avoid the slandering attack, a router should only trust itself. Therefore it only trusts the outcome of the local intrusion detection process. However, the route reevaluation has to be done in a globally consistent manner. And how is this possible if each router uses a different set of intrusion detection outcome? Without communicating the misbehavior detection outcome to the rest of the network, global consistency cannot be assured. This can result in routing loops in newly evaluated routes.

To address these problems, we propose the following CONS-ROUTE algorithm for recovery from detected intrusions.

We argued that it is dangerous to allow a router to blacklist any other router. We know that if router A broadcasts a complaint about router B, either router A is maliciously slandering router B; or router A is well-behaved in which case router B must be the malicious router.

When router C receives a complaint about router B from router A, it would be nice if router C could tell which of A and B is the good router. This is sometimes possible through some investigation on router C's part. However, in many cases, C may not be able to judge between A and B. Meanwhile, because in routing we require all routers in the network to make consistent routing decisions so as to avoid routing loops, hence we want all routers to be able to independently reach the same conclusion in order to make use of that information.

Instead of blacklisting A or B or both, we can use the following strategy: When A complains about B, all the routers in the network set A and B as hostile routers.

We would like the property that any path that goes through A does not go through B.

2.2 Problem Formulation

We now present a theoretical formulation of the problem:
Shortest Path Routing with Constraints

We have a directed graph with non-negative weights on edges, and a list of incompatible constraints, each constaint being a pair (u, v) stating that any path must not go through u and v at the same time. We would like to find the shortest path in between a pair of nodes satisfying the list of constraints.

Unfortunately, we prove that this theoretical formulation is NP-Complete. Please refer to Appendix A.

In Appendix B, we show a gap-preserving polynomial time reduction from set cover to the routing under constraints problem. Because set cover is $O(\log n)$-hard to approximate, the routing under constraint is also $O(\log n)$-hard to approximate.

2.3 Heuristic Algorithm

Because this problem is NPC, we propose the following heuristic algorithm. The algorithm we propose is similar to an algorithm that has been used in the literature for network partitioning [4].

First we consider a non-oblivious version of the routing problem. We assume the router determines the next hop based on the packet's source and destination address.

Consider the following variation of the CONS-ROUTE problem: We would like to look for a path that violates the minimium number of constraints from s to t.

We use the following strategy: each time we delete one node that causes a violation of constraints from the graph. And we compute the shortest path from s to t in the new graph. The following algorithm decribes which node to delete:

In each round, pick a node that

1) appears on the current shortest path between s and t;

2) violates a constraint;

3) if we delete that node, the decrease in the number of violated constraints is maximized.

Now we delete that node from the graph, and delete all constraints pertaining to that node.

Continue this process until either

1) we found a path that violates 0 constraints;

2) if we delete any node s and t will be disconnected.

In any step, when we delete a node, the decrease in the number of violated constraints can be negative, i.e., violated constraints increase. At the very end of this process, we pick the best path we found in history.

3 Evaluation

3.1 Comparison against Simple Alternative Routing Algorithms

We will compare our scheme with a few simple alternative path routing algorithms. We implemented the following three algorithms for the purpose of comparison:

Node-disjoint path The simple node-disjoint path algorithm does not use intrusion detection results when it reevaluates routes. It simply picks an alternative path that is node-disjoint from the default path. The algorithm fails when no node-disjoint path exists. This algorithm requires non-oblivious routing, i.e., routing decisions are dependent on both the source and destination addresses.

Alternative next-hop The alternative next-hop algorithm requires that the edge router closest to the sender to pick an alternative next hop other than the default one.

Simple Secure Traceroute based alternative path In this algorithm, whenever an endhost detects packet dropping and sets the complaint bit, the edge router closest to the sender performs Secure Traceroute and detects the first misbehaving router downstream on the path. Then it picks an alternative route to go around that malicious router. This algorithm requires strict source routing, i.e., the edge router must encode the new path in the packet header.

3.2 Simulation Setup

We construct a simulator capable of representing a static network topology and its link metrics. A combination of synthetic and actual network topologies of various sizes are used for evaluation [1].

The simulator will randomly select a set of faulty nodes from the topology. These malicious nodes uniformly drop all packets in the data plane.

For small network topologies, an exhaustive simulation is performed: all sets of endpoints will be tested. However, for larger networks, we randomly sample a certain percentage of source and destination pairs.

3.3 Metrics Evaluated

Success rate: The number of pairs of nodes that can communicate using our algorithm divided by the number of pairs that can communicate assuming we have perfect knowledge which are the compromised nodes.

Alternative path stretch: The stretch is defined as the path length between *s* and *t* found by our algorithm divided by the shortest path length found by an offline omniscient algorithm that knows exactly which routers are malicious.

Response time: We shall evaluate the response time of our recovery mechanism. Our CONS-ROUTE routing algorithm tries to find a path that does not traverse two distrusting nodes simultaneously. If the new route happens to include the malicious node of the distrusting pair, the new route still does not work. However, now more routers will initiate intrusion detection, and as that malicious router gets complaints from more routers, it is less likely to be included in an alternative path.

The response time of our recovery mechanism can be characterized by the number of failed rounds before a new working route is found. Note that the response time is also dependent on the real-time traffic pattern, i.e., which pairs of nodes are talking to each other. If only a single pair of nodes are talking to each other, it may conceivably take longer to isolate a malicious node than when many pairs of nodes are talking to each other. Since when there are many pairs of communicating nodes, a malicious node will be detected on more paths, and since routers on all these paths will be initiating intrusion detection simutaneously, the malicious router will accumulate complaints more quickly than the single pair case.

Ideally, we would like to use real traffic traces to evaluate the response time. However, before we are able to obtain such data, we use a simplified model here, where we have *t* pairs of communicating nodes, and assume synchronized intrusion detection rounds. We evaluate how the response time varies with respect to the number of communicating pairs.

3.4 Results

Figure 1 and Figure 2 compare various alternative path routing algorithms in terms of the success rate. The figures plot the ratio of improvement over the non-secured version of OSPF against various percentage of randomly compromised nodes.

Figure 3 plots the path stretch of the CONS-ROUTE algorithm. Both the maximum and the mean stretch are plotted.

Figure 4 plots the response time of the CONS-ROUTE algorithm. The response time is expressed in terms of the number of failed routes before a working path is found. Both the maximum and mean response time are plotted.

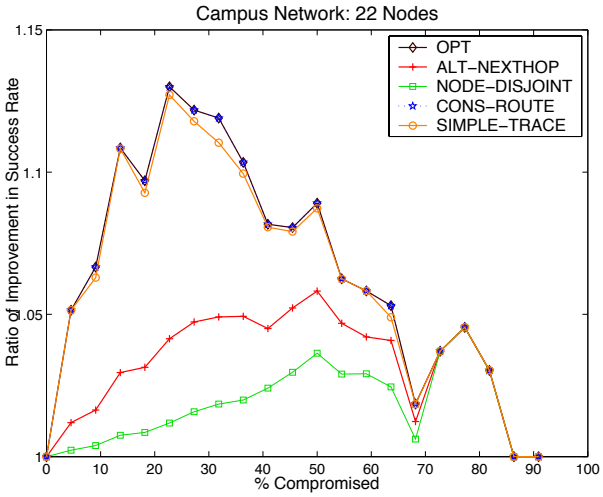


Figure 1: Success rate: 22 node campus network

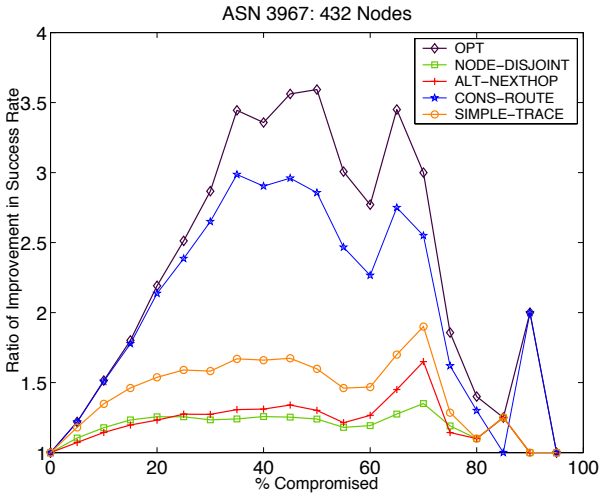


Figure 2: Success rate: 432 node network, ASN 3967

4 Related Work

4.1 OSPF Control-plane Vulnerabilities

Researchers have systematically evaluated the control plane vulnerabilities of the OSPF routing protocol [14, 7]. In particular, we make the distinction between

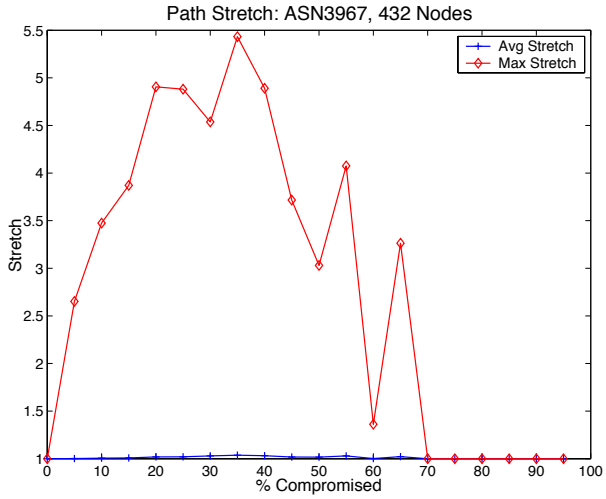


Figure 3: Stretch: 432 node network, ASN 3967

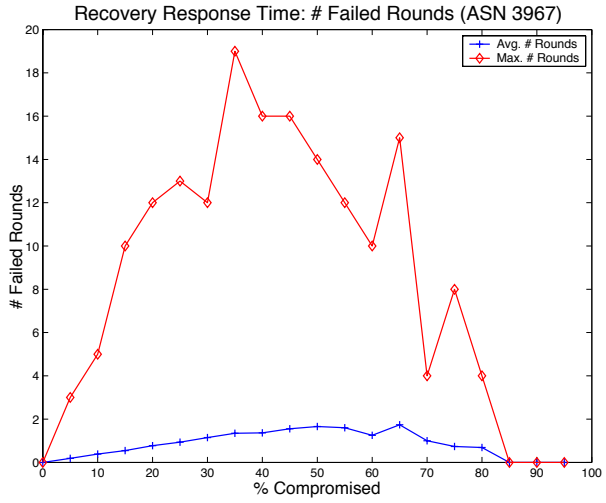


Figure 4: Response time: 432 node network, ASN 3967

outsider attacks and *insider attacks*. Outsider attacks come from unauthenticated parties who try to inject or alter routing updates. Outsider attacks can be addressed by a broadcast authentication [3, 11] or a one-time signature [10, 12] scheme. On the other hand, insider attacks come from compromised routers. Cryptography

alone does not suffice to counter insider attacks, for compromised routers possess the legitimate keys required to disguise as authenticated participants.

4.2 Securing OSPF

Table 1 summarizes the related work in the space of securing OSPF.

	Flavors of security		
	Prevention	Detection	Recovery/Resilience
Control plane	efficient broadcast authentication [3, 11], one-time signatures [10, 12]	[2, 13]	
Data plane		Secure TraceRoute [9]	[15, 5, 6]

Table 1: Literature Survey

Control-plane prevention techniques The following cryptographic constructions can be employed to authenticate link-state advertisements.

Efficient broadcast authentication TESLA offers an efficient broadcast authentication mechanism for securing LSU flooding. Cheung et al. [3] also proposed a similar LSU authentication scheme.

TESLA assumes weak time synchronization between routers, and offers delayed authentication.

To get around delayed authentication, we can build the authenticators before they are needed:

In OSPF each router periodically broadcasts LSUs unless triggered by changes in network dynamics. In OSPF, a router may already predict what its next LSU is before sending it out. Therefore, it can build an authenticator for its future LSA and disseminate the authenticator before the LSU is broadcast. When the LSA is broadcast, the advertising router reveals the new key used to compute the authenticator. In this way, we can achieve instant authentication.

This technique can be used when one of the following conditions is true:

1) Network condition is stable, and consecutive LSUs remain stable.

2) The LSA has a small number of discrete multiple values. For example, when the link metric remains stable and each LSA only announces whether the link is up or down. In this case, an advertising router can just reveal all possible authenticators for the next LSA, and reveal the key when the LSA is broadcast.[1]

[1]The OSPF specification does not specify how the metric should be defined. On some routers, the metric is statically configured on each router interface.

This technique does not work when the metric changes dynamically as link utilization changes, and when the network conditions are not stable. Under such circumtances, a router may not be able to predict what its next LSA will be. However, fast convergence is particularly important when network conditions are unstable. To achieve instant authentication in such circumtances, we need to use a more expensive one-time signature scheme as described below.
Using One-Time Signatures Potential one-time signature schemes we can use include hash chains, Merkle hash trees, BIBA and HORS. Among them HORS offers shorter signature length and more efficient generation/verification. Using one-time signature is more expensive than TELSA-like broadcast authentication, however it offers the property of instant authentication.

Control-plane intrusion detection techniques Researchers proposed intrustion detection techniques for OSPF. In particular, Chang et al. proposed the JiNao [2, 13] intrustion detection system for securing OSPF. JiNao is designed to detect control plane attacks. Several attacks that JiNao detects are outsider attacks and can easily be prevented through broadcast authentication. So instead of detecting these attacks after they happen, we should prevent them from happening at all. JiNao uses a combination of statistical and rule-based techniques. In this project, we will look at both control plane and data plane intrusion detection, and more importantly, we will look at how to combine intrusion detection with data plane alternative path routing techniques, to achieve automated attack response.

Data-plane resilience techniques Prior work has developed a number of methods of establishing routes that act as alternatives to the shortest path. These methods have been used in the past to relieve congestion on the shortest path; they may also be used to avoid faulty/attacking routers.

Prior methods have provided similar solutions aimed at minimizing alternate route computation time. Wang and Crowcroft proposed a solution of providing "emergency exits" [15] whenever possible to provide an alternative route to the shortest path, while Nelson, Sayood, and Chang proposed a very similar method of selecting alternative next hop routers [8].

5 Conclusion and Future Work

In this paper, we propose CONS-ROUTE, an intrusion recovery technique for securing the data-plane of the OSPF routing protocol. While the theoretical formulation of the CONS-ROUTE problem is NPC, we propose a simple heuristic algorithm. Simulation results show that the heuristic allows us to achieve near optimal results under realistic network topologies.

In our future work, we would like to extend the heuristic algorithm to support oblivious routing, where routing decisions only rely on the destination address of a packet. In addition, we would like to fully optimize the data structures in the implementation of the algorithm to exploit maximum performance. We also plan to implement the algorithm using an event-based simulator such as ns-2, and measure the real-time behavior of the algorithm.

References

[1] An Internet Topology for Simulation. `http://www.crhc.uiuc.edu/~jasonliu/projects/topo/`, December 2003.

[2] Ho-Yen Chang, S. Felix Wu, and Y. Frank Jou. Real-time protocol analysis for detecting link-state routing protocol attacks. *ACM Transactions on Information and System Security (TISSEC)*, 4(1):1–36, 2001.

[3] S. Cheung. An Efficient Message Authentication Scheme for Link State Routing. In *13th Annual Computer Security Applications Conference*, pages 90–98, 1997.

[4] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *25 years of DAC: Papers on Twenty-five years of electronic design automation*, pages 241–247, New York, NY, USA, 1988. ACM Press.

[5] Pierre Fransson and Lenka Carr-Motyckova. Brief Announcement: An Incremental Algorithm for Calculation of Backup-Paths in Link-State Networks. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2004.

[6] Yanxia Jia, Ioanis Nikolaidis, and Pawel Gburzynski. Alternative paths vs. inaccurate link state information in realistic network topologies. In *Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2002)*, pages 162–169, Jul 2002.

[7] Emanuele Jones and Olivier Le Moigne. OSPF Security Vulnerabilities Analysis , draft-ietf-rpsec-ospf-vuln-01.txt. Draft, Internet Engineering Task Force, December 2004.

[8] D. J. Nelson, K. Sayood, and H. Chang. An Extended Least-Hop Distributed Routing Algorithm. *IEEE Transactions on Communications*, 38(4):520–528, April 1990.

[9] V. N. Padmanabhan and D. R. Simon. Secure Traceroute to Detect Faulty or Malicious Routing. *ACM SIGCOMM Computer Communications Review*, 33(1):77–82, January 2003.

[10] Adrian Perrig. The BiBa One-Time Signature and Broadcast Authentication Protocol. In *Proceedings of ACM CCS 2001*, pages 28–37, November 2001.

[11] Adrian Perrig, Ran Canetti, J.D̃. Tygar, and Dawn Song. The TESLA Broadcast Authentication Protocol. *RSA CryptoBytes*, 5(Summer), 2002.

[12] L. Reyzin and N. Reyzin. Better than Biba: Short One-Time Signatures with Fast Signing and Verifying. In *Information Security and Privacy — 7th Australasian Conference (ACSIP 2002)*, July 2002.

[13] F. Wang, H. Qi, F. Gong, and S. F. Wu. Design and Implementation of Property-oriented Detection for Link State Routing Protocols. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, pages 91–99, 2001.

[14] Feiyi Wang and S. Felix Wu. On the Vulnerabilities and Protection of OSPF Routing Protocol. In *IC3N '98: Proceedings of the International Conference on Computer Communications and Networks*, page 148. IEEE Computer Society, 1998.

[15] Z. Wang and J. Crowcroft. Shortest Path First with Emergency Exits. *ACM SIGCOMM Computer Communications Review*, 20(4):166–176, September 1990.

A Proof of NP-Completeness

Instead of proving that finding the shortest path under a set of constraints is NP-Complete, we prove an easier version that finding any path satisfying a set of constraints is NP-Complete.

We reduce SAT to our problem which we refer to as PATH-CONS henceforth.

Assume we have a SAT instance $(x_1 \vee \bar{x_2} \vee x_3) \wedge (\bar{x_1} \vee \bar{x_2} \vee x_3)$, in poly-time, we can transform it into a PATH-CONS instance as in Figure 5 with the following set of constraints:

$(x_{1,1}, \bar{x}_{1,1}), (x_{1,2}, \bar{x}_{1,1}), (x_{1,1}, \bar{x}_{1,2}), (x_{1,2}, \bar{x}_{1,2})$;
$(x_{2,1}, \bar{x}_{2,1}), (x_{2,2}, \bar{x}_{2,1}), (x_{2,1}, \bar{x}_{2,2}), (x_{2,2}, \bar{x}_{2,2})$.

It is easy to show that if the SAT instance is satisfiable, we have a path from s_0 to s_2 satisfying the above constraints. On the other hand, if we have a path from s_0 to s_2 satisfying the above constraints, it maps back to a satisfying assignment for the SAT instance.

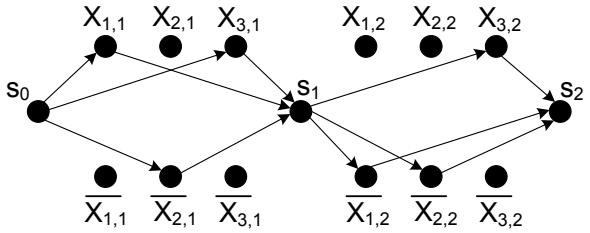


Figure 5: NPC Proof

B Reducing Set Cover to PATH-CONS

In this section, we show a gap-preserving polynomial time reduction from Set Cover to PATH-CONS, since Set Cover is $O(\log n)$-hard to approximate, so is PATH-CONS.

In Set Cover, we have a universe $U = \{u_1, u_2, \ldots, u_n\}$, sets $S_1, S_2, \ldots, S_n \subseteq U$, we ask the question: does there exist a set $V \subseteq U$, s.t. $\forall i, S_i \cap V \neq \emptyset$, and $|V| \leq k$.

We consider the following version of the PATH-CONS problem: Given directed graph $G = (V, E)$, a set of constraints $C = \{(u_1, v_1), (u_2, v_2), \ldots, (u_m, v_m)\}$, a source s and a destination t, we ask the question: does there exist a path from s to t that violates $\leq k$ constraints?

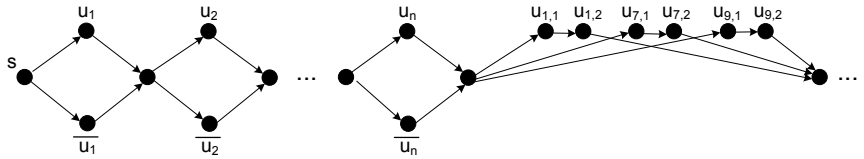


Figure 6: Reducing Set Cover to PATH-CONS

Given a Set Cover instance $S_1 = \{u_1, u_7, u_9\}, S_2 = \ldots$, Figure 6 shows a polynomial time reduction to the PATH-CONS problem. The set of constraints are:

$$(s, u_1), (s, u_2), \ldots, (s, u_n)$$

$$(u_{1,1}, \bar{u_1}), (u_{1,2}, \bar{u_1}), (u_{7,1}, \bar{u_7}), (u_{7,2}, \bar{u_7}), (u_{9,1}, \bar{u_9}), (u_{9,2}, \bar{u_9}), \ldots$$

Intuitively, if we look at a path from s to t (s being the leftmost node, and t being the rightmost node) in this graph, if the path traverses u_1, then u_1 is included in V; else if the path traverses $\bar{u_1}$, then u_1 is not included in V.

It is easy to prove that if the Set Cover instance has a solution $V = \{u_{i1}, u_{i2} \ldots u_{ir}\}$, s.t., $|V| = r \leq k$, then $\exists$ a path from s to t in the corresponding PATH-CONS instance that violates $r \leq k$ constraints. The path goes through $u_{i1}, u_{i2} \ldots u_{ir}$ in the first half, and for the latter half, pick just any path.

On the other hand, if the PATH-CONS instance has a path from s to t that violates $\leq k$ constraints, and if that path goes through $\bar{u_i}$ and $u_{i,1}, u_{i,2}$, we can always replace $\bar{u_i}$ with u_i, and the number of constraints violated does not change. After we perform this kind of replacement, we can find a path from s to t that never violates the latter type of constraints. Then $\forall i$, if the path traverses u_i then u_i is included in V. It is easy to show that the V constructed in this way is always a valid set cover, and of size $\leq k$.

A Case for Network Attached Storage

Project Report

Amber Palekar, Rahul Iyer

{apalekar, rni}@andrew.cmu.edu

School of Computer Science

Carnegie Mellon University

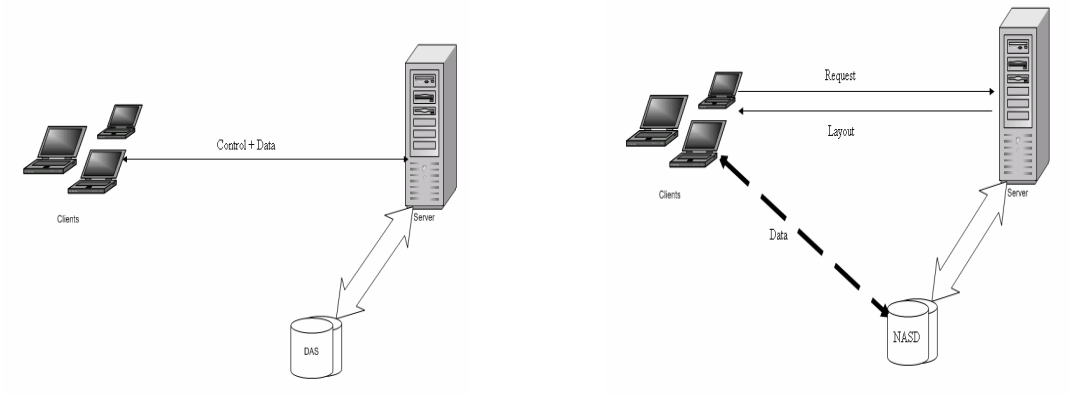
We are trying to answer the following question: "When is out-of-band access to data better than the traditional way (in-band) to access it (e.g. in NFS)". Our ultimate aim is to make this decision at the application level as opposed to the site level. Given a scenario or an application defined by parameters like file size, access patterns, we argue whether the traditional or the out-of-band distributed file system model will be more suitable for it in terms of I/O throughput and scalability.

1. Introduction

Data intensive applications like data mining, video conferencing have caused a shift in the data types towards richer ones like audio and video. This has resulted in continuous increase in the demands for storage bandwidth. Many efforts have been made to enable storage systems to deliver linearly scalable I/O bandwidth i.e. a performance linearly proportional to the number of clients and number of disks [Gibson98, lustre03, Panasas, Nagle04]. One such popular approach is asymmetric shared file system approach [Okefee98]. This approach separates the data and the control path allowing the aggregate I/O bandwidth to scale with the number of clients and number of disks.

In such systems a client typically makes a request through a file manager running on a separate machine. The file manager, also known as a metadata server, is responsible for managing the file system metadata, checking file access permissions and providing clients *layout* information indicating how data is present on the actual storage. Once a client is approved by the file manager, the former can directly transfer data to and from the storage device. Such architecture involves a message passing overhead in terms of the

aforementioned initial messages that flow between the three entities: the client, the file



(a) In-Band (b) Out-of-Band

Figure 1: Two ways of data access in a file level network attached storage. The in-band approach does not scale as it might get bottlenecked at the network link and/or the server. In the out-of-band scenario, the clients make requests to the server. The server hands them a "layout" of the requested file on disk. The client then uses this layout to access the storage directly for the data it needs.

manager and the device. These overheads are offset in *some scenarios* by the benefits that this model offers in terms of scalable I/O bandwidth due to the separation of the control and data paths. Here we investigate the cost to benefit argument for the parallel model given different scenarios. In particular, we analyze different scenarios based on the following set of parameters:

- Access Patterns
- File Size
- Access Size
- Concurrency
- Metadata Commits
- Read/Write Ratio

  Currently, we have carried out experiments for the first two parameters.

2. Background and Related Work

Traditional file servers

Traditionally, file servers using NFS follow a simple deployment form where all the data and control goes directly through the central file server. This model has obvious scalability problems due the network and server bottleneck.

Parallel file server architecture

Demands for storage bandwidth are increasing due to various reasons like improved client performance, data intensive applications, etc [Gibson98]. Various bottlenecks limit the traditional storage architecture from satisfying these demands. A main reason for the bottlenecks is the presence of a single store-and-forward type file server. There have been many research projects dealing with the idea of avoiding the single server bottleneck [gibson98, thekkat94].

Thekkath et al. discussed how separation of control and data transfer paths eliminate un-necessary control transfers and help distributed systems give a better throughput better than the traditional model [thekkat94]. Gibson et al. built a storage architecture build on top of object based storage that enabled the clients to talk to the storage subsystem directly without the overhead of going through the file server again and again [Gibson98]. Their NASD implementation provides storage bandwidth that is linearly scalable with the increasing number of clients and storage devices.

Another project on the initiated in recent times is Parallel NFS or pNFS[Gibson00]. This project aims at creating an extension to NFSv4 to adapt it to the out of band architecture.

3. Parameters of Interest

Access Patterns

Various workloads access files in different ways. For example, a multimedia workload is more likely to read a file sequentially from beginning to end. Database applications on the other hand are likely to access the file in fixed size chunks and in a random fashion. Alternatively, a supercomputing application where a large amount of parallel access is the focus, the data will be accessed in large blocks and in parallel.

A large number of small files: This scenario represents a typical real-life workload especially mail servers, web servers. If the file size involved here is small and the number of files is sufficiently large, we have a larger metadata to data ratio. The file manager in the parallel model is still the central entity which could be subject to bottlenecks. This problem is addressed by distributing the task in between multiple file managers.

File Size

Large file sizes: This scenario is more suited to the parallel model. In this case, there will be a large number of data requests as opposed to metadata requests. Thus, the involvement of the metadata server is fairly low. This exposes the file server bottleneck in the traditional model even more.

We consider the following parameters interesting to evaluate the relative performance of both the models, though we have not carried out any experiments related to them.

Access Size

The size of the requests is another parameter of interest. The smaller the request size, the larger will be the number of requests to the file. This means a larger number of metadata requests and transfers. The larger the request size, the greater the data to metadata ratio and hence a better amortization of the metadata overhead.

Concurrency

In this parameter, we try to cover two aspects:

- *Number of Clients:* The number of clients that can access the storage simultaneously is an important parameter. The major concern is how the storage system scales with respect to the increase in number of clients.
- *Overlapping Data:* Another issue is that of overlapping data. When data accessed by clients overlaps, it becomes necessary to bring in some sort of locking protocol. Often, this can become highly contented and the overhead of a distributed locking protocol makes it harder.

Metadata Commits

When the clients need to access data, they get the file layouts from the metadata server. Whenever a client modifies the file to change the metadata, the changes exist only on the local copy at the client. This needs to be reflected on the metadata server so that the other clients accessing this file can see the changed metadata.

In order to do this, the metadata is committed at intervals. While increasing this interval increases performance by reducing the number of round trips, it decreases concurrency as fewer clients can see the latest copy of the metadata.

Read Write Ratio

The Read/Write ratio is another parameter that can affect the performance of the parallel model. For instance a low Read/Write ratio (large number of writes) will result in data

being constantly changed. As a result, there is a lesser chance for caching as the number of invalidations will be high. On the other hand a high Read/Write ratio means that there is less invalidation and the client can aggressively cache, reducing the number of round trips.

4. Panasas Storage Architecture

We do the evaluation using Panasas Active Scale Storage Cluster. The reason for choosing this product is its ability to simultaneously support both modes of data access: the DirectFlow data path (PanFS) and the NFS data path [Panasas].

PanFS is an ideal example of the parallel model - it decouples the data path from the control path. The storage devices are network attached Object-based Storage Devices (OSDs). Metadata is stored in a metadata server called the Director Blade.

The Panasas Storage Cluster has 5 major components:

- The object that is stored on the OSD
- The OSD itself (StorageBlade)
- The metadata server (DirectorBlade)
- The Panasas Filesystem client module
- The Gigabit Ethernet interconnect

The Panasas filesystem is implemented on the clients via the PanFS module. The module performs the following functions:

- Provides a POSIX filesystem interface
- Disk caching
- Striping/RAID
- iSCSI protocol support

We plan to carry out our experiments using a single PanFS shelf – which is a configuration containing 10 OSD's and one node for running the MDS. A Panasas shelf is typically linked to the customer network via 4 Gigabit Ethernet aggregated links [Nagle04].

5. Hypothesis

As mentioned earlier, asymmetric shared file systems involve an additional overhead of extra metadata messages passing during operations like file creation. The effect of this overhead depends on the type of implementation. These additional costs are often

overshadowed by the benefits offered by such file systems. Having said that, there are also scenarios where the costs involved in this model do not justify its benefits.

Larger data sizes (file and access size) will mean that more data gets amortized over the metadata transfers. It, hence, follows that the parallel model will be more beneficial than a traditional model when the size of data transferred is large as compared to the metadata transfers. We expect the direct mode of PanFS to perform better for big block I/O access patterns as well as for larger file sizes. As opposed to this, a scenario involving accesses by a large number of small files is less likely to justify the need of the direct mode of PanFS because of a small data to metadata ratio.

A higher read/write ratio shows off the benefits of aggressive caching of layouts or delegations in case of PanFS in direct mode. By this logic, PanFS direct mode is expected to perform (scale) better than the in-band mode for higher read/write ratios.

An I/O pattern that is sequential allows prefetching of data. Random I/O patterns do not allow this, and prefetching is largely wasted. We expect the in-band model to perform worse than the parallel mode on Random I/O as the lack of prefetching opportunity will result in a large number of acceses to the file server, thus making the server bottleneck faster. In the sequential case, we expect the performance to be comparable to the parallel case until the server bottlenecks due to a large number of clients.

6. Benchmarks Used

The project involves benchmarking the performance of the two models of data access using existing benchmarks and applications representative of typical loads.

We are using the following benchmarks for conducting our experiments:

Iozone: Iozone is a file system benchmark that generates and measures different file operations on varying file sizes [Iozone]. We use Iozone to measure the effects of file size on the two data access models. We also carry out random I/O experiments using Iozone.

PostMark: Postmark is a filesystem benchmark developed by Network Appliance. The benchmark is targeted at looking at filesystem performance when posed with a workload consisting of a large number of small files also known as ISP workloads. We felt this was relevant as this is the type of workloads that are posed to servers such as mail servers and Web servers.

We find the following benchmarks relevant for carrying out experiments for the remaining scenarios:

Iometer: Iometer is a tool to obtain performance measurements of an I/O subsystem for single and clustered systems [Iometer]. This tool provides good support for varying the read/write ratios accesses carried out.

Bonnie++: Bonnie++ is another tool used to perform various simple tests on a file system to measure its performance. We find Bonnie++ relevant for sequential and random data transfers. The benchmark allows use of large file sizes [Bonnie++].

IOR (Interleaved or Random): This benchmark represents a well formed I/O workload which involves large block transfers with less metadata transfers compared to the interactive workloads like email servers. IOR performs parallel writes and reads to/from a file using POSIX or MPI-IO calls and reports the throughput rates [IOR].

OSDB (Open Source Database Benchmark): Database accesses represent a very important type of workload with characteristics of random I/O operations. We chose OSDB since it is freely available [OSDB].

7. Experiments

7.1 Configuration

We carried out our experiments on a single Panasas shelf containing 1 Director Blade and 10 Storage Blades. We used a standard linux box with 2.4 GHz Intel P4 and 1GB of RAM on it (kernel version 2.4.24). The Director Blade is a 2.4GHz Intel Xeon with 4GB of DDR RAM and an 80GB ATA-100 Drive. Each Storage Blade is a 1.2GHz Intel Celeron with 512MB of SDR RAM and 500GB of storage capacity (2 X 250GB Serial-ATA Drives). We have used NFSv3 over TCP for our experiments.

All experiments were carried with a single client running a single thread. An important future task for us is to carry out experiments with multiple clients running multiple threads.

7.2 File Size

We used Iozone to perform this experiment. RAID5 was used to store the data.

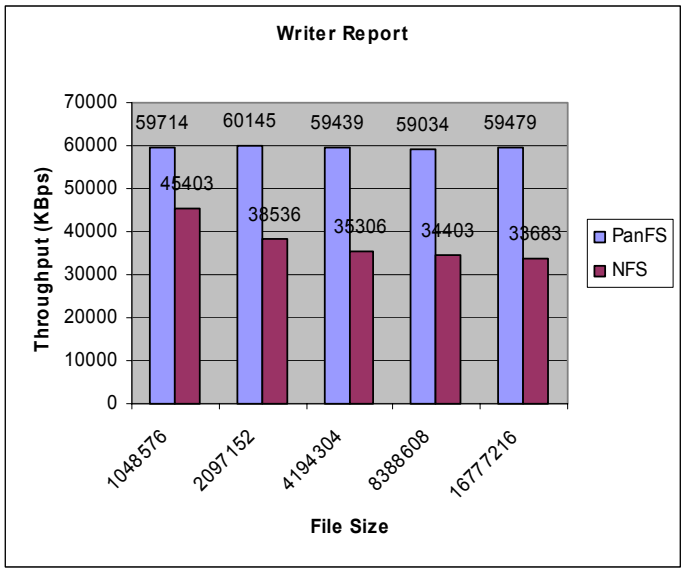


Figure 2: Sequential Write Performance for large file sizes

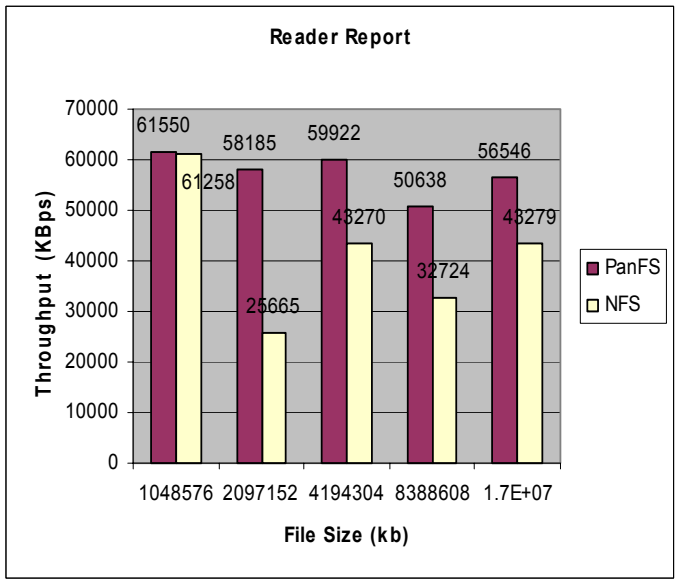


Figure 3: Sequential Read Performance for large file sizes

We took large file size to overcome the cache effects (as mentioned earlier, the NFS server under test had a large buffer cache size of around 2GB). As we see from figure 2 and 3, PanFS sustains an I/O bandwidth around 56-61 Mbps for large file sizes. NFS, on the other hand, suffers for large file sizes and performs worse than PanFS. This is due to the fact that the server side cache on the NFS is not in the picture. Thus, the I/O bandwidth that is seen is the raw disk bandwidth. For PanFS, this data is accessed by the client directly without going to the DirectorBlade. For NFS, every single byte of data is going through the server causing it to bottleneck.

We also compared NFS and PanFS for smaller file sizes, which turned out to be an implementation comparison as opposed to a protocol comparison.

Figure 4 shows that for smaller file sizes, we end up comparing the client side caching implementations for NFS and PanFS rather than the two protocols. The sharp dip in the throughput for NFS and PanFS towards the 1GB file size is because it overwhelms the buffer cache on the client (the client under consideration has 1GB of RAM). We are getting an anomaly towards the file sizes after 1GB where NFS is performing better than PanFS. This experiment does not contribute towards a solution to the problem that we are addressing but we thought it was interesting to mention the anomaly. The graphs clearly show that the NFS client side caching scores a big win over PanFS for smaller file sizes (which fit in the client buffer caches).

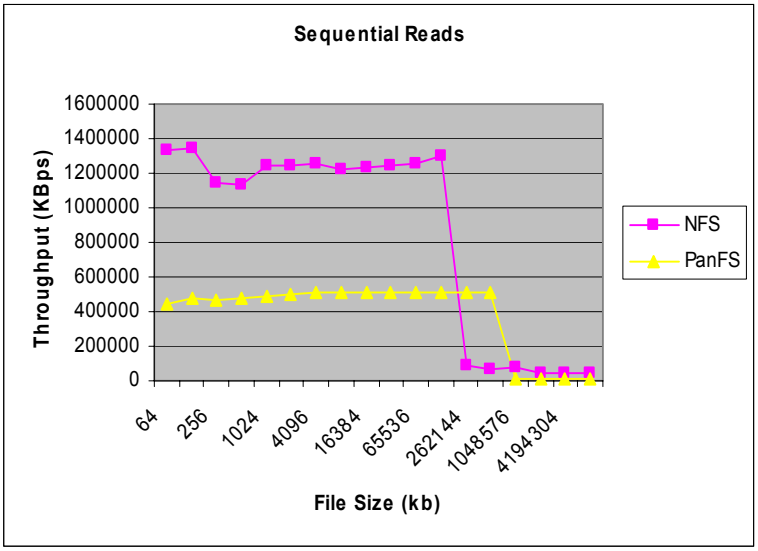
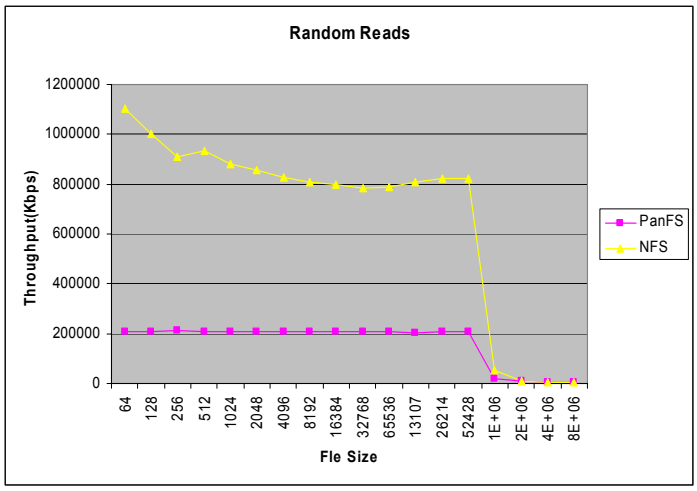


Figure 4: Sequential reads with small file size

7.3 Access Patterns

The following graphs show the bandwidth delivered on Random reads and writes. The large difference between the two is largely because of client side caching. NFS seems to make optimal use of the client buffer cache (the Linux Page cache), while PanFS doesn't seem leverage this so much. In fact, the size of the RAM on the client was 1GB, and we can see the bandwidth drop sharply at 1G file sizes, verifying this. However, we do note that this is an implementation effect and not an inherent flaw in the protocols just as observed in Figure 3 and 4. Also, it can be observed that PanFS does better than NFS on

larger file sizes, when the cache effect is nullified. (We did not run these tests on larger file sizes as we ran into technical issues with the equipment with respect to Random writes)



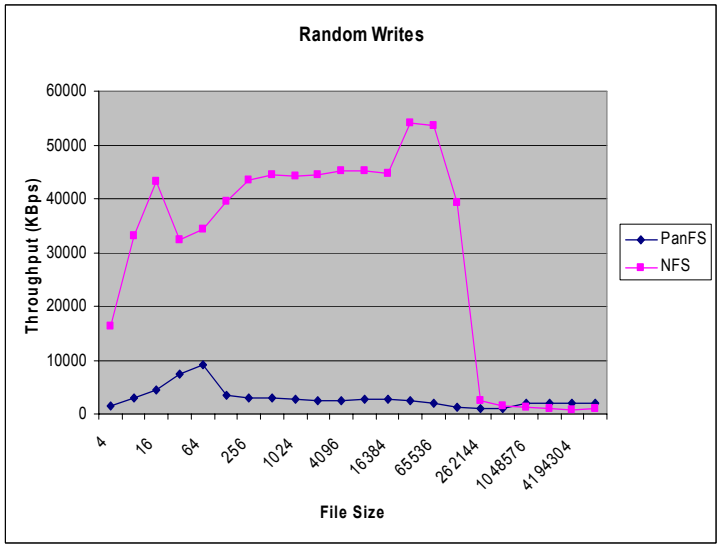


Figure 5: Random Reads/Writes

7.4 Large Number of Small files

One type of workload common in the enterprise is where there are a large number of small files. Typical of this type are email servers, web servers and net-news servers. Postmark is a filesystem benchmark that lets us evaluate this type of workload. Using Postmark, we created a number of files with sizes varying from 4KB to 32KB. The number of files varied from 1000 to 100,000. We then compared how NFS and PanFS fared on these tests.

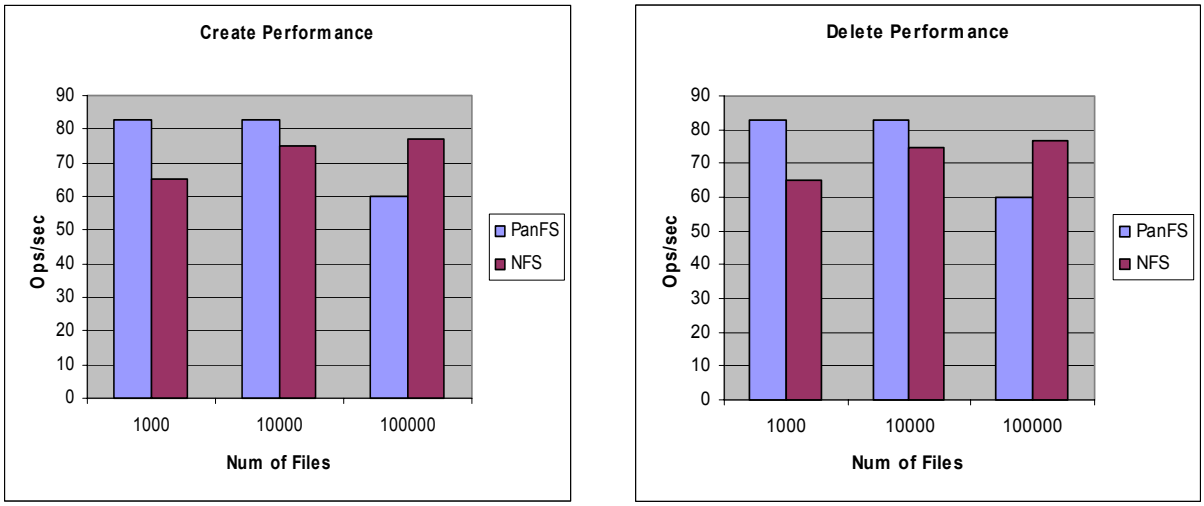


Figure 6: Create and Delete Performance

Figure 6 shows the create performance on both PanFS and NFS. It is observed that PanFS does better than NFS for 1000 and 1000 thousand files. However, NFS does better for 100,000 files. We speculate that this is a caching effect. The disk writes caused by the creation/deletion of the files are cached by the NFS server and the OSDs. The independent data and control paths on PanFS help it do better than NFS for 1000 and 10,000 files. However, when the number of files increases to 100,000, the writes are flushed to media on the OSDs because of the limited cache size. However, the large cache on the NFS server prevents this from happening in the case of NFS.

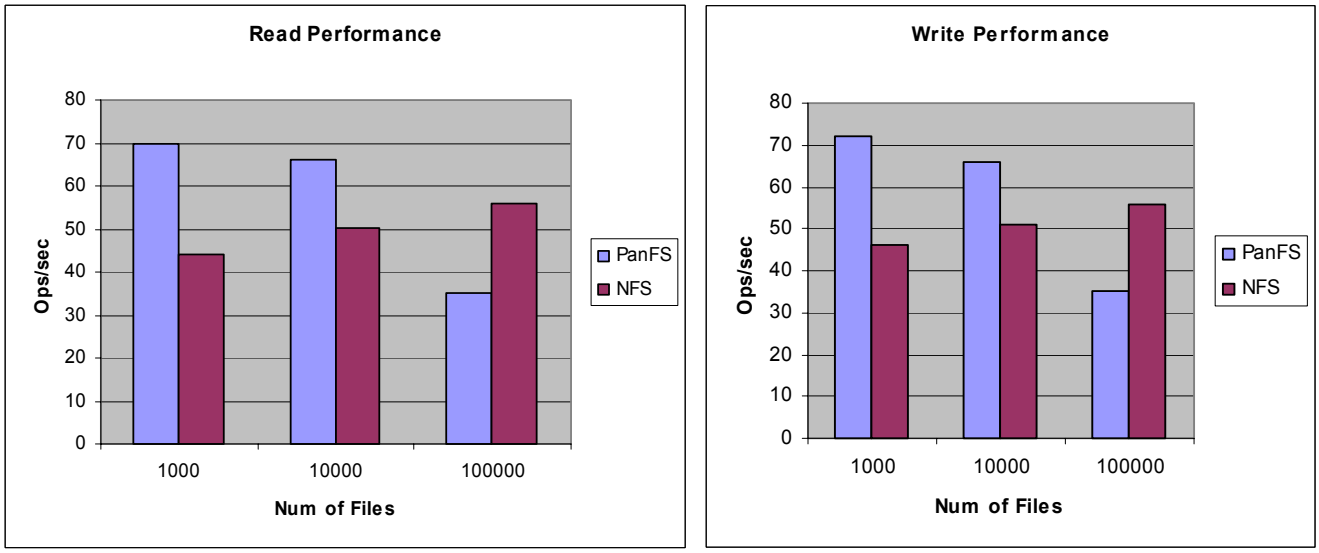


Figure 7: Read and Write Performance for large number of small files

We see a similar pattern in Reads and Writes as well. The separate data path of PanFS allows for higher bandwidth, but the moment PanFS is forced to go to disk, its bandwidth drops by nearly half.

In all, we believe that caching effects play a major role in our tests and using these results to draw inferences about the protocols would be incorrect. We would have liked to increase the number of files to a larger figure, but Postmark didn't seem to support it.

8. Logistics

Course Material Relationships

This project is relevant to the course material since we are dealing with evaluating two different storage architectures for different scenarios. The issues we are addressing come under the scope of Operating Systems and Distributed Systems which is the theme of the course.

9. Lessons Learnt

This experiment was an invaluable learning experience to us. During the course of the experiments, we realized that at many occasions, we came dangerously close to comparing two implementations against each other rather than the underlying protocols and architecture.

Also, this was our first attempt at benchmarking filesystems. We realized that there are many variable factors involved and the results we see could be heavily affected by them.

10. Conclusion

In our experiments, we found that out of band is better than in-band for sustained I/O bandwidth requirements for small file sizes. However, when small file sizes are involved, caching plays a huge role on the bandwidth seen by the clients. All in all, there is no silver bullet – it is a sum of many tradeoffs.

11. Future Work

An immediate next step is to carry out experiments on the remaining parameters: read/write ratio, concurrency, metadata commits. We envision a tool that will take the decision of the type of data access (in-band or out-of-band) per application. This can be, for example, a library which when linked with an application will act as a switch choosing one of the two data access types depending on the parameters that we mentioned.

12. Bibliography

[Bonnie++] http://www.coker.com.au/bonnie++/

[Gibson98] A Cost-Effective, High-Bandwidth Storage Architecture. Gibson, G.A., Nagle, D.F., Amiri, K., Butler, J., Chang, F.W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D. and Zelenka, J. *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems.*

[Gibson00] Parallel NFS Requirements and Design Considerations
http://bgp.potaroo.net/ietf/idref/draft-gibson-pnfs-reqs/

[Iometer] http://www.iometer.org/

[IOR] http://www.llnl.gov/asci/purple/benchmarks/limited/ior/

[Iozone] http://iozone.org/docs/IOzone_msword_98.pdf

[Nagle04] The Panasas ActiveScale Storage Cluster – Delivering Scalable High Bandwidth Storage. David Nagle, Denis Serenyi, Abbie Matthews. *Proceedings of the ACM/IEEE SC2004 Conference, November 6-12, 2004, Pittsburgh, PA, USA.*

[Okeefe98] Shared File Systems and Fibre Channel. Matthew T. O'Keefe. *Sixth NASA Goddard Space Flight Center Conference on Mass Storage and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems*

[OSDB] http://osdb.sourceforge.net/

[Panasas] http://www.panasas.com/

[PostMark] PostMark: A New File System Benchmark:
www.netapp.com/tech_library/3022.html