Selected Reports:
Fall 1997 Software Systems Course

edited by Garth A. Gibson
C. Colohan, C. Rosenberg, G. Steffan;
D. Petrou, J. Milford; O. Cheiner, I. Derenyi;
J. Gao, S. Rao, P. Venable; D. Rohde, R. Romero, P. Wickline;
M. Mateas, K. Nigam; M. Budiu, R. Budiu

April 4, 1998
CMU-CS-98-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

This technical report contains seven final project reports contributed by sixteen participants in CMU's Fall97 Systems Software introductory graduate course offered by professor Garth Gibson. This course studies the design and analysis of operating systems and distributed systems through a series of background lectures, paper readings, guest lectures and group projects. Projects were done in groups of two or three, required some kind of implementation and evaluation pertaining to the classroom 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 the completed projects, albeit with less thoroughness in the related work category than is expected in most conferences.

The reports that follow cover a broad range of topics. Specifically, these reports describe implementations and experimentation with: secure file systems when servers and administrators are untrusted; proportional share allocation for processor scheduling and its interaction with kernel realities such as locks; eventually serializable replicated databases with constant-order dependency checking; compressed file data structures optimized to specific access patterns; atomic, shared object semantics for distributed computing in JAVA; transaction semantics for federated agent databases; and user-level file service offering enhanced memory caching for remote files.

All reports include implementations and experimentation. Two involve operating system kernel changes, four use a middleware/library approach and one implements a client/server system. Three involve Linux specific modifications, one is specific to FreeBSD, one extends JAVA programming, and one exploits MPI communications. Evaluations include microbenchmark measurements, formal correctness evaluation, synthetic benchmarks, and more than a couple specifically developed application codes.

While not all of these reports report definitely and positively, all involve novelty in either the systems explored or the applications applied and all are worth reading.

Keywords: Security and protection, Scheduling, Transaction processing, Data compaction and compression, Distributed programming, Access methods.

Contents

Note: Pages are numbered in the lower-left corner of each page.

Project Final Report:
Secure Sharing with Satan's File System

Chris Colohan, Chuck Rosenberg, Greg Steffan
Software Systems–December 15, 1997

Abstract

The Secure File System (SFS) is a mechanism which allows the secure sharing of data between machines without direct or secure communication between them. This paper outlines the design of the SFS filesystem, and a prototype implementation. The performance of the system and the security it offers are also analyzed. Finally, future directions and improvements are discussed.

1 Introduction

A brave new venture, Hellfire consulting has just been formed. It has three consultants: Joe Belial who lives in LA, John Beelzebub from Toronto, and Joan Lucifer from Boston. They have just started coding a new software project for a hot new client, and need a way of sharing their code over the Internet while keeping it private. Joe has leased some storage space on a local Internet service provider, but he doesn't trust the ISP to keep their files secure. They want to be sure that only the three of them can read and modify their source code — and so they turned to the Secure File System.

The Secure File System (SFS) is a layer built on top of traditional Unix file systems which offers some guarantees about the origin and accessibility of the files contained on it. It does so making no assumptions about the integrity of the filesystem or administrators of the machine on which it resides. Communication between users is minimized — public keys must be securely exchanged between users once, and after that all communication is done through the file system itself.

The goal of this project is to implement a shared secure file system. Specifically, the implementation ensures that a read of an encrypted file is successful only if that file was written by a trusted entity, and that no entities other than the trusted group members can read the *plain text* contents of that file.

1.1 Design Overview

Since our design is built in a layer on top of existing file systems, all management of the media and all mechanisms for the sharing of files between machines are inherited from these file systems. We assume that the underlying file system is unsecure. Because of this, our implementation does not guard against the deletion or corruption of a file by a super user on a remote file system — however, we will be able to detect tampering with files. In addition, we assume that each local machine is a secure system — i.e., we do not have to protect the local memory of a user's machine and we can assume that information stored there is safe.

The system uses various encryption systems to protect the secure data and meta-data: a private-key based encryption scheme is used to encrypt the data portion of the file, a public-key encryption mechanism is used for key management, and a signature mechanism is used for authentication. We assume that the user's private key and other access related meta-data is stored in a secure and reliable way within the user's local system.

To manage shared access, an access list is associated with each file. The access list allows all enabled users of a file to know the identities of the corresponding trusted readers and writers, and provides them with the key to decrypt its contents. A naive implementation would maintain an access list for each file — this would incur much overhead, and would also require the individual maintenance of each access list. Furthermore, this approach does not mirror the way that shared file systems are typically used: as a means of exchanging data in a group of people working together. Our design overcomes these issues

and provides a sharing model which is more useful. We achieve this goal by introducing the notion of a *group*, its *administrator*, and a mapping to the directories where the group is trusted.

A group is a set of users who have read/write access for a specific set of directories. Associated with each group is an access list which contains a list of users in the group and the decryption key for all files owned by the group, which is encrypted under the public key for each authorized user. Each access list is stored as an encrypted file on the shared file system, which we shall refer to as the *group file*. Associated with each group file is a trusted administrator: a special user who is trusted with the maintenance of the group file.

Since each access list will be maintained by the group administrator, a regular user only needs to maintain four pieces of information in his local system to gain access to a shared file: his private key, his public key, the root directory of the sub-tree associated with each group, and the public key of the administrator of each group.

In our implementation, the group file will reside in the root directory for the group, and all sub-directories of the root directory for a group will inherit the group access of the root directory — e.g., `/usr/marketing/reports` inherits access from the root directory `/usr/marketing`. The use of a rooted sub-tree eliminates the need to directly associate each sub-directory or file with a group file.

1.2 Course Material Relationships

This project involves the following material from the course: security and access control; authentication; BAN logic; and file systems.

2 Using SFS

This section presents an overview of how to set up and use our prototype of SFS.

The first thing that is required to use SFS is a shared volume between the machines of the SFS users. This could be a NFS directory, an AFS directory, or any location shared by all of the SFS users. All of the SFS users must be able to read and write this partition. In addition, the full pathnames of files must be the same on each user's machine (which means an NFS partition must have the same mount point).

To install SFS on a user's system, the new `libc.so` must be placed in the dynamic link path (such as in `/lib`), and the executables must be placed in the command search path. Each user of SFS has to have a private directory on their local machine which holds their private key as well as access information for SFS. Once this directory is created, the `SFS_HOME` environment variable must be set to point to it. To generate a public/private key pair, each user must run the `keys_new` utility. This utility generates an RSA key pair using `/dev/random` as a source of a truely random seed. The user's public key can then be printed using the `keys_list` utility, and the public key must then be distributed to the other users of SFS through a reliable channel.

Each user has to have a list of users and their public keys so that SFS can verify the origins of files. This list is stored in `$SFS_HOME/users.sfs`, and is a list of user names and public keys. It is a plain text file, and can be created using a standard text editor. Each user also has a list of SFS file systems that they are participating in, which includes the administrator, full path of the file system, and a version number.

A new file system is created by making a list of authorized users with their desired permissions, and running the `grp_new` utility. Each user has to create a file `$SFS_HOME/groups.sfs` which lists the administrator, location, and version of each secure file system.

Once this is done, the user can treat the secure file system as any other file system. Files that are copied to or edited on the new file system are automatically encrypted before being written to the underlying file system, and any files that are read are automatically decrypted before reading them.

A number of tools are also provided for maintaining the SFS once it is created. `els` will list the files in the current directory including the correct (unencrypted) file size, embedded nonce, and author. `grp_copy` allows the user list and permissions from one file system to be used when creating another one. `revoke` removes a user from the access list of a file system, and re-encrypts all of the files on the file system so that the user can no longer read or write them. After revoking access, the administrator must inform all of the users of the filesystem of the new group file version number to ensure freshness and avoid spoofing attacks.

3 Design

3.1 Data Structures

The secure file system requires the user's private key and the access data for a group in order to access a group file–this information resides in the local, secure system. The access data for a group consists of the absolute pathname of the root directory for that group, and the public key of the administrator of the group. The local data structure for storing access data will have the following format:

$DirectoryTreeName, User_1, K_1$
$DirectoryTreeName, User_2, K_2$
$DirectoryTreeName, User_3, K_3$

The following describes each field: the $DirectoryTreeName$ field maps a directory tree to a specific group, the $User_a$ field is the name of the group administrator and is used for informational purposes, and K_a is the public key for the administrator ($User_a$).

The format for a group file is as follows:

$VersionNumber$
$Filename$
$User_1, \{K_G\}_{K_1}, K_1, WriteAccess_1$
$User_2, \{K_G\}_{K_2}, K_2, WriteAccess_2$
$User_a$
$Signature$

The following describes each field: the $VersionNumber$ field is the current version number of the group file to provide freshness; the $Filename$ field is the absolute pathname of the file; for each user ($User_i$) the group key (K_G) is encrypted under his public key (K_i) and a flag ($WriteAccess_i$) is specified describing his write permission for the group; the field $User_a$ is the name of the administrator; and $Signature$ is a signed digest which verifies the contents of the file.

The format for a regular file is as follows:

$Nonce$
$Filename$
$\{DATA\}_{K_G}$
$User_W$
$Signature$

The following describes each field: the $Nonce$ field is a unique value generated each time the file is written which can be used to provide freshness; the $Filename$ field is the absolute pathname of the file; $\{DATA\}_{K_g}$ is the data portion of the file encrypted under the group key; $User_W$ is the name of the user who last wrote the file; and $Signature$ is a digest, signed by the last writer of the file.

4 Security Evaluation

SFS attempts to provide some security where none existed before using a medium that can store data, but offers no assurance of the integrity or security of the data. On this medium, a hostile attacker can:

- read any files they like;
- alter files as they please;
- erase files;
- replace new versions of files with old versions.

A BAN analysis provided in the appendix outlines the logic that leads us to these beliefs.

The principal design goal of SFS is to provide some security while eliminating the need for secure and timely communication between the users of the file system. Secure and timely communication could

be achieved either through a secure messaging protocol between users or through a centralized trusted server — both of which would reduce the usability of the system if they were required. As a result of this, absolute security has not been attained, and the system is still open to spoofing attacks due to the lack of freshness in files.

Since there is no communication between users, a reader of a file has no way of knowing if what is being read is the most recent version of the file. This means that a hostile attacker could substitute older versions of files for newer version without being detected by anyone other than the original writer. To solve this problem, a nonce is inserted in the SFS file which is changed to a unique value every time the file is updated. If a writer of a file communicated the nonce to a reader through an outside secure channel, then the reader could be assured that the file is fresh.

The same problem affects the group files that list who is permitted to access a file system. A hostile attacker could replace a group list file with an older version at any time. As a result, it is impossible to revoke someone's access to a file system — they could easily replace the new group file with the old one, meaning that users other than the administrator would continue to unknowingly share files with that user. Hopefully it is uncommon for a user's access to be revoked, so explicit communication with the SFS users can be used to notify everyone of the revocation. To revoke someone's access, the entire file system is re-encrypted with new keys, and a new group file is created that excludes the revoked user. A version number is attached to the group file, and the version is incremented every time a user is removed from the group. To prevent the spoofing, the administrator has to notify all of the users of the file system of the new group version — once a user is using the new group version they no longer accept files with an earlier version number.

In addition to freshness attacks on SFS itself, there are a number of places where our prototype of SFS has potential holes. Our prototype uses the IDEA algorithm for encrypting file contents, MD5 for computing digests of files, and RSA for key distribution and signing digests. Any attacks on any of those algorithms would also work on SFS.

Our prototype uses a single IDEA key to encrypt all of the files in a file system. This means that there is a lot of data for a cryptanalysis attack on a single key. This could be prevented by limiting the usage of a single key, and having multiple keys per file system.

The regularity of the encrypted files can also be useful for cryptanalysis. Our prototype encrypts file data as written, while compressing the files before encrypting them would make the keys harder to crack as well as conserving storage.

Currently, our prototype allows a symbolic link between a directory on a secure file system and an insecure file system. This could be used to spoof secure files and replace them with a directory of insecure ones. We regard this as a bug in our prototype, but it also can make the secure file system more usable.

5 Performance Evaluation

Our hypothesis as stated in the design document is the following: we can build a secure file system which allows sharing under encryption, but sacrifices performance to do so. The following performance evaluation will show that we have proven the hypothesis with performance degradation to spare. We will investigate the overheads of the SFS system, as well as the performance on realistic benchmarks.

All of our timing measurements were performed on the local system hard disk. The local disk system was chosen over a remote filesystem like AFS to remove the variability in timing that would result from unpredictable AFS loads. Application timings were performed with command line timing. Function call timings were performed with specially written test software which utilized the system interval timer. In all cases, the measurement result reported is the result of averaging at least five measurements together. In the case of the function call timing, error bars are also reported which are +/- a standard deviation.

The overhead of the SFS system includes the storage of metadata in each encrypted file, performing encryption and decryption, using twice the memory to store a file (to encrypt to and from). As shown in Figure 1, the size of the metadata overhead is constant, with a size of roughly 170 bytes depending on the length of the absolute path name.

When an encrypted file is opened under SFS, the entire file is read into memory and decrypted. This adds significant overhead to the opening of files, as shown in Figure 2. For unencrypted files, SFS is about 10 times slower than with SFS disabled due to having to read configuration files. For large encrypted files, the slowdown is significant since decryption is fairly computationally intensive. Figure 3 shows that

Table 1: Slowdown of SFS on benchmarks.

Benchmark	Slowdown
co	38.7
make	7.19
latex	6.85

Table 2: Slowdown of SFS on `co` for varying number of users.

Number of Users	Slowdown
1	38.7
10	40.2
100	44.3

opening then closing the file performs similarly. These two experiments indicate that SFS will perform poorly for applications which open and close files without doing any work on them.

Since the entire encrypted file is read into memory on open, reading the file should be significantly faster. Figure 4 shows the performance of SFS when files are opened and then read to completion. For large files, the overhead of reading configuration files is almost completely hidden. However, for large encrypted files the overhead of decrypting the file is still dominant.

When writing to an encrypted file, SFS buffers the entire file in memory. When the file is closed, SFS performs encryption and writes the file out with one system call. As shown in Figure 5, writing a large encrypted file in blocks under SFS can outperform an writing to an unencrypted file, since only one system call is made to write the file. Normally, many system calls would be made on each block.

The throughput of SFS is given in Figure 6. Here SFS on encrypted files performs from about 20 to 100 times worse than with SFS disabled, depending on file size.

To measure the performance of SFS under simple usage conditions, we copied files to, from and within an encrypted directory. Figure 7 shows the slowdown of SFS relative to SFS disabled. When copying without encryption, the slowdown of SFS varies from 1 to 5 times, with the exception of a perturbation for the 100 byte file. When copying to or from an encrypted directory slowdown varies from 5 to 25 times, and when copying within an encrypted directory slowdown varies from 5 to 40 times. All copies involving encryption have a peak slowdown for 10KB to 100KB files, since encryption dominates for these file sizes. For files greater than 100KB the slowdown improves due to the fewer numbers of system calls for writing. For large encrypted files the performance degrades again since SFS requires twice the memory for the file (to encrypt/decrypt), and thus the memory requirement approaches the cache size (64MB).

We now investigate the performance of SFS on several realistic workloads: `co`—an RCS check out of 41 files (the source for SFS); `make`— compiling SFS with gcc; and `latex`—compiling the SFS design document. As shown in Table 1, `co` achieves a slowdown of nearly 40 times when checking-out into an encrypted directory. This slowdown is large because `co` does little work on the files, it simply reads and writes them. `make` and `latex` have similar slowdowns of around 7 times when compiling in an encrypted directory.

Finally, we investigate the performance of SFS when varying the number of sharing users in a group. As shown in Table 2, slowdown for `co` increases slightly when the number of users varies from 1 to 100. This is due to the overhead of reading a large access list for every file read.

6 Prototype Implementation

The prototype which we have prepared of SFS under Linux demonstrates the feasibility of the security design. The prototype was created by altering libc.so, so that dynamically linked executables on the system would have their file related system calls intercepted and redirected through the SFS code. In

particular, whenever a file is open()ed, it is checked to see if it is on SFS. If it is not, it is passed on to the OS, and all future calls on that file descriptor are also passed to the OS to handle. If the file is on SFS, then the file is read and decrypted into a memory buffer, and all future calls are redirected to the memory buffer. When the file is closed, then it is encrypted and written back to the underlying file system. Ideally, the encryption and decryption should be transparent to applications. So the SFS effectively operates as a layer on top of any existing file system, allowing it to be used to enhance the security of a wide variety of configurations.

Due to time constraints, not every file related system call was fully implemented in SFS, but a subset robust enough for a significant set of Unix tools to work was completed. Of the tools we tested, the ones that work without errors are `cp`, `mv`, `rm`, `emacs`, `netscape`, `rcs`, `gcc`, `make`, `ld`, `ftp`, `latex`, `gv`, `dvips`, `xdvi`, and `xfig`.

What is more interesting is the commands that don't work. These failures can be explained in terms of the system calls we have not implemented in libc. We do not support memory mapped files, so any program that uses mmap() will not work correctly. This includes running executables from our file system, and loading files in StarOffice 4.0 Beta.

The functions `stat()` and `lstat()` return the size of the actual file on disk, including the encryption metadata. This means that commands such as `ls` will report files to be larger than they are if copied to a non-encrypted file system. As a result of this discrepancy, `zip` and `tar` get confused and do not work correctly.

The two problems listed above can be resolved within libc. The third one is not so simple. If a program does a `fork()` followed by an `exec()` (such as a shell loading a program to run it), then the child process is supposed to inherit the parents file descriptors. Our implementation keeps a file descriptor table in memory to track encrypted file information. Since this table is in user memory, it is re-initialized to zero when the new program starts. This means that our library can "forget" that one of the standard streams has been redirected to an encrypted file, since that is determined when the file is opened. As a result, redirecting stdin, stdout, and stderr to or from encrypted files is unreliable, since doing so sometimes bypasses our encryption mechanism producing unencrypted files which our library views as corrupted.

7 Future Work

There is a lot of room for improvement in the design of SFS itself. The SFS design has two weak points, both of which involve freshness in files. Since a writer of a file can only communicate with the reader of the file through the file itself, there is no way of a reader validating the freshness of a file without communicating with the writer. This also applies when removing a user from an access group — there is no way of knowing that an access group is fresh without contacting the group administrator. A more efficient way of communicating freshness information for files and groups needs to be found.

Our prototype of SFS has a lot of room for improvement. Since security is its primary concern of SFS, the security is the first thing that should be addressed. Stronger encryption algorithms and longer keys could be used to improve security. More than one IDEA key could be used for an access group to improve resistance to cryptanalysis. File compression would also strengthen security. When looking at the PGP source code, it tries to erase all traces of unencrypted data from buffers in RAM after they are no longer needed — this practice should be adopted in our prototype. Internal buffers could also be stored in non-pageable RAM, so that unencrypted data is never recorded on a non-volatile medium. Existing, well tested mechanisms for public key storage and distribution could be used, such as using PGP key management and key rings. Easier to use administrative tools would make mistakes less common, and therefore improve security. And formal verification of code that handles unencrypted data and keys could be used to help ensure that they implement the specification of SFS.

Once security is addressed, the usability of SFS is a concern. If SFS is easier to use, then it is more likely to be used. SFS is no protection at all if it is not used. Firstly, SFS should be complete, and offer all the features that other Unix file systems offer. Currently, the inability to mmap() files and the inability to execute files from SFS are the two main holes. These features should be added to make SFS a complete filesystem.

Performance is also important if SFS is to be usable. There are a number of ways that the performance of SFS can be improved. Currently, files are treated as single units, which are encrypted and decrypted all at once. A blocking implementation would be able to address encrypted files a block at a time, and reduce

both open and close latency, as well as disk I/O. A blocking implementation would also work around the current problem of the entire file being buffered unencrypted in memory, and hence reduce the memory usage of SFS. If we find that files are not always read in their entirety, a blocking implementation would also help performance by only decrypting the parts of a file that are actually used.

Reliability is also a major concern in any file system. Right now SFS still has a number of unanticipated features which need to be corrected before it would be called robust. Since it is promising security, using program verification to validate that the implementation matches the specification. Error handling could be improved to provide more friendly and informative information to end users.

8 Conclusions

The secure file system was supposed to provide the ability to securely share files on an unsecure medium. We have achieved that in our prototype, at the cost of performance. Our testing was comparing the performance of local disk accesses to encrypted local disk accesses, which is a worst case scenario — when accessing a file remotely over a wide area network, the overhead of our encryption should not be as bad. The prototype has proven the feasibility and usefulness of SFS, and it shows that there is a lot of room for future improvements.

9 Appendix A — Exact File Formats

This appendix describes the exact file formats, byte for byte, that will be used to implement the secure file system. Because it was necessary to agree on some convention for word order, we have decided that all word and long word values will be stored in little endian order.

First, we define some notation:

Symbol	Interpretation
L_{K_G}	the length, in bytes, of a group key, typically 16 bytes
L_{K_U}	the length, in bytes, of a user's public key
L_U	the length, in bytes, of a user's user name including the null
L_S	the length, in bytes, of a signature, typically 16 bytes

The following is the format of a group file:

Byte Offset	Description
$0-3$	magic number to identify file
$4-5$	version number of the secure file system
$6-9$	version number of the group file
$10-11$	length of absolute filename field (m) in bytes (includes terminating null)
$12-15$	length of data portion of the file in bytes (n)
$16-(m+15)$	full absolute path filename, null terminated
$(m+16)-(m+n+15)$	data portion of the file - the group access list
$(m+n+16)-(m+n+L_U+15)$	user name, null terminated
$(m+n+16+L_U)-(m+n+L_U+L_S+15)$	writer's signature, signature includes all bytes $0-(m+n+L_U)$ in digest calculation

The group file data is a list of fixed size user entries. The entire list is sorted in ascending order by user name, to facilitate binary search of the data. Each of these entries has the following structure:

Byte Offset	Description
$0-31$	user name, null terminated, all unused bytes set to null
$32-33$	user's permission flags for the group
$34-(33+L_{K_G})$	the group key encrypted with the user's public key
$(34+L_{K_G})-(33+L_{K_G}+L_{K_U})$	the user's public key

The following is the format of a regular file:

Byte Offset	Description
$0-3$	magic number to identify file
$4-5$	version number of the secure file system
$6-9$	nonce for this file
$10-11$	length of absolute filename field (m) in bytes (includes terminating null)
$12-15$	length of data portion of the file in bytes (n)
$16-(m+15)$	full absolute path filename, null terminated
$(m+16)-(m+n+15)$	encrypted data portion of the file
$(m+n+16)-(m+n+L_U+15)$	user name, null terminated
$(m+n+16+Lu)-(m+n+L_U+L_S+15)$	writer's signature, signature includes all bytes $0-(m+n+15)$ in digest calculation

10 Appendix B — BAN Analysis of Scheme

In this section we apply *BAN* authentication logic [1] to our scheme.

10.1 Additional Notation

First, we augment BAN logic with some additional notation to allow the representation of our scheme.

10.1.1 Sets

We define four sets which enumerate elements of the scheme:

- $USERS$: the enumeration of all users.
- $ADMIN$: the enumeration of all administrators of groups, $ADMIN \subseteq USERS$.
- $ACCESS_G$: the enumeration of all users who have access in a given group G,$ACCESS \subseteq USERS$.
- $GROUPS$: the enumeration of all groups.

10.1.2 Elements

Now we define the following elements:

- U_i: a user where $i \in USERS$.
- K_i: public key for U_i.
- K_i^{-1}: private key for U_i.
- G_j: a group where $j \in GROUPS$.
- K_{G_j}: group key for group G_j.
- D: a digest (digests are taken on the entire file in question).
- F: a file.
- FN: the global, unique path name of a file F.
- DIR: the global, unique path name of a directory.
- GF_j: a group file for group G_j.
- GFN: the global, unique path name of a group file GF_j.
- WA_i: write access for U_i (a boolean value)
- U_a: administrator of a group where $a \in ADMIN$.
- U_m: last modifier of a file where $m \in ACCESS$ and $WA_m = true$.

10.1.3 Operators

We define the following operators:

- $K_i \rightarrow U_i$: U_i has key K_i.
- $WA_i \rightarrow U_i$: U_i has write access WA_i.
- $GF_x \rightarrow DIR_i$: GF_x applies to directory DIR_x.
- $FN\ in\ DIR$: the file name is in the directory.
- $F\ contains\ X$: file F contains element X.
- $D\ certifies\ F$: D is the correct digest of file F.

10.1.4 File Formats

We define the format for group files and regular files:

- $GF\ contains\ (GFN, \{U_i, K_i, \{K_G\}_{K_i}, WA_i | i \in ACCESS\}, \{D\}_{K_a^{-1}})$
- $F\ contains\ (FN, \{Data\}_{K_G}, \{D\}_{K_m^{-1}})$

10.2 BAN Analysis

We now proceed with the BAN analysis of our scheme. For this analysis, we will assume the identity of U_0, an arbitrary user.

10.2.1 Initial Information

We begin the analysis by listing the beliefs which are based on information which is stored in the local system, and is thus trustworthy.

$$U_0\ believes\ K_0 \rightarrow U_0. \tag{1}$$

$$U_0\ believes\ K_0^{-1} \rightarrow U_0. \tag{2}$$

Equation 1 and Equation 2 state our trust in our private and public keys.

$$U_0\ believes\ K_a \rightarrow U_a, \forall a \in ADMIN. \tag{3}$$

So we also have the public key of each group administrator.

$$U_0\ believes\ (G_j\ controls\ DIR), \forall j \in GROUPS. \tag{4}$$

This equation represents the mapping from each group to the directory which it controls. In reality, a group may control more than one directory, but we will continue under this simplified model.

$$U_0\ believes\ (U_a\ controls\ G_j), \tag{5}$$

such that for each $j \in GROUPS$, there exists exactly one $a \in ADMIN$–so each group is controlled by exactly one administrator.

$$U_0\ believes\ (U_a\ controls\ K_{G_j}), \tag{6}$$

such that for each $j \in GROUPS$, there exists exactly one $a \in ADMIN$–this states our belief that the administrator also controls the key for the group.

$$\frac{U_0\ believes\ (U_a\ controls\ G), U_0\ believes\ (G\ controls\ DIR)}{U_0\ believes\ (U_a\ controls\ DIR)}. \tag{7}$$

So by associativity of control, we believe that the group administrator may be trusted in files located in the directory.

9

10.3 Authentication of a Group File

Now we will analyze the authentication of a group file. To reduce the amount of notation in this proof, we will omit the indices from group files, groups, and directories since we are dealing with exactly one of each.

For group file GF, where $GF\ in\ DIR$, we begin by reading the group file:

$$U_0\ sees\ GF. \tag{8}$$

From 3 and 8, we get:

$$\frac{U_0\ believes\ (K_a \rightarrow U_a), GF\ contains\ \{D\}_{K_a^{-1}}}{U_0\ believes\ (U_a\ said\ D)}. \tag{9}$$

Since the digest is signed by the administrator, we believe that the administrator computed the digest. Using 9 and the digest:

$$\frac{U_0\ believes\ (U_a\ said\ D), D\ certifies\ GF}{U_0\ believes\ (U_a\ said\ GF)}. \tag{10}$$

Since the digest coincides with the content of the group file, we also believe that the administrator created the group file. We use our mapping from groups to directories (4), the global file name included in the group file:

$$\frac{U_0\ believes\ (U_a\ said\ GF), GF\ contains\ GFN, GFN\ in\ DIR}{U_0\ believes\ (U_a said (GF\ in\ DIR))}, \tag{11}$$

to assert that the administrator placed this group file in this directory. Using 11 and the fact that the administrator may be trusted in the directory (7):

$$\frac{U_0\ believes\ (U_a\ said\ (GF\ in\ DIR)), U_0\ believes\ (U_a\ controls\ DIR)}{U_0\ believes\ (U_a\ controls\ GF)}, \tag{12}$$

so the group file is controlled by the administrator since it came from the directory where we trust the administrator. Using 12 and the jurisdiction rule, we get:

$$\frac{U_0\ believes\ (U_a\ believes\ GF), U_0\ believes\ (U_a\ controls\ GF)}{U_0\ believes\ GF}. \tag{13}$$

Because we trust the administrator for this group, and that administrator created the group file, we may now trust the group file. Since we believe the group file, we therefore also believe its contents:

$$\frac{U_0\ believes\ GF, GF\ contains\ (U_i, K_i, WA_i)}{U_0\ believes\ (K_i \rightarrow U_i, WA_i \rightarrow U_i)}, \forall i \in ACCESS_G. \tag{14}$$

The access list of users, their public keys, and their write access are now available.

$$\frac{U_0\ believes\ GF, GF\ contains\ \{K_G\}_{K_0}, U_0\ believes\ (K_0 \rightarrow U_0)}{U_0\ believes\ K_G \rightarrow G}. \tag{15}$$

Assuming that we have access to this particular group, we may decrypt the key for the group using our private key, and we may trust that this is the proper group key. Finally, we must verify that this is the proper group file for the directory we are accessing (in case an evil user has switched around group files.

10.4 Authentication of a File

Now that we have authenticated the group file, we may proceed to authenticate the target file itself.

For file F, where $F\ in\ DIR$, we first read the file:

$$U_0\ sees\ F. \tag{16}$$

Since the file contains the global file name, we can look-up the group file that applies using 4 and 12:

$$\frac{F\ contains\ FN, FN\ in\ DIR, U_0\ believes\ G\ controls\ DIR}{U_0\ believes\ GF \rightarrow F}. \tag{17}$$

Using this result, 4 and 18:

$$\frac{U_0 \ believes \ GF \rightarrow F, F \ contains \ \{D\}_{K_m^{-1}}, U_0 \ believes \ K_m \rightarrow U_m}{U_0 \ believes \ (U_m \ said \ D)}. \tag{18}$$

The digest is signed by the user to last modify the file. We use the group file to look-up the public key for the user, and verify that the user computed the digest. Using this fact:

$$\frac{U_0 \ believes \ (U_m \ said \ D), D \ certifies \ F}{U_0 \ believes \ (U_m \ said \ F)}. \tag{19}$$

So the user also created the file, since the digest coincides with the file. Using the group file (14):

$$\frac{U_0 \ believes \ (U_m \ said \ F), U_0 \ believes \ WA_m \rightarrow U_m, WA_m = true}{U_0 \ believes \ F}. \tag{20}$$

This states that if this user has valid write access according to the group file, then we may believe that the contents of the file are valid. Using the group key (15), we may decrypt the data, and trust that it is valid:

$$\frac{U_0 \ believes \ F, F \ contains \ \{Data\}_{K_G}, U_0 \ believes \ K_G \rightarrow G}{U_0 \ believes \ Data}. \tag{21}$$

The following fact is obviated by the BAN logic: since we do not deal with *nonces* or the concept of *freshness*, we do not have protection against attacks which replace newer versions of files with older ones. However, this attack does not violate our original goal to ensure that only authorized users will be able to read the content of shared files. It does mean that we may not support the semantics of deleting a user from a group once the group file has been made public. An evil user who is deleted from the group could add himself back to the group by replacing the new group file with the old one which listed him as a user. In order to support deletion of users from groups, we have added a nonce to each file which must be verified by a secure means outside of SFS.

10.5 Other Actions

It is not necessary to list the ban logic for other types of file accesses, since they do no more than trust the information in the group files, as already shown. For example, to create a new group file, the administrator need only trust his local list of public keys, and then create the group file (in proper form). When any user in a given group creates a new file, the group file is consulted to obtain the group key, and then the file is created. A similar process is performed for updating an existing file.

11 Appendix C — Syscalls Intercepted in `Libc`

The following syscalls are intercepted to implement SFS, and are functional in our prototype:

close Close a file. If the file is encrypted, encrypt the buffered contents, write them out, and sign the file.

creat Create a new file. If encrypted, allocate a buffer to hold the contents of the file.

dup,dup2 Duplicate a file descriptor. If the file is encrypted, the duplication should be tracked.

fcntl Manipulate file descriptor. No special actions are taken for an encrypted file.

fstat Returns status information about the specified file. If the file is encrypted, then the meta-data size has to be subtracted from the returned length.

ftruncate Truncate a file to a specified length. If the file is encrypted, then it should be decrypted, truncated, then re-encrypted.

lseek Reposition read/write file offset. If the file is encrypted, keep a copy of the new offset locally for future reads and writes.

open Open and possibly create a file. If the file is encrypted, decrypt and read the entire contents of the file into a buffer.

read Read bytes from a file descriptor. If it is encrypted, simply read from the buffer.

rename Change the name or location of a file. If the file is encrypted, it needs to be decrypted before the move, and possibly reencrypted after the move.

write Write bytes to a file descriptor. If it is encrypted, simply write to the buffer.

The following syscalls are intercepted to implement SFS, but have not yet been completed in our prototype:

chroot Change root directory. All encrypted filenames should be adjusted to reflect this change.

exit Close any open files. If a file is encrypted, encrypt its buffered contents, write them out, and sign the file.

llseek Reposition read/write file offset. If the file is encrypted, keep a copy of the new offset locally for future reads and writes.

mmap,munmap Map or unmap files or devices into memory.

readv,writev Reads and writes to / from vectors.

stat,lstat Returns status information about the specified file. If the file is encrypted, then the meta-data size has to be subtracted from the returned length.

sync Commit buffer cache to disk. If there are encrypted files open, then they should be first written to disk.

truncate Truncate a file to a specified length. If the file is encrypted, then it should be decrypted, truncated, then re-encrypted.

References

[1] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

12 Appendix D – Figures Detailing System Performance

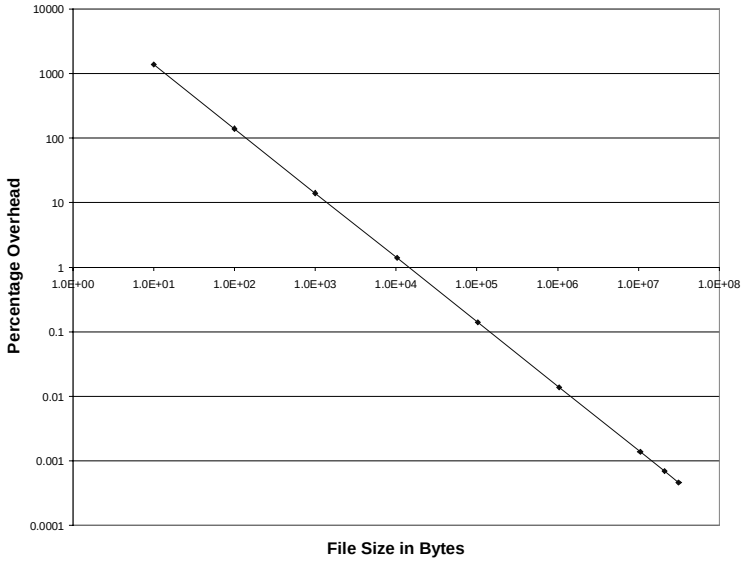

Figure 1: Percentage space overhead.

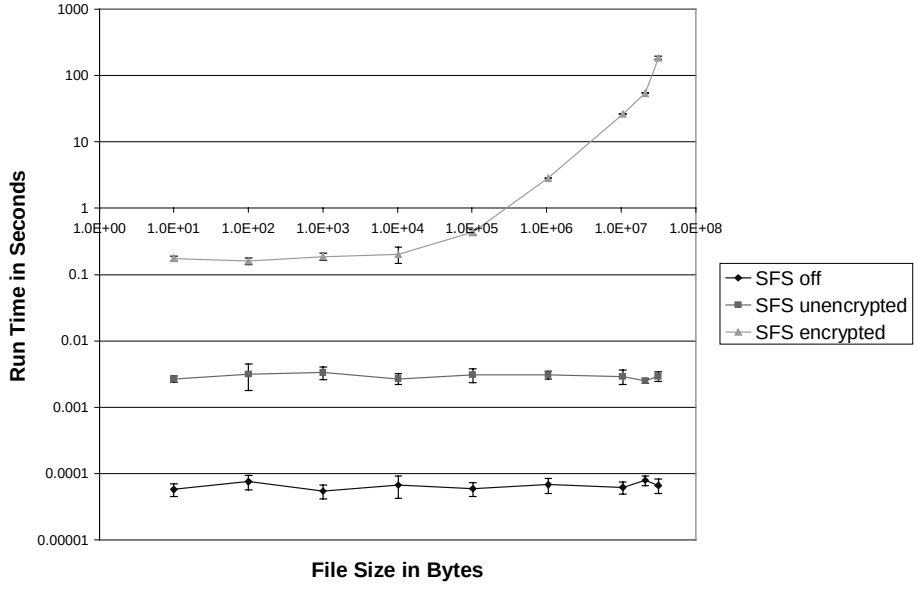

Figure 2: Performance evalaution of calling **`fopen`** in read only mode.

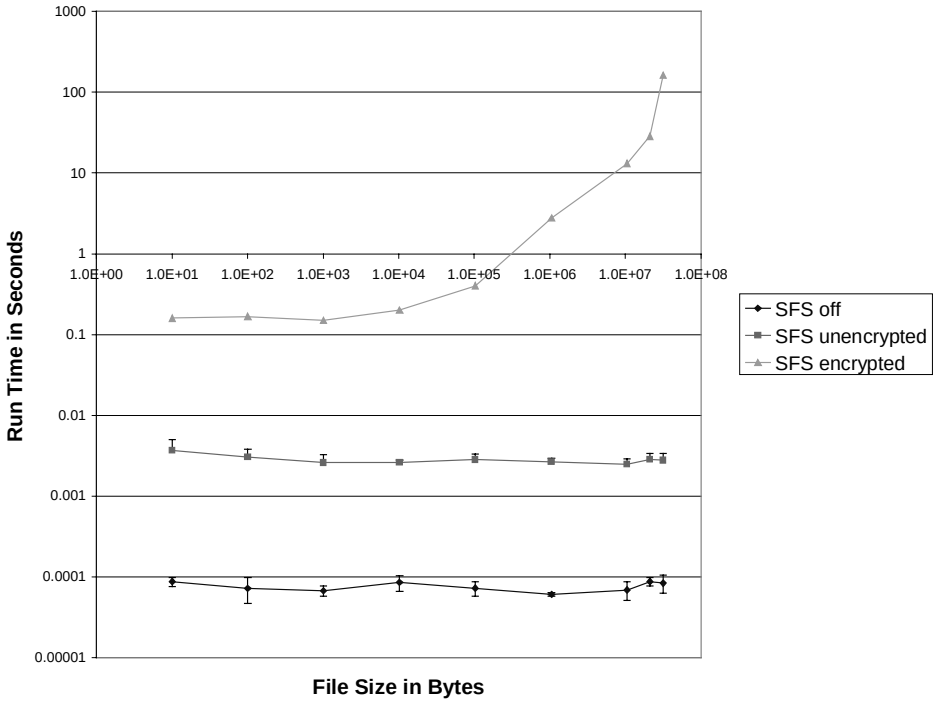

Figure 3: Performance evaluation of calling **`fopen`** and then **`fclose`**.

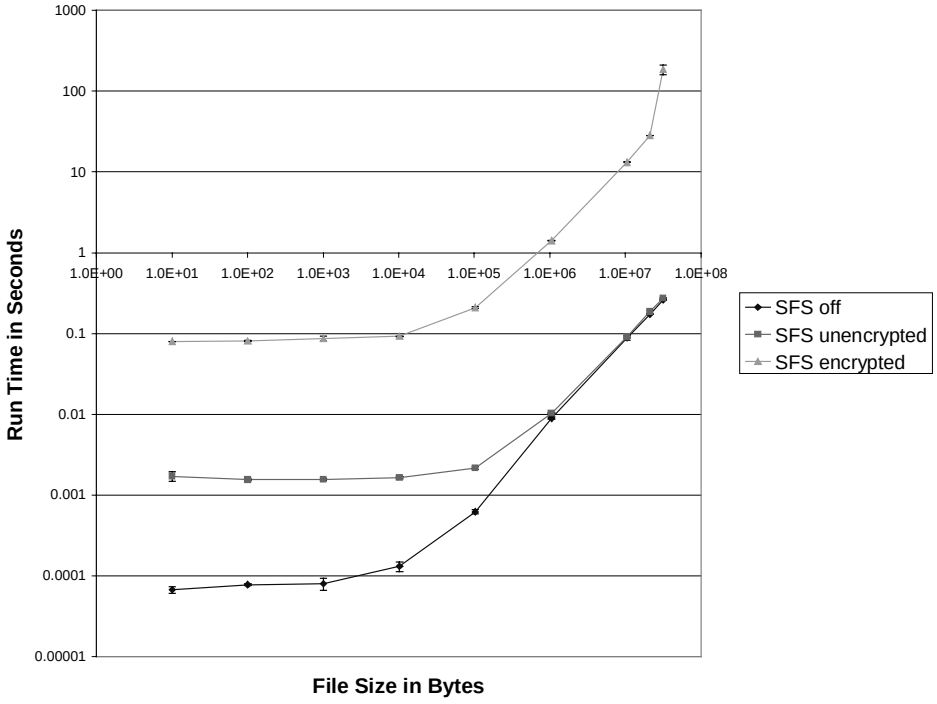

Figure 4: Performance evaluation of calling **`fopen`** and then **`fread`**.

14

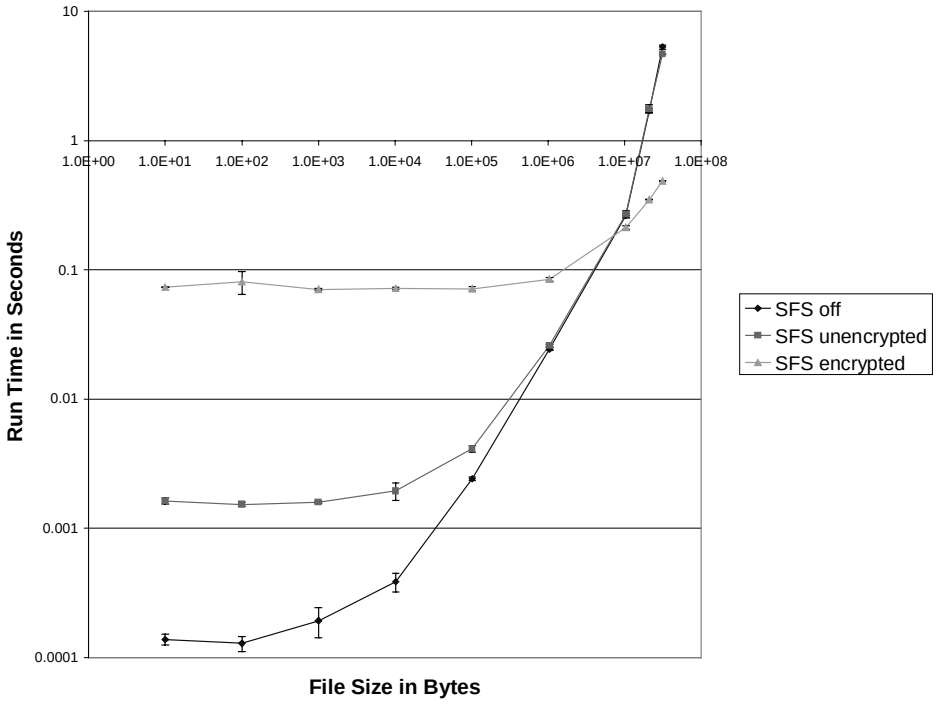

Figure 5: Performance evaluation of calling **`fopen`** and then **`fwrite`**.

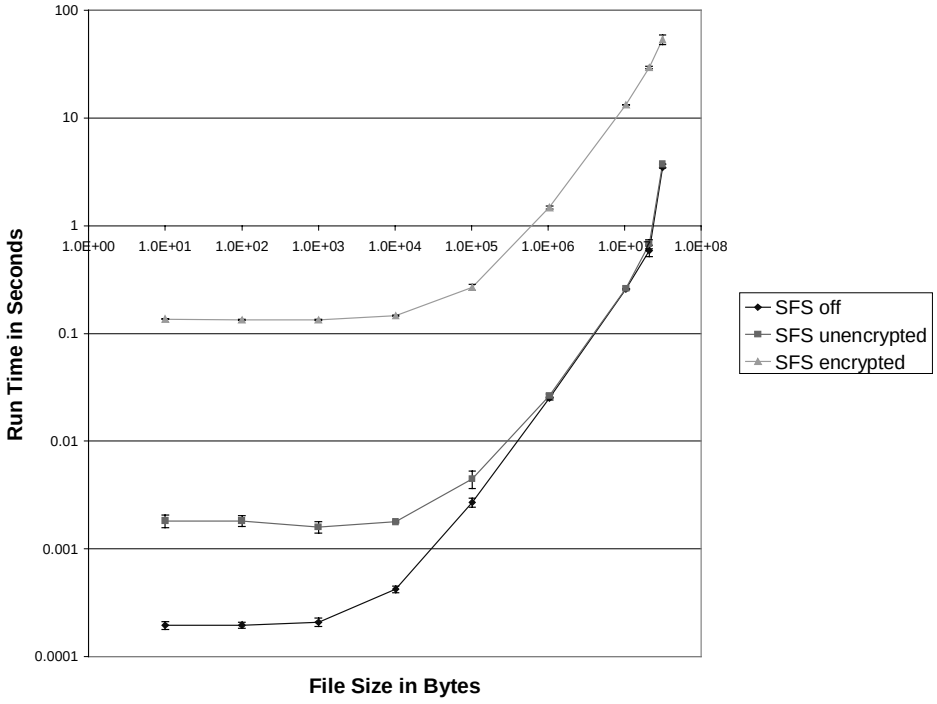

Figure 6: Performance evaluation of calling **`fopen`**, **`fwrite`** and then **`fclose`**.

15

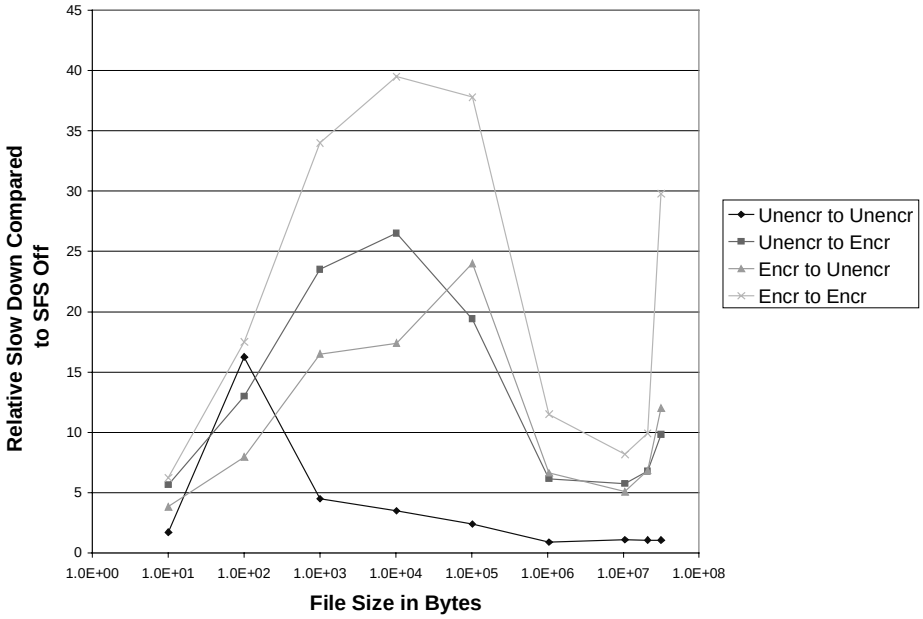

Figure 7: Relative slow down of the **cp** command.

16

Proportional-Share Scheduling: Implementation and Evaluation in a Widely-Deployed Operating System

David Petrou and John Milford
dpetrou@cs.cmu.edu, jwm@csua.berkeley.edu

Abstract

This paper explores the feasibility of using lottery scheduling, a proportional-share resource management algorithm, to schedule processes under the FreeBSD operating system. Proportional-share scheduling enables flexible control over relative process execution rates and processor load insulation among groups of processes. We show that a straight implementation of lottery scheduling performs worse than the standard FreeBSD scheduler. This initial result prompted us to extend lottery scheduling. Except for one test we run, our resulting system performs within one percent of the FreeBSD scheduler. We describe our design, evaluate our implementation, and relate our experience in deploying our lottery scheduler on production machines.

1 Introduction

This paper explores the feasibility of proportional-share resource management in a modern and well-understood operating system. Specifically, we employ lottery scheduling [Waldspurger & Weihl 1994] to allocate processor time in FreeBSD 2.2.5R. We describe and evaluate our implementation, a hybrid of the FreeBSD scheduler and lottery scheduling, and relate our experiences in using this scheduler on production machines. To begin, we motivate this paper with a summary of the benefits of proportional-share resource management.

Running processes make progress by consuming system resources such as processor cycles, network bandwidth, and storage. Considering processor cycles in particular, multiprogrammed systems allocate fixed quanta of processor time to running processes. A CPU scheduling algorithm based on proportional-share scheduling enables flexible control over the relative rates processes are allocated processor time quanta. For CPU-bound workloads, such as those found in some scientific and engineering environments, these rates are equal to the relative rates processes consume CPU.

For example, consider two independent CPU-intensive scientific simulations. These simulations begin running at timestep 0 and output intermediate results at the end of every timestep to be used by a 3-D rendering application. A researcher wants to visualize the results of these timesteps synchronously, but unfortunately the first simulation requires less computation and thus runs three times faster. Using a proportional-share process scheduler, the researcher can easily adjust the relative execution rates of the simulations so that the second process is chosen to run three times as often.

From a security perspective, proportional-share process scheduling enables users to carefully restrict the CPU consumption rate of untrusted binaries such as Java applications, or trusted binaries that use untrusted data such as WWW helper applications [Goldberg et al. 1996].

We have demonstrated the utility of proportional-share scheduling with respect to individual processes. Now we generalize and consider allocating proportions of processor time to individual users. In a time-sharing system, users own processes which compete for, among other resources, CPU time. Proportional-share process scheduling enables control over the relative rate at which users are permitted to use this resource. For example, one policy enforces users with CPU-bound jobs to consume the CPU at an equal rate. At a given time, if there is one user that wishes to make progress, he may consume 100% of the CPU, and if there are two users, they each may consume 50% of the CPU, *etc.*, regardless of the number of processes they own. Such control is useful for Internet service providers, in which hundreds of competing users log into one machine. With conventional processor schedulers, it is trivial for one user to monopolize the system with his own processes. Considering desktop workstations, another policy enables the console user to consume CPU time at a faster rate than users logged into the system remotely. Naturally, these policies also permit individual users to control the relative rate of CPU time that their own processes consume as in the scientific simulation example.

We believe that the features offered by proportional-share resource management are powerful and desirable. Lottery scheduling, introduced in 1994, is a scheduling algorithm that implements the described features but that is not in wide use today. Hence, our prime motivation was to determine if there are technical reasons why lottery scheduling has not supplanted standard operating system schedulers.

After building a straight implementation of the lottery scheduling algorithm we achieved the desirable features described. However, using the standard FreeBSD scheduler as a baseline for comparison, we experienced poorer responsiveness with interactive applications. We discuss the reasons for this unexpected performance anomaly and then present extensions to the lottery scheduling algorithm that remedy it. Our final implementation, which is currently running on two servers and one personal machine, nearly equals the performance of the FreeBSD scheduler under the benchmarks we run while providing the desired functionality of proportional-share resource management. These results indicate that lottery scheduling should be considered for wide use. However, we do not claim that our implementation

makes "optimal" scheduling decisions. Our experience with implementing and examining a process scheduler have raised some interesting questions that we wish to explore in future work.

The rest of the paper is organized as follows. In section 2 we describe both FreeBSD's and the lottery scheduling algorithm. Section 3 explains our extensions to the core lottery scheduling algorithm while section 4 details our implementation. We evaluate and compare our lottery scheduler with the FreeBSD scheduler in section 5. In section 6 we present our experience in deploying our scheduler on production machines. Section 7 discusses several situations in which our scheduler "does the wrong thing" and in which correct solutions are not obvious. Finally, section 8 concludes our paper.

2 Background

An operating system's process scheduler has several conflicting goals. The scheduler must schedule interactive processes so that they are responsive to user input despite not being able to accurately distinguish interactive processes from non-interactive processes. The scheduler must also efficiently schedule batch processes to maximize throughput despite potential lock conflicts between such processes. At the same time, the scheduler must ensure that no processes starve. In some contexts, schedulers must ensure that processes meet real-time deadlines, although we are not concerned with such systems here.

Following, we contrast the characteristics and goals of the FreeBSD scheduler with the lottery scheduler.

2.1 Scheduling in FreeBSD

FreeBSD [Lehey 1996, FreeBSD 1997] is a UNIX [Ritchie & Thompson 1974] operating system for the Intel 80386 platform based on UC Berkeley's 4.4BSD-Lite [4.4BSD 1994, McKusick et al. 1996] release. FreeBSD's scheduler is a typical example of decay usage priority scheduling also used in System V and Mach with 100 ms time slices. The scheduler is implemented with a multi-level feedback queue. Processes with equal priority are placed onto the same queue. The scheduler runs round-robin the processes from the highest priority non-empty queue. The scheduler favors interactive processes via two mechanisms: (1) the scheduler lowers the priority of processes that consume CPU, thereby moving them to lower queues, and (2), the scheduler preempts processes before their quanta expire if a higher priority sleeping process wakes up. FreeBSD's scheduler also employs static priorities for processes holding kernel resources. These priorities are higher than priorities held by userlevel processes and exist to enable processes to release high-demand kernel resources quickly. Starvation is avoided by periodically raising the priority of processes that have not recently run.

The FreeBSD scheduler has several limitations. [Hellerstein 1993] demonstrates the difficulty in using priorities in decay usage schedulers to adjust processor consumption rates. It is non-trivial and potentially computationally expensive to dynamically adjust priorities of running processes to attain service rates such as those easily attainable from proportional-share schedulers. Further, FreeBSD attempts to provide load-insulation via two crude and unacceptable mechanisms. The operating system can limit the number of processes that one user may run simultaneously, and the operating system can terminate processes that accumulate more than a certain amount of processor time. These mechanisms prevent a user from starting many processes that consume processor time slowly and from completing a few processes that consume a lot of processor time over a long period of time. Lastly, we have noticed odd behavior under the FreeBSD scheduler when simultaneously starting a large number ($\approx$ 100) of CPU-bound processes. Initially, the responsiveness of the entire system (subjectively measured by observing typing and mouse movement latency) becomes very poor. After a few moments, the interactivity improves only to become poor again moments later. We believe this behavior results from the CPU-bound processes synchronously moving up and down in the priority queues.

2.2 Lottery Scheduling

Lottery scheduling is a simple scheduling algorithm that provides proportional-share resource management. Processes are assigned a number of *tickets*. When the scheduler needs to choose a process to run it performs a *lottery* and chooses the process that holds the winning ticket. The ratios of tickets that processes hold determines the ratios of expected times processes win lotteries. Only the processes that are runnable (not sleeping, stopped, swapped out, *etc.*) are eligible for the lottery. Starvation is avoided since no matter how few tickets a process holds, eventually it will win a lottery. We note that responsiveness with lottery scheduling appears to degrade more gracefully than with the FreeBSD scheduler when the system is loaded with a large number of CPU-bound processes.

A *currency* mechanism enables users and processes to hold tickets in different denominations, allowing the aggregate execution rate of all of a user's processes to be varied with respect to other users. This mechanism provides *load insulation* among users. When making scheduling decisions, the scheduler converts the tickets that a process holds into tickets in a *base* currency according to an exchange rate derived from the number of base tickets that fund a user's currency.

A *compensation tickets* mechanism improves the relative CPU consumption accuracy for preempted and short-sleeping processes and also improves responsiveness for interactive programs. A process that wins a lottery but only consumes a fraction of its time quantum is granted a temporary number of compensation tickets valid until the process is chosen to run again. The process holds an *effective* number of tickets, equal to the compensation tickets plus the process's tickets, during the next time the process participates in a lottery. For example, a process with 10 tickets that blocked after using half of its time quantum will receive 20 effective tickets during its next lottery, making it twice as likely to be chosen to run than without these compensation tickets. Ideally, processes that often block for very short periods of time (before the next scheduling decision is made) will receive processor time in proportion to the relative number of tickets that they hold regardless of what fractions of their quanta they use. Further, interactive processes, which generally block for a long period of time after using a small fraction of their quanta, are very likely to be chosen to run during the next scheduling decision after they wake up.

3 Design

The preceding section described our first lottery scheduling implementation following the description in [Waldspurger & Weihl 1994]. With this implementation we were able to

control the relative execution rates of CPU-bound programs and insulate user workloads. In other areas the scheduler performed worse than the FreeBSD scheduler. We observed "choppy" response when editing files with background processes running and general system slowness when loading the system with many types of active processes. Our task at that point was to emulate features in the FreeBSD scheduler to remedy these anomalies while upholding the proportional processor utilization promises that lottery scheduling provides. In addition, user feedback prompted us to emulate the semantics of the UNIX `nice` utility. The following sections describe these topics.

3.1 Abbreviated Quanta

Users demand fast response from interactive applications. In the FreeBSD scheduler, various events on a process can potentially trigger a context switch before the running process uses its full time quantum. Two such events are a process waking up and a process receiving certain signals. If the process that receives the event has a higher priority than the running process, a context switch is forced immediately. The process will have a higher priority than the running process if it has used less CPU time than the running process recently (or if it was blocked in the kernel as described in section 3.2). We call this behavior *abbreviated quanta*, and we desire to replicate it in our lottery scheduler.

If a process receives an event of this type and has run for a shorter period of time during its last quantum than the currently running process[1], we force a context switch. The lottery scheduler will likely choose this process to run over other processes if it has earned many compensation tickets. To ensure that the preempted process will receive the processor in proportion to the tickets that it holds, the scheduler awards this process compensation tickets. (However, abbreviated quanta will still negatively affect processes that perform poorly when receiving less than a full time quantum, however). Note that after a process receives one of these events and we force a context switch, we do *not* unconditionally switch to this process but we rather perform the lottery as usual. One might therefore suppose that there is little utility to abbreviated quanta because the preempted process might simply reacquire the CPU. This chance exists, but it is not common, since we only force a context switch when the event-receiving process has used less CPU that the running process, making it probable that it earns more effective tickets than the preempted processes.

3.2 Kernel Priorities

Priority inversion [Lampson & Redell 1980, Birrell 1989, Hauser et al. 1993] is a well-documented problem in concurrent systems. Consider the following situation. A low priority process acquires a *non-preemptable* kernel resource such as a lock. A medium priority process begins running after a high priority process blocks waiting for the kernel lock to be released. The end-result is the high priority process remains blocked until the scheduler chooses the low priority process over the medium priority process.

[1]We actually want to compare the number of effective tickets in the base currency that the event-receiving process holds with the effective tickets in the base currency that would be earned by the running process if it is preempted at this point. These values are expensive to compute so we use the time heuristic instead. If our heuristic fails and we unnecessarily force a rescheduling event, our scheduler still performs correctly albeit there is the added overhead of an extra context switch.

In the FreeBSD scheduler, processes are assigned kernel priorities when they block waiting for a kernel resource. These kernel priorities are higher than priorities held by userlevel processes and are ordered in importance. Kernel priorities exist to enable processes to release high-demand resources quickly [Vahalia 1996, Jolitz & Jolitz 1996, McKusick et al. 1996].

The solution to priority inversion described in [Waldspurger & Weihl 1996] is for processes that are blocked on a resource to temporarily transfer their tickets to the process that holds the resource[2]. We admire the elegance of this solution, but it incurs overhead not found in the FreeBSD solution. A process that needs a kernel resources will find the resource in use, transfer its tickets, block, and trigger a context switch. These steps occur before the resource-holding process gets its boost in CPU consumption rate. The kernel priorities in FreeBSD minimize the chance that processes will find resources in use altogether.

Another example merits discussion. Consider a swapping process that goes to sleep waiting for a page from the swap device. In the FreeBSD scheduler, when the data becomes available the process wakes with a very high kernel priority, enabling it to be chosen to run during the next context switch. In this example, the resource held by the process when it wakes up, a page in main memory, is *preemptable* by other processes. However, preempting this resource is very expensive. A lottery scheduler that ignores kernel priorities can exhibit the following pathological behavior. A process wakes up when it receives a needed page. The process, however, is not chosen to run during the next context switch because there are other processes with more effective tickets in the base currency. These other processes, also needing memory, cause either the recently loaded page or another page within the process's working set to be evicted, requiring another page fault before the process can continue. The end-result is poor system throughput.

These are examples of the straight lottery scheduling algorithm proportionally provisioning the processor, yet failing to provide good overall system performance. The underlying reason why lottery scheduling fails is because it makes localized resource allocation decisions without considering the behavior of the system as a whole. We remind the reader that our goal was to achieve performance similar to the FreeBSD scheduler. To that end, we implemented the following kernel priority extension to lottery scheduling.

We maintain a list of processes that have woken up after being blocked in the kernel sorted by the FreeBSD kernel priority number. Before performing the lottery during a context switch, the scheduler checks to see if the list of processes with kernel priorities is empty. If it is not empty, the scheduler chooses the first process on this sorted list to run. At this point we solve the problems described above but we violate the proportional processor allocation that was the motivating reason for implementing lottery scheduling.

We now describe our fix that gives us the best of both worlds. A timer tracks the total amount of time that the process has run for, starting when it was picked in a lottery until it is descheduled in userland. The process may be descheduled many times in the kernel as it acquires kernel resources and the timer continues to increase when the process runs until the process is descheduled outside of the kernel. At this point, the scheduler computes its compensation tickets as described in section 2.2. If the process used more time than one time quantum, a *negative* number of

[2]This mechanism is similar to *priority inheritance* but permits additive ticket transfers from multiple blocked processes.

compensation tickets results. The more a process overran its quantum, the more negative compensation tickets it receives. Negative compensation tickets make a process *less* likely to be chosen to run. Our timer ensures that we do not violate the lottery scheduling goals[3] while enabling the scheduler to reduce kernel resource contention.[4]

3.3 `nice` Emulation

While deploying our lottery scheduler to production systems we received requests for our scheduler to support the UNIX `nice` utility. `nice` enables a user to vary the priority of his processes from -20 (high) to 20 (low) relative to all other processes in the system. We first show that mapping `nice` semantics to the lottery scheduler is not obvious and then present our design.

A naïve approach to emulating `nice` semantics is to map nice values to a process's tickets. This approach fails because only the relative execution rate of a user's own processes are affected. If a user has only one CPU-bound job running that is `nice`'d to +20, then it will receive few tickets, but since it is the only running process belonging to that user, the exchange rate will give it all of the base tickets that fund that user's currency. Alternatively, `nice` can affect the base tickets that fund a user instead of affecting the process's tickets. This approach also fails, but for a different reason. Now if a user `nice`'s a process, all of the user's processes will have a lower priority.

We implement the following solution. We map `nice` values to tickets and base tickets. When a process is positively `nice`'d, the process holds *at most* a number of tickets equal to the `nice` value to ticket mapping and *at most* a number of base tickets equal to the `nice` value to base ticket mapping when a scheduling decision is made. When a process is negatively `nice`'d, the process holds *at least* a number of tickets and *at least* a number of base tickets equal to the respective `nice` value mappings when a scheduling decision is made. (Recall that only root can negatively `nice` a process.) This solution avoids the problems with our approaches described above while satisfactorily emulating `nice` semantics and remaining in line with the lottery scheduler's goals.

4 Implementation

Our system is divided into two parts. The bulk of the code is in the kernel files implementing our lottery scheduler. The rest of the system are small userlevel programs that make system calls to adjust scheduling parameters. For brevity, we leave out some implementation details.

[3] There is a slight complication to our algorithm. Consider a process that spends very little time in userland because it continually makes blocking system calls. Such a process will receive the processor during the next context switch after it wakes up in virtue of having a kernel priority. Eventually it will be descheduled in userland (by the time slice interrupt) and it will receive a large number of negative compensation tickets, causing it to wait a relatively long time before running again. We would prefer a process to lose its kernel priority status more often and incur fewer negative compensation tickets. Our solution is to force a context switch if the process is entering userland (implying that it does not hold a kernel priority) and has accumulated more than one time quanta on its timer.

[4] It may be possible to achieve higher process throughput in the kernel if the ordering of FreeBSD kernel priorities is incorrect for certain workloads. However, as our goal is to achieve performance similar to FreeBSD's scheduler, we do not attempt this.

4.1 Kernel Functionality

To describe our kernel code we first describe critical pieces of the standard FreeBSD scheduler. There are 32 multi-level feedback queues called *runqueues*. After becoming runnable, processes are put onto the appropriate runqueue by the assembly language routine *setrunqueue()*. Processes are removed from runqueues when chosen to run by the scheduler. The assembly language routine, *cpu_switch()*, (1) saves process context, (2) chooses the next process to run (if no processes are runnable it idles), and (3) switches to that process. Apparently for performance considerations, this function violates the good programming practice that functions should "do one thing well."[5] [Lampson 1984]

We modified *cpu_switch()* to decouple the scheduling decision from the scheduling mechanism. Specifically, we rearrange some of the assembly code and make calls to a C function called *lott_choose_next_runner()*. We store ticket information along with other necessary scheduling information in a per-process structure called *lott_proc*. During boot-time initialization, our function *lott_init()* initializes a hash table that allows us to access *lott_proc*'s quickly. We also maintain a doubly-linked list of processes that are runnable. We replace *setrunqueue()* with a C function that manipulates this list. We add the "hand-created" process 0 and all future processes created by *fork1()* to our hash table with a call to *lott_add_proc()*, which assigns processes an initial 10 tickets. Likewise, processes are removed in *exit1()* with a call to *lott_remove_proc()*. We use a hash table to efficiently access a data structure called *lott_user* which holds the number of base tickets that fund all of the runnable processes corresponding to a user ID and a reference count that enables us to garbage collect these structures when all of the processes owned by a user terminate. The *setuid()* system call (*cf.* `login`) is modified to call *lott_add_user()* which creates a *lott_user* structure and funds the user with 1000 base tickets.

Any lottery scheduling implementation with compensation tickets and currencies will be more computationally expensive than the FreeBSD scheduler. As our goal was to achieve comparable performance to FreeBSD, we expended much effort in optimizing our implementation. Motivated by [Massalin & Pu 1989] we factor invariants, defer work, use many inline functions, and aggressively cache computed values.

The heart of the lottery scheduling algorithm is in *lott_choose_next_runner()*. First we check to see if both the lists of processes with kernel priorities and without kernel priorities are empty. If they are empty, we return 0, which causes *cpu_switch()* to idle. If the kernel priority list is not empty, we select the first process on this list to run. If it is empty, we proceed to the lottery scheduling algorithm. We need to find the total number of effective tickets in the system so that we can choose a winning ticket within this range. For each runnable process we perform the following steps. We check to see if our cached value of the process's effective tickets is valid. If it is not, we compute its effective tickets by dividing the number of microseconds in one time quanta by the number of microseconds used by the process (this value is stored by the timer described in section 3.2). We

[5] The opaque structure of this function is the reason for, in our opinion, it remaining largely unmodified from the earliest version of the code that we found. [Parnas 1972] explains that systems should be decomposed based on "difficult design decisions or design decisions which are likely to change." We find the poor modularization of the FreeBSD scheduler unfortunate, as it impedes scheduler experimentation.

then multiply this value by the number of tickets that the process holds to get the number of effective tickets. Now we check to see if our cache of the exchange rate between the process's effective tickets and tickets in the base currency is valid. If it is not, we compute this exchange rate by dividing the number of base tickets that fund the user's currency by the number of tickets in all of the user's runnable processes. We then check to see if our cache of base tickets held by the process is valid. If it is not, we compute it by multiplying the user's exchange rate by the process's effective tickets. At this point we implement the `nice` "at most/at least" algorithm described in section 3.3 by ensuring that the number of base tickets lies within the allowable range[6]. We go through these computations for each process and maintain a running count of the total number of base tickets in the system. When finished, we pick a random number between one and this number. Finally, we iterate through the list of runnable processes one more time to find which process holds the winning ticket.

A few details are worth mentioning. Floating point operations are not permitted in the kernel, so for accuracy we use fixed point arithmetic. Different parts of the algorithm require different amounts of precision, and some variables are capable only of representing a small amount of precision because the rest of their bits are required to represent whole number components. Therefore we use both 8 and 12 bit fractional components for different parts of the algorithm. For most of the algorithm we compute with 32-bit integers, allowing us to do arithmetic purely in hardware. Toward the end of the computation we must use 64 bits. Specifically, when computing a process's base tickets, the exchange rate and the effective tickets are both 32-bit values that when multiplied, are likely to result in a 64-bit integer. The Intel 80386 architecture can do a 32-bit times 32-bit to 64-bit operation in hardware. We use an inline assembly instruction for this operation as we could only get the gcc compiler to either store the result as a 32-bit integer (losing the most significant 32 bits) or promote the operands and perform a 64-bit times 64-bit to 64-bit slow software multiplication. The only 64-bit software emulated arithmetic in our algorithm is where we mod the random number by the total number of base tickets in the system.

We instrumented both the unmodified FreeBSD kernel and our lottery scheduler to provide us with profiling information, some of which will be discussed in sections 5 and 6.2. We can display information about the last n scheduling events, including information about processes releasing the CPU, processes waking up, and time quanta expirations. This information has been invaluable in helping us debug our implementation and in measuring our system against the FreeBSD baseline. A user program requests this information by making a system call with arguments that specify a buffer location in userspace, and the number, n, of entries requested. The user process then blocks until n scheduling events are made. At this point, the kernel calls *copyout()* to copy the requested information to the user buffer and calls *wakeup()* to unblock the user process. We note that there were several potential synchronization problems including the "lost-wakeup" problem that we had to avoid when implementing our profiling code.

[6] The mapping from `nice` values to base tickets is basetickets = $10^{\frac{1}{20}(-\texttt{nice}+20)+2}$.

4.2 Userlevel Programs

We now show how users adjust lottery scheduling parameters. All of these programs make system calls to achieve the described functionality. The first set of functions are available to all users.

`set_tickets -p<pid> -t<tickets>` — Changes the number of tickets held by `<pid>` to `<tickets>`. The user must own `<pid>` unless the user is root. `<tickets>` is in user tickets (as opposed to base tickets).

`run_tickets <tickets> <prog> [<arg1> ...]` — Executes `<prog>` with optional `<argN>`'s using `<tickets>` represented in user tickets.

`show_tickets [-u<uid>] [-p<pid>] [-x]` — Shows the number of tickets held by `<pid>`. The `<pid>` must be owned by the user, unless the user is root. A `<pid>` of -1 shows the number of tickets held by all of the user's processes, the default. Root can view information on other users by specifying the `<uid>`, or -1 for all users. `-x` shows additional information.

`set_funding [-u<uid>] -t<base_tickets>` — Root sets the number of `<base_tickets>` that funds `<uid>`'s processes. The default `<uid>` is the user ID running the program.

`show_funding [-u<uid>]` — Shows the number of base tickets that fund `<uid>`. The default `<uid>` is the user ID running the program. If `<uid>` is -1, all users are shown. Only root can view another `<uid>`.

`force_sched -p<pid> -m|-l|-n <base_tickets>` — Enables a user to control the "at most/at least" `nice` emulation without explicitly making the *setpriority()* system call used by `nice` and `renice`. `-p` chooses a process. `-m` means schedule the process with at most `<base_tickets>`. `-l` means schedule the process with at least `<base_tickets>` (restricted to root). `-n` means use standard lottery scheduling.

`lott_chuser -p<pid> -u<uid>` — Takes `<pid>` and puts it under `<uid>`'s currency. Useful for moving processes like X which run as root under the currency of the user using the process. This command is restricted to root.

The following programs are available only for lottery scheduling development.

`lott_ctl -a <on|off>` — Root-only command that turns on or off the abbreviated quanta feature. Future controllable features will be turned on and off via this command.

`lott_stat [-i<on|off>] [-l<on|off>] [-c<on|off>] [-p<pid>]` — Turns on or off profiling information gathering and printing. `-i` controls statistical information printed out to the system console in real-time. `-l` controls recording of process scheduling data retrievable via `lott_rinfo`. `-c` controls recording of cycle count performance data for scheduling functions, again retrievable via `lott_rinfo`. The defaults for these options are off. `-p` makes the output reflect only the chosen process, -1 for all processes, the default.

`lott_rinfo [-p<on|off>] [-i<on|off>] [-q<on|off>] [-c<on|off>] [-s<on|off>] -n<num_entries>` — Outputs data concerning the next `<num_entries>` scheduling events as described in section 4.1. `-p` controls reporting of process information. `-i` controls reporting of when the kernel is idling. `-q` controls reporting of when time slices expire. `-c` controls reporting of cycle count performance data. `-s` alters the output to be machine readable for statistical analysis. All of the previous options are `on` by default except for `-s`.

5 Evaluation

The variety of potential workloads makes evaluating scheduler performance difficult. In general, however, our day-to-day experience matches the results that we obtain from the benchmarks in this section. Exceptions are explored in section 6.

All experimental results were obtained from `partita`, David Petrou's personal machine. This machine has 64 MB

of main memory and a 200 MHz AMD K6 (pentium compatible) processor. Unless otherwise noted, no tests caused the machine to page. In the following figures error bars represent 95% confidence intervals.

We first show that the overhead incurred by our lottery scheduler is minimal, and then we demonstrate the proportional-share resource management properties that we gain with the lottery scheduler.

5.1 Overhead

We divide our overhead measurements in two parts: (1) microbenchmarks that quantify the differences in time it takes to execute scheduling operations under both the FreeBSD scheduler and the lottery scheduler kernels, and (2) macrobenchmarks that determine whether these differences are visible when running applications.

5.1.1 Microbenchmarks

We measure scheduling code fragments to quantify time spent in scheduler overhead. We use the the cycle counter instruction `RDTSC` (Read Time-Stamp Counter) which increments a counter every clock cycle [Int 1996] to obtain accurate measurements. As we ran our tests on a 200 MHz machine, divide "cycles" by 200 to obtain microseconds.

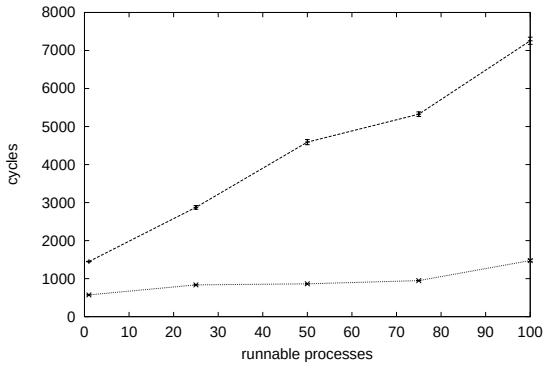

Figure 1: This figure shows the average number of cycles (out of at least 1000 measurements) to perform a context switch via the *cpu_switch()* function while varying the number of runnable processes. The top curve represents lottery scheduling while the bottom represents the FreeBSD scheduler.

The two most common scheduling operations in both the FreeBSD and our lottery schedulers are *cpu_switch()* and *setrunqueue()*. Under the FreeBSD scheduler, *cpu_switch()* makes a scheduling decision and performs a context switch. In our lottery scheduler, *cpu_switch()* calls *lott_choose_next_runner()* to make a scheduling decision and performs a context switch. Figure 1 shows the number of cycles it takes to run *cpu_switch()* while varying the number of processes[7]. Our lottery scheduling algorithm is $O(n)$ in the number of runnable processes while the FreeBSD scheduler is O(1). This difference in algorithmic complexity is visible in these results. Figure 2 shows the number of cycles to execute

[7] Naturally, we include the cycles in *lott_choose_next_runner()* in these measurements.

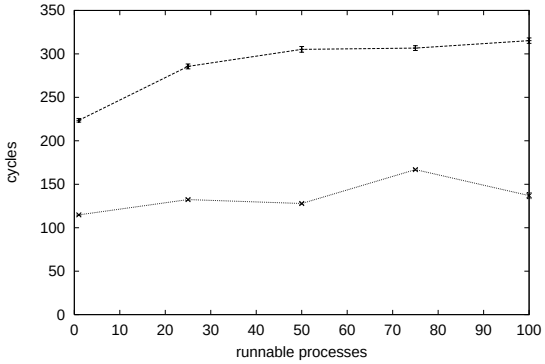

Figure 2: This figure shows the average number of cycles (out of at least 1000 measurements) to mark a process runnable via the *setrunqueue()* function while varying the number of runnable processes. The top curve represents lottery scheduling while the bottom represents the FreeBSD scheduler.

setrunqueue(), which makes a process runnable. This function is short and O(1) in both schedulers. We believe that the FreeBSD version is faster because it is written in assembly. We do not know why the FreeBSD curve fluctuates. With small standard errors, those fluctuations are statistically significant. Table 1 presents this data in numerical format.

5.1.2 Macrobenchmarks

The previous section uncovered measurable differences between the FreeBSD scheduler and our lottery scheduler. In this section we determine how visible these differences are on a larger scale. We measure two classes of programs: (1) interactive programs, and (2) batch programs. Our quality of service metric for interactive applications is response time while we consider throughput for batch applications.

We examine interactive applications first. We wrote a benchmark called `interactive` that continually goes to sleep for the shortest time possible and measures the time between going to sleep and the time that it receives the processor when it wakes up. While we run `interactive`, we also run a CPU-bound process in the background. Again, we use the `RDTSC` cycle count instruction. We keep in mind that the minimum latency that humans can discern varies between 50–150 ms depending on the individual [Shneiderman 1992]. Figure 3 measures response time under the FreeBSD scheduler as a baseline. Due to the way timeouts are handled in FreeBSD, processes cannot sleep on a timer event for less than 20 ms. Figure 4 shows the same experiment under the lottery scheduler. We achieve very similar results. We were curious about the utility of abbreviated quanta, so we disabled this feature within our lottery scheduler and reran this test. Our poor results are presented in figure 5. Table 2 presents the data in these figures numerically.

We now consider throughput of batch processes. Our test application is `rc564` [rc564 1997], a program that tries to find the solution to RSA's 64-bit secret-key challenge. To exacerbate the affect of our added overhead while running `rc564`, we also run a varying number of `interactive` processes. These `interactive` processes, due to abbreviated

		FreeBSD Scheduler mean/std. err.	Lottery Scheduler mean/std. err.
1 process	*cpu_switch()*	572.54/2.17	1449.13/3.74
	setrunqueue()	114.79/0.51	223.62/1.06
25 processes	*cpu_switch()*	835.90/2.35	2874.79/24.76
	setrunqueue()	132.40/0.67	285.59/56.73
50 processes	*cpu_switch()*	863.11/2.46	4589.64/34.30
	setrunqueue()	127.83/0.74	305.19/1.70
75 processes	*cpu_switch()*	948.68/2.84	5325.49/30.55
	setrunqueue()	166.88/0.53	306.70/1.40
100 processes	*cpu_switch()*	1474.50/13.14	7255.03/48.29
	setrunqueue()	137.00/1.78	315.27/1.58

Table 1: This table presents the data from figures 1 and 2 in numerical format. The values are in number of cycles.

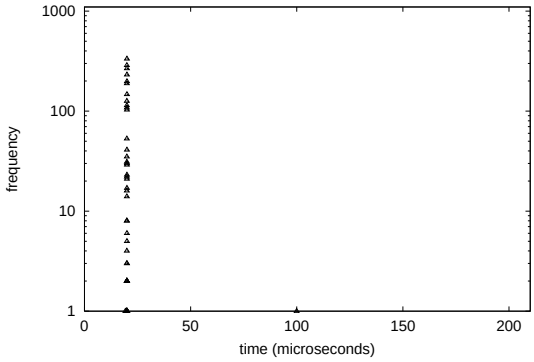

Figure 3: This histogram shows the response time distribution of `interactive` under the FreeBSD scheduler while a CPU-bound process runs. Almost all times are clustered around 20 ms.

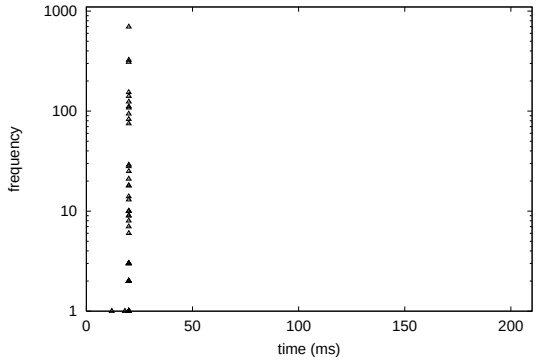

Figure 4: This histogram shows the response time distribution of `interactive` under the lottery scheduler while a CPU-bound process runs. As in the FreeBSD scheduler test, almost all times are clustered around 20 ms.

	mean/std. dev.
FreeBSD Scheduler	20.02/1.60
Lottery Scheduler	20.03/2.00
Lottery (no a.q.)	100.39/7.73

Table 2: This table presents the data from figures 3, 4, and 5 in numerical format. The times are in milliseconds. Over 2000 numbers were gathered for each case.

quanta, increase the number of context switch operations as shown in figure 6. The throughput of `rc564` versus the number of `interactive` processes is shown in figure 7. We note that as more `interactive` processes are run, the performance of `rc564` under the FreeBSD scheduler and the lottery scheduler worsens and diverges. In all runs, `rc564` under the lottery scheduler is less than one percent slower than under the FreeBSD scheduler.

As the previous experiment did not show a large difference between the FreeBSD and lottery scheduler, we ran one last macrobenchmark. In figure 8 we show the progress of a program called `count`, which simply loops and maintains a counter of how many loops it made, while running 100 `interactive` processes. These `interactive` processes pushed the number of context switches per second up to 5160 averaged over the run. Again, we achieve performance within one percent of the FreeBSD scheduler. One potential criticism is that there are systems that context switch more than 5160 times per second which may cause our scheduling overhead to be more apparent. However, we measured the average number of context switches per second over a 30 second interval on `wcarchive`, the world's largest and busiest FTP site,[8] to be 2589.

5.2 Flexible Execution Rate Control

The previous section demonstrated that the overhead of our lottery scheduler is negligible for the tests that we ran. Now we demonstrate the features that we have gained by replacing FreeBSD's decay usage scheduler with lottery scheduling. These are simple measurements and we refer the reader to [Waldspurger 1995] for an extensive analysis of lottery scheduling.

We demonstrate the ease at which a user can control the execution rate of his programs in figure 9. This figure shows three processes assigned tickets in a 3:2:1 ratio. Their

[8] `wcarchive` supports up to and often reaches 2750 simultaneous connections and stores 142 GB on-line. `wcarchive` is located at ftp://ftp.cdrom.com/.

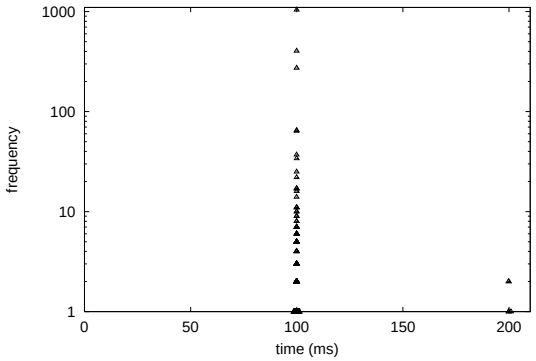

Figure 5: This histogram shows the response time distribution of `interactive` under the lottery scheduler without our abbreviated quanta extension while a CPU-bound process runs. The process now must wait a full time quantum, 100 ms, before running.

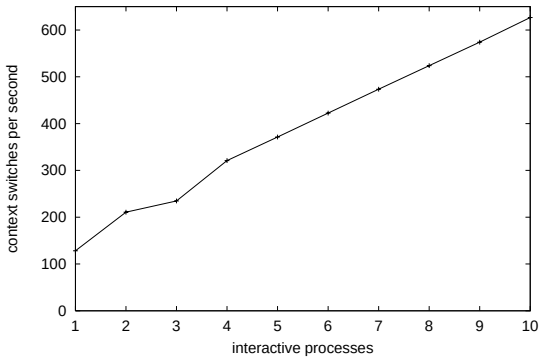

Figure 6: This figure shows the effect of increasing the number of `interactive` processes on the number of context switches per second.

progress is plotted along with ideal lines. We were curious how hard this was to reproduce using the FreeBSD scheduler. We achieve something close in figure 10. However, the `nice` values that we discovered, 10, 5, and 0, are hardly intuitively mappable to our goal of 3:2:1. The curves are also somewhat asymptotic and may diverge over time. Further, while lottery scheduling maintains the 3:2:1 ratio irrespective of system load, the FreeBSD scheduler unpredictably schedules these processes if the system is otherwise loaded.

We demonstrate user workload insulation in figure 11. Despite one user running two CPU-bound processes, the second user is able to receive twice the throughput from his one CPU-bound process. We also demonstrate user workload insulation with respect to processes that consume CPU and I/O. Figure 3 shows the time it takes to compile a program under an unloaded FreeBSD scheduler, under the FreeBSD scheduler with another user running 10 CPU-bound processes, and under the lottery scheduler with the same 10 process load. With lottery scheduling, the compiling user makes

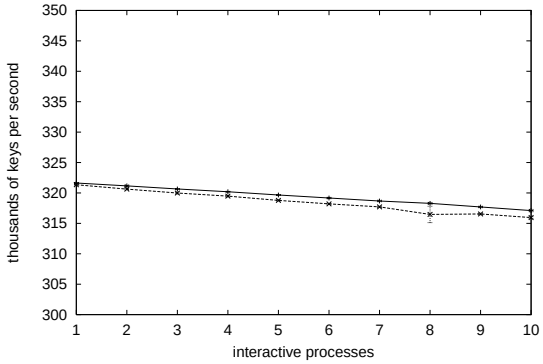

Figure 7: This figure shows the number of keys tried by `rc564` per second while varying the number of interactive processes. The top curve represents throughput under the FreeBSD scheduler while the bottom represents the lottery scheduler. Notice that the scale on the y-axis does not begin at zero. The performance of the system under the lottery scheduler is within 1% of the FreeBSD scheduler.

progress much faster than under the FreeBSD scheduler[9]. One may doubt the likelihood of a system with 10 runnable processes. A simple count showed 167 runnable processes (out of 2820 total processes) on `wcarchive`.

	mean/std. err.
Idle FreeBSD Scheduler	4.95/0.04
Loaded FreeBSD Scheduler	48.55/1.52
Loaded Lottery Scheduler	16.11/0.19

Table 3: This table shows the average time in seconds of 10 trials to compile and link a 2275 line program. We run this test under an unloaded FreeBSD scheduler, the FreeBSD scheduler while another user runs 10 CPU-bound jobs, and the lottery scheduler while another user runs 10 CPU-bound jobs.

6 Experience

Here we discuss our experience in using the lottery scheduler on two production machines and one personal machine.

6.1 `soda` and `meeko`

We have deployed our lottery scheduler on two production machines, `soda.csua.berkeley.edu` and `meeko.eecs.berkeley.edu`. `soda` is the central machine for the Computer Science Undergraduate Association at UC Berkeley. `soda` has 2300 accounts and often has over one hundred users logged on. Common users activity include participation in a chat room and code development. `soda` also manages 973 mailing lists besides serving mail for its users. `meeko` belongs to the FreeBSD Users' Group at UC Berkeley. `meeko` offers WWW service and mirrors part of `wcarchive` on its FTP

[9] We expected the compile under lottery scheduling to run faster than shown. We are investigating what our bottleneck is.

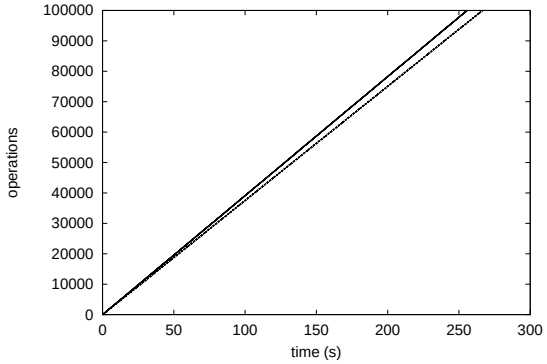

Figure 8: We run `interactive` 100 times and plot the progress of `count`. The FreeBSD scheduler is represented by the curve with the steeper slope while the second line represents progress under the lottery scheduler. The `interactive` processes caused 5160 context switches on average, well above what we experience on the busiest systems. Lottery scheduling is still less than 1% slower in this case.

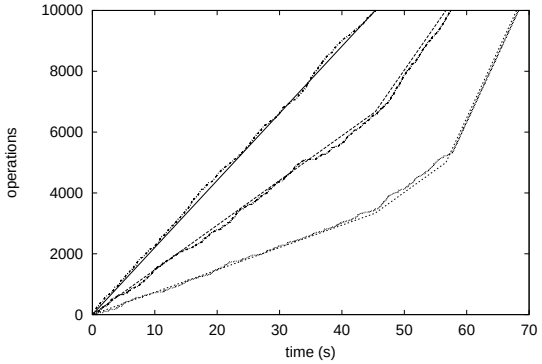

Figure 9: This figures demonstrates the progress of three `count` processes under lottery scheduling. The curves from top to bottom have 30, 20, and 10 tickets respectively. Also shows are straight lines representing ideal processor utilization. Notice that when processes finish, the remaining processes execute faster.

server. In addition, `meeko` exports a filesystem via NFS. There are usually 5 users logged into `meeko` actively developing code. That these systems have been running and in heavy use for weeks and that we have received no performance complaints from users is a testament to our code's stability and performance. Users report no lag related to the scheduler showing that the compensation ticket mechanism coupled with abbreviated quanta and kernel priorities are sufficient to provide good responsiveness.

We determine that load insulation works on our deployed code by looking at the output of the UNIX `top` utility while two users run the CPU-bound processes `xoopic` and `rc564`. `xoopic` [xoopic 1997] is a particle-in-cell plasma simulation that calculates particle positions and velocities by discretizing Maxwell's equations in time and space on a 2-D mesh. Tables 4 and 5 show the results.

PID	USERNAME	PRI	NICE	SIZE	RES	STAT	TIME	WCPU	CPU	COMMND
555	jwm	92	0	808K	164K	RUN	0:17	16.34%	16.25%	rc564
553	peterm	90	0	7392K	8012K	RUN	0:18	16.28%	16.21%	xoopic
552	peterm	90	0	7392K	8012K	RUN	0:18	16.12%	16.06%	xoopic
550	peterm	90	0	7392K	7852K	RUN	0:18	16.12%	16.06%	xoopic
551	peterm	90	0	7392K	7864K	RUN	0:18	16.08%	16.02%	xoopic
554	peterm	89	0	7392K	8012K	RUN	0:18	16.05%	15.98%	xoopic

Table 4: This table shows output from `top` while two users are running one and five CPU-bound processes respectively under the FreeBSD scheduler. The lack of load insulation enables `peterm` to obtain an unfair percentage of the CPU.

We have experienced an interesting problem with our emulation of `nice` semantics on `meeko`. Memory was slightly overcommitted and we `nice`'d `rc564` and `interactive` processes to -20 (highest priority). At this point the performance of many processes plummeted and a lot of CPU time was spent in the kernel (system time). Our untested theory is that other processes were spending most of their time page faulting. Before one of these low priority processes is able to make progress in userland it is context switched out

PID	USERNAME	PRI	NICE	SIZE	RES	STAT	TIME	WCPU	CPU	COMMND
296	jwm	98	0	808K	392K	RUN	0:28	52.21%	48.71%	rc564
272	peterm	76	0	7392K	7544K	RUN	1:02	11.63%	11.63%	xoopic
275	peterm	65	0	7392K	7716K	RUN	0:57	9.61%	9.61%	xoopic
282	peterm	64	0	7392K	8032K	RUN	0:50	9.50%	9.50%	xoopic
274	peterm	55	0	7392K	7636K	RUN	0:57	7.90%	7.90%	xoopic
273	peterm	53	0	7392K	7600K	RUN	0:55	7.13%	7.13%	xoopic

Table 5: This table shows output from `top` while two users are running one and five CPU-bound processes respectively under the lottery scheduler. `jwm` is able to receive about 50% of the CPU despite having only one runnable process.

and stands the chance of having pages in its working set evicted. We currently map a nice value of -20 to 10000 base tickets, an order of magnitude higher than the default number of base tickets, 1000, that fund a user's currency. We believe that this mapping is too aggressive and the problem we observed may diminish if we map -20 to a lower number such as 5000 base tickets.

6.2 `partita`

`partita` is the only machine running our lottery scheduler that is also running the X window system. We have observed the following problem that we have not been able to address. We experience slight choppiness when holding a key down (30 keys/s) in `emacs` using X while also running a CPU-bound process in the background. After closely examining the output from our `lott_rinfo` scheduler profiler, we determined that our scheduler was functioning correctly, and that `emacs` was running as soon as it had a keystroke, preempting the CPU-bound job. We also found no choppiness when running `emacs` outside of X. From the following series of events, we see that it is X that is not running as often as it should, causing the perceived choppiness: First X is running and `emacs` is asleep waiting for a keystroke. We press a key. Since we implement abbreviated quanta, and since `emacs` holds a kernel priority, `emacs` preempts X and runs immediately. Now `emacs` goes to sleep waiting for

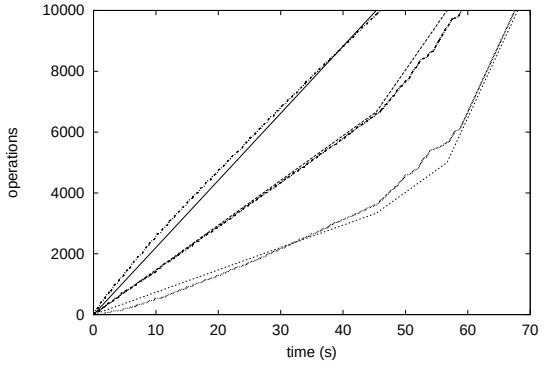

Figure 10: This figures demonstrates the progress of three `count` processes under the FreeBSD scheduler. We sought to emulate figure 9 and ran the programs from top to bottom with `nice 0`, `nice +5`, and `nice +10` respectively. Also shows are straight lines representing ideal processor utilization.

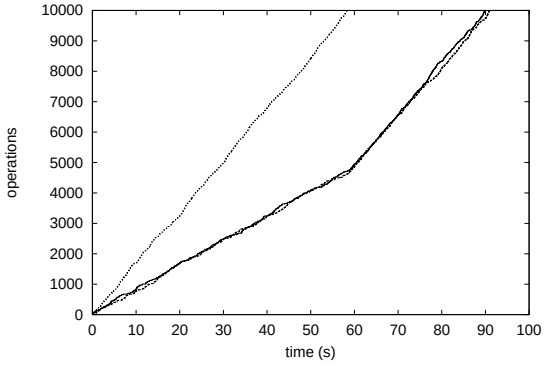

Figure 11: This figure shows the progress of three `count` processes under lottery scheduling. The top curve represents a `count` process run by one user while the bottom curves represent two `count` processes run by another user. When the first `count` finishes, the other `count` processes have made 47% and 46% progress toward completion (50% is ideal).

the next keystroke. Up to this point, both the FreeBSD scheduler and the lottery scheduler behave the same. Both X and the CPU-bound job are runnable and as both processes were preempted running userlevel code, neither have a kernel priority. X probably has more compensation tickets than the CPU-bound job, making it more likely to be chosen to run, but occasionally, the CPU-bound job will be run instead, resulting in choppiness. In the FreeBSD scheduler, the CPU-bound job runs often enough during the times when both X and `emacs` are sleeping to attain a lower priority. In this case, X always holds a higher priority than the CPU-bound job and is always chosen to run first. We can virtually eliminate this choppiness under the lottery scheduler by assigning the X window process 100 tickets, ten times more than the default of 10 tickets. We are trying to determine how common this problem occurs while investigating more general solutions.

7 Future Work

Besides the X anomaly described in the previous section, our work raises many issues that we leave for future work. We describe some of these in turn.

The ticket transfer mechanism is one aspect of lottery scheduling that we have not implemented. If one process desires a result that another process computes, the first process can "loan" its own CPU consumption rate to the second process. Now this second process will use CPU time at the aggregate rate initially granted to both processes. When the result is computed, it sends the result to the first process along with the right to consume CPU at the certain rate "borrowed" from the first process.

For example, a process blocked on interprocess communication can loan its tickets the server processes. Clients and servers can be explicitly written to make these transfers, or we can try to provide this functionality automatically. We notice that most IPC take the form of *read()* and *write()* system calls. In many cases we can determine which processes are communicating and automatically transfer tickets from the reader (the client blocked on a result) to the future writer (the server computing the result), and back again after the write completes. Such transfers can alleviate priority inversion among userlevel code.

In a distributed setting, an RPC layer can be written to transfer tickets among clients and servers. Consider a busy web server. Important clients can pay money for a certain number of tickets. These tickets are transmitted to web servers along with a URL in an unforgeable capability. The web server then schedules responses to queries based on the number of tickets that web clients hold.

There are some client/server scenarios on a single machine in which it is not clear that ticket transfers will work. Consider the printer command `lpr` and the printer daemon `lpd` owned by root. A user executes `lpr` which terminates and then at some time in the future, `lpd` executes on that user's behalf. This violates load insulation as the root-owned process is working on behalf of a user from its own tickets. There is no clear way to transfer tickets from the user to root because `lpr` has terminated before `lpd` runs. Similar arguments can be made for other system daemons such as `sendmail`.

A more difficult problem concerns scheduling diverse resources. For example, consider the `pagedaemon` process, a root-owned process that moves pages between memory and the swap device on behalf of all users. A process that is paging is not penalized for the time that the `pagedaemon` is executing on its behalf. On one hand, we would like to reduce the rate at which large processes page memory on the assumption that this will free up disk-bandwidth for other processes. Unfortunately, this may simply cause the large process to page even more as it doesn't have a chance to use its pages before they are evicted.

[Waldspurger & Weihl 1994] explains that proportional-share resource management can be applied across diverse resources but does not provide an algorithm or policy by which an entire system can be scheduled toward optimal system performance. [Hauser et al. 1993, Nieh et al. 1994] suggest that adjusting priorities (tickets) toward higher-level scheduling goals is very difficult or intractable. We find the pursuit of more intelligent system-wide resource schedulers

an exciting research question. We envision a framework by which a process registers its scheduling goals and the operating system schedules system resources to the process based on ticket-like objects. A cost-benefit analysis such as in [Patterson et al. 1995] considers the types of resources requested and how other processes may be affected by having those resources revoked.

8 Conclusion

We began this work with the goal of discovering why lottery scheduling, and by extension proportional-share scheduling, is not a standard part of modern operating systems. Our initial implementation followed [Waldspurger & Weihl 1994] and enabled control over process execution rates and processor load insulation at the cost of system responsiveness. After examining the FreeBSD scheduler, we decided to apply both abbreviated quanta and kernel priorities to our lottery scheduler. These techniques have been applied without altering the proportional-resource management semantics. In addition, user feedback prompted us to add support for the UNIX `nice` utility. Our measurements show that our optimized scheduler incurs more overhead than the FreeBSD scheduler, but that these differences are negligible even under large workloads. We achieve throughput and responsiveness nearly equal to FreeBSD except for one case concerning the X window system. Our remaining benchmarks indicate that we do achieve flexible control over the rate at which processes consume CPU. Our lottery scheduler was deployed to two production machines where we observe similar results. Our experience spurred many issues that we wish to explore in future work. This paper demonstrates that lottery scheduling is a viable process scheduler for the workloads we have tested. Our findings warrant further investigation into incorporating lottery scheduling as a standard part of operating system kernels.

9 Availability

Our lottery scheduler is available from http://www.cs.cmu.edu/~dpetrou/freebsd_lottery.tar.gz. Included are two new kernel source files, a context diff for applying patches to 14 existing kernel files, and the source for 10 userlevel programs that interact with the scheduler.

10 Acknowledgments

We thank UC Berkeley's Computer Science Undergraduate Association and FreeBSD Users' Group for permitting us to deploy our experimental kernel on their production machines. Peter Mardahl kindly offered `xoopic` for some of our benchmarks. We thank Aaron Smith for convincing us that emulating `nice` semantics was necessary. Thanks also go to David Greenman for providing us with statistics on `wcarchive`.

References

[4.4BSD 1994] The 4.4BSD source code, 1994. See ftp://ftp.cdrom.com/pub/bsd-sources.

[Birrell 1989] Birrell, A. D. An introduction to programming with threads. Technical Report 35, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, January 1989.

[FreeBSD 1997] The FreeBSD Operating System, 1997. See http://www.freebsd.org/.

[Goldberg et al. 1996] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[Hauser et al. 1993] Hauser, C., Jacobi, C., Theimer, M., Welch, B., and Weiser, M. Using threads in interactive systems: A case study. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 94–105, December 5–8 1993.

[Hellerstein 1993] Hellerstein, J. L. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[Int 1996] Intel. *Pentium Pro Family Developer's Manual*, volume 2, 1996.

[Jolitz & Jolitz 1996] Jolitz, W. F. and Jolitz, L. G. *Source Code Secrets: The Basic Kernel*, volume 1. Peer-to-Peer Communications, Inc., 1996.

[Lampson & Redell 1980] Lampson, B. W. and Redell, D. D. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[Lampson 1984] Lampson, B. W. Hints for Computer System Design. *IEEE Software*, 1(1):11–28, January 1984.

[Lehey 1996] Lehey, G. *The Complete FreeBSD*. Walnut Creek, September 1996.

[Massalin & Pu 1989] Massalin, H. and Pu, C. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, volume 23, pp. 191–201, December 1989.

[McKusick et al. 1996] McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc., 1996.

[Nieh et al. 1994] Nieh, J., Hanko, J. G., Northcutt, J. D., and Wall, G. A. SVR4 UNIX scheduler unacceptable for multimedia applications. *Lecture Notes in Computer Science*, 846, 1994.

[Parnas 1972] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Patterson et al. 1995] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 79–95, December 3–6 1995.

[rc564 1997] Project RC5, 1997. See http://www.distributed.net/rc5/.

[Ritchie & Thompson 1974] Ritchie, D. M. and Thompson, K. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[Shneiderman 1992] Shneiderman, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing Co., Reading, MA, second edition, 1992.

[Vahalia 1996] Vahalia, U. *UNIX Internals: The New Frontiers*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[Waldspurger & Weihl 1994] Waldspurger, C. A. and Weihl, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Mangement. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pp. 1–11, November 14–17 1994.

[Waldspurger & Weihl 1996] Waldspurger, C. A. and Weihl, W. E. An object-oriented framework for modular resource management. In *Fifth Workshop on Object-Orientation in Operating Systems (IWOOOS '96), October, 1996. Seattle, WA*, 1996.

[Waldspurger 1995] Waldspurger, C. A. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD dissertation, Massachusetts Institute of Technology, September 1995.

[xoopic 1997] XOOPIC Plasma Simulation Program, 1997. See http://ptsg.eecs.berkeley.edu/xoopic/xoopic.html.

Fault Tolerance in an Eventually-Serializable Data Service

Oleg Cheiner* Istvan Derenyi†

January 30, 1998

1 Introduction

Replication is used in distributed systems to improve availability and to increase throughput. The disadvantage of replication is the additional effort required to maintain consistency among replicas when serializing operations submitted by clients. Several notions of consistency have been defined. The strongest notion of consistency is *atomicity*, in which replicas emulate a single centralized object. Methods to achieve atomicity include write-all/read-one [4], primary copy [5, 6, 7], majority consensus [8], and quorum consensus [9, 10]. Achieving atomicity often has a high cost; some applications, such as directory services, are willing to tolerate some transient inconsistencies to avoid paying this cost. This gives rise to different notions of consistency. *Sequential consistency* [11], guaranteed by systems such as Orca [12], allows operations to be reordered as long as they remain consistent with the view of isolated clients. Other systems provide even weaker guarantees to the clients [13, 14, 15] to get better performance.

Fekete et al. [1] defined a highly available eventually-serializable data service (ESDS). They specified general conditions for such a service, and presented an algorithm based on *lazy replication*, in which operations received by each replica are *gossiped* in the background. Responses to operations may be out-of-date, not reflecting the effects of operations that have not yet been received by a given replica. The definition of ESDS includes a formal specification of the data service and an abstract distributed algorithm that implements the service.

ESDS relaxes consistency guarantees provided by serializable distributed data services to improve system efficiency and availability. An important consideration in the design of ESDS was that it could be employed in building real systems. Cheiner recently developed a distributed experimental prototype of the ESDS system [2]. Empirical tests on the implementation showed that ESDS scales at least up to 20 replicas and can exploit parallel replication to achieve better throughput. The tests also showed how ESDS exposes a tradeoff between performance and consistency to its users. The tradeoff balance can be shifted toward consistency and away from performance (and vice versa) by varying consistency restrictions on the operations submitted to the system.

2 Our Contributions

The ESDS implementation in [2] does not tolerate replica failures. However, ESDS lends itself to a fault-tolerant distributed implementation due to the redundancy inherent in data replication. We extend the implementation in [2] to tolerate replica faults, while preserving guarantees of eventual consistency and the performance advantages of replication. Our modifications deal with two models

*Carnegie Mellon University, oleg@cs.cmu.edu
†Carnegie Mellon University, derenyi@cs.cmu.edu

of replica failure. In the *fail-stop* model, replica nodes may *crash*. Crashed replicas do not respond to user requests or send gossip messages to other replicas. In the *fail-stop-restart* model, the replicas are allowed to rejoin the system after a crash and restart.

Our report specifies the semantics of the failure models and the desired behavior of working replicas in the face of failures.

To compensate for the lack of immediate consistency guarantees, ESDS provides a mechanism for specifying dependencies between operations explicitly. The abstract algorithm in [1] uses *prev* sets to identify the dependencies. Each operation x submitted to the system includes a *prev* set of operations that must be applied to the state before x. However, it is impractical to require users of an ESDS-based system to specify large dependency sets. Furthermore, *prev* sets are inefficient. A *prev* set may include any operations that have been previously submitted to the system, and therefore the upper bound on the size of *prev* sets grows linearly with the number of operations submitted to the system. The time it takes to verify that an operation's prev set has been satisfied, the memory required to store an operation, and the cost of gossiping an operation will all grow linearly with the number of operations as well. In addition, the system is not able to take advantage of stabilization of old operations and discard their identifiers because the identifiers may later appear in a new operation's *prev* set.

Our changes to the ESDS implementation remove the inefficiencies that result from using *prev* sets. We substitute the *multipart timestamp* technique in place of *prev* sets to keep track of system dependencies. The multipart timestamp technique is based on Lamport's logical clocks [16]. The approach is similar to the multipart timestamp implementation in [18].

3 ESDS Overview

This section is an overview of the work upon which our project is based. Section 3.1 describes the abstract algorithm from [1]. Section 3.2 briefly summarizes the prototype implementation from [2].

3.1 The Algorithm

ESDS is a distributed data service based on the *lazy replication* model. The service maintains copies of its state at multiple distributed *replicas*. A typical operation is executed by a single replica, which immediately returns an answer to the user. Replicas update each other by "lazy" exchange of *gossip* messages in the background. Thus, ESDS relaxes consistency guarantees provided by more traditional (serializable) distributed data services in return for lower latency and higher throughput.

The nodes participating in an execution of ESDS consist of a fixed number of replicas and a set of client *front ends*. Front ends interact with the replicas on behalf of users. They submit user requests to the replicas, attempting to balance the load on the replicas and obtain an answer with the smallest possible latency. The front ends may submit an operation more than once in order to find a closer replica, or to make progress in the event that some of the replicas are down.

For every operation submitted to the system the front ends must generate an identifier *uid* that is globally unique across the system. In addition, an operation has several attributes that specify its semantics. First, the operation defines the transformation to be applied to the data object and the value to be returned to the user. Second, the operation may be *causally dependent* on other operations previously processed by the system [18]. To capture this dependence, the operation's state contains a *prev* component, a set of operation *uid*s that must be executed before it.

Despite allowing transient inconsistency among the replicas, ESDS provides provable guarantees that in the limit an *eventual total order* is established on all operations, and that the total order is identical at all replicas. For some operations it may be desirable to disallow even transient inconsistency. An operation is defined to be *stable* if the prefix of the eventual total order up to that operation is known at every replica. The last state component of an operation is a *strict* boolean flag. By setting the strict flag, the client can force the replicas to stabilize the operation before returning an answer for it. Thus, if all operations are strict, the data service becomes atomic.

Each replica r maintains several state components to support lazy replication and guarantee eventual serialization:

- $done_r[i]$, $i \in [1..n]$ (n is the number of replicas)
- $solid_r[i]$, $i \in [1..n]$
- $minlabel_r$.

For all i, $done_r[i]$ is the set of *uid*s of operations that replica r knows are "done" at replica i. An operation is *done* at replica i if its value can be computed at that replica. To preserve dependencies specified by an operation's *prev* set, a replica does not enter the operation into its *done* set until all operations in the *prev* are done at the replica.

$Solid_r[i]$ is likewise a set of operation *uid*s. The interpretation of $x \in solid_r[i]$ is that replica r knows that replica i knows that x is done at every replica. When an operation is in $solid_r[i]$ for all i, it has stabilized. A replica sends answers for strict operations only after they have entered all *solid* sets at that replica. $Minlabel_r$ is a function that assigns a globally unique label to each done operation. The labels in the range of $minlabel_r$ are totally ordered. Thus, $minlabel_r$ specifies the order in which the operations have been done. To compute a value for a done operation, replicas first apply all preceding operations in the $minlabel_r$ order to the initial state, and then apply the operation for which they are computing a value. As an optimization, replicas can keep a copy of the *stable state* - the result of applying all stable operations to the initial state. The value of a done operation that has not yet stabilized is computed by applying all unstable done operations to a copy of the stable state, without modifying the stable state itself.

Replicas update each other via gossip messages. A gossip message from replica r to replica r' includes $done_r[r]$, $solid_r[r]$, and $minlabel_r$. Upon receiving the message, r' updates its local *done* and *solid* sets for r and r' and merges $minlabel_{r'}$ with $minlabel_r$. In the absence of new operations from clients, after a few rounds of gossip replicas gravitate towards identical *done* and *solid* sets and *minlabel* orders.

For a more detailed description of the algorithm, the reader is referred to [1].

3.2 A Prototype Implementation

The original ESDS prototype [2] is a straight translation of the algorithm into a distributed program written in C++. The prototype uses an implementation of the MPI Standard [19] for communications between system nodes. The major design modules include replica nodes, front end nodes, and operations. The state of each C++ module is exactly as specified in the algorithm, except for some additional bookkeeping state.

The prototype defines an abstraction barrier between the ESDS layer and the data service application running on top of it. Th ESDS layer acts as a building block for constructing distributed data services. An ESDS-based application always includes three modules: (1) service state, (2) service operations, and (3) return values for service operations. Together these modules implement a non-distributed version of the data service. The job of the ESDS layer is to distribute the application service across multiple replicas according to the semantics of the ESDS algorithm. The three application modules are opaque to the ESDS layer, which makes calls to the modules to handle all application-specific computation (e.g. applying an operation to the application state and producing a return value).

4 Multipart Timestamps

We substituted a more efficient method for tracking causal dependencies between operations in place of *prev* sets. Our approach is based on a technique called *multipart timestamps*.

A multipart timestamp t is a n-tuple $(t_1, \ldots, t_n)$ of nonnegative integer counters. In the context of ESDS, n corresponds to the number of replicas. A partial order is defined on multipart timestamps: $t \leq s$ iff $t_j \leq s_j$ for $j \in [1..n]$. Two multipart timestamps are *merged* by taking their component-wise maximum. Our design includes a module implementing this definition of multipart timestamps.

In this optimization we remove *prev* sets from operation state and redefine the protocol for keeping track of dependencies between operations using multipart timestamps.

In the new protocol the state of operation j includes two new multipart timestamp components, *prev-ts* and *op-ts* (*op-ts* is initially all zeros). Replica state also gets two multipart timestamp components, *val-ts* and *rep-ts*, both initially all zeros. The meanings of the new state components are as follows:

- *op-ts* is assigned to each new operation by the receiving replica in the manner described below. *Op-ts* is guaranteed to be unique for each operation.
- *prev-ts* plays the same role that the *prev* set played in the unoptimized version. It specifies that any other operation with an *op-ts* smaller than this operation's *prev-ts* must be done before this operation. In other words, for each pair of operations i and j, $j.op\text{-}ts < i.prev\text{-}ts \Rightarrow j$ is in i's *prev* set.
- *val-ts* is the merge of *op-ts* timestamps of all operations done at the replica.
- *rep-ts* is the current replica timestamp, used to assign values to *op-ts* of newly submitted operations in the protocol below.

The protocol works as follows:

1. Every time replica r receives a new operation i from a frontend, it increments *r.rep-ts[r]*, assigns $r.rep\text{-}ts$ to $i.op\text{-}ts$, and sends the value of $i.op\text{-}ts$ to the frontend. The frontend then forwards $i.op\text{-}ts$ to the client.
2. When a client wants to specify that operations $i_1, \ldots, i_k$ must precede operation j, it merges $i_1.op\text{-}ts, \ldots, i_k.op\text{-}ts$ and assigns the result to $j.prev\text{-}ts$.
3. When replica r does operation i (i.e. moves it into $done_r[r]$), it merges $i.op\text{-}ts$ into $r.val\text{-}ts$.
4. Gossip messages from replica r to replica r' contain $r.rep\text{-}ts$. Upon receipt of the gossip message, replica r' merges $r.rep\text{-}ts$ into $r'.rep\text{-}ts$. For all operations $i_k \in done_r[r]$ included in the gossip message, r' merges $i_k.op\text{-}ts$ into $r'.val\text{-}ts$.
5. When replica r wants to do operation i and needs to check that i's dependencies have been satisfied, it checks that $i.prev\text{-}ts \leq r.val\text{-}ts$.

Ladin et. al. [18] give a semiformal argument that this protocol never violates dependency specifications.

Figure 1 demonstrates timestamp-based constraints in action with a small example execution. Two replicas and one client participate; the frontends have been omitted to reduce clutter. The sequence of messages between participants is indicated by the numbers on the arrows.

The hypothetical data service's state is a string. Each operation appends its *uid* to the end of the string, and the resulting string is returned to the client. The client's intent is to submit two operations and make sure the first operation occurs before the second. The execution starts in Figure 1a with the client submitting operation 1 to replica 0, with $prev\text{-}ts = < 0, 0 >$. The client gets back $op\text{-}ts = < 1, 0 >$ for operation 1. The client now sends operation 2 to replica 1. The *prev-ts* of operation 2 is set to *op-ts* of operation 1 by the client, indicating that operation 1 is in operation 2's *prev* set[1]. Next, replica 1 checks $2.prev\text{-}ts$ against $1.val\text{-}ts$. Since $2.prev\text{-}ts \not\leq 1.val\text{-}ts$

[1] the message from replica 1 to the client with operation 2's *op-ts* is omitted from the figure

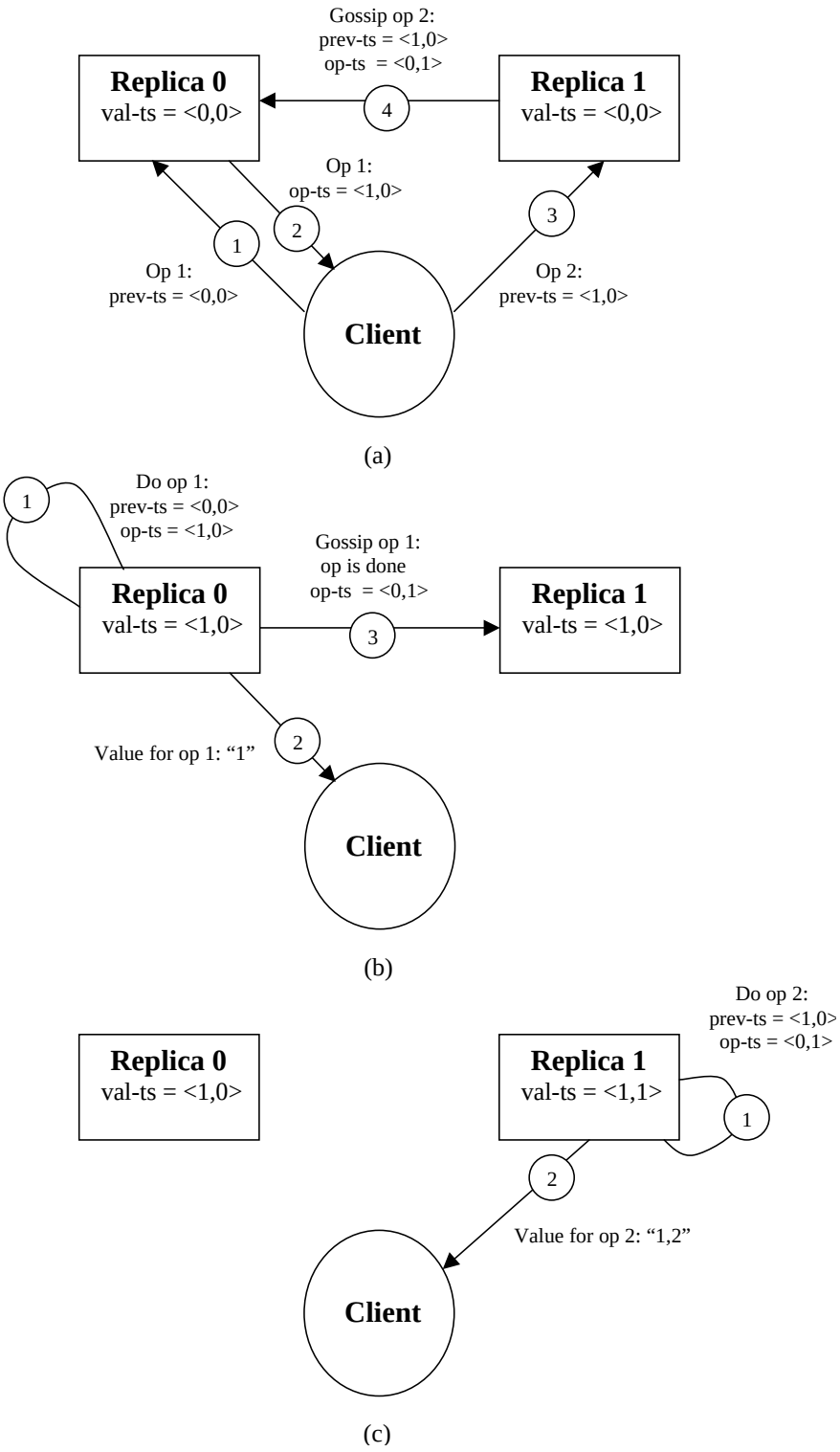

Figure 1: Multipart Timestamp Example

(i.e. operation 2's *prev-ts* is not smaller than replica 1's *val-ts*), replica 1 knows that operation 2's dependencies have not been satisfied, so it is not possible to do the operation at this point. Instead replica 1 sends a gossip message to replica 0.

The execution continues in Figure 1b. Replica 0 now has two undone operations, but it cannot do operation 2 (for the same reason replica 1 could not do it). So replica 0 does operation 1 and sends the value to the user. It then gossips the new information about operation 0 to replica 1. Upon receiving this gossip message, replica 1 knows that operation 1 has been done and merges 1.*op-ts* into 1.*val-ts*. At this point 2.*prev-ts* $\leq$ 1.*val-ts*, so operation 2 is ready to be done. In Figure 1c replica 1 does operation 2 and returns the correct value to the client.

Remark: To complete the multipart timestamp implementation, it is necessary to take care of the case when a client submits an operation to more than one replica simultaneously and gets different *op-ts* values for the operation. This has been relegated to future work, as it is not essential to our goal of implementing a working timestamp prototype and measuring its advantages against the *prev* set approach.

5 Fault Tolerance

We describe the failure models and our implementation of fault tolerance in the presence of faults that conform to those models in this section.

5.1 Failure Models

In this project we consider two fault models: *fail-stop* and *fail-stop-restart*. In the fail-stop model the failed replica *crashes* and never comes back up. All the state of a crashed replica and the messages queued in its input channels is lost. In the fail-stop-restart model, crashed replicas are allowed to rejoin the computation after crashing.

We would like for the system's external behavior to remain unchanged (except perhaps in performance) in the presents of stopping failures. Clients should be able to continue submitting operations to the system and receive responses according to normal ESDS semantics. Failure recovery should also be transparent to the clients, except for possible performance degradation.

5.2 Fault Tolerance Design

In our protocol for dealing with stopping failures each replica periodically broadcasts *keep-alive* messages to other replicas, indicating that it is still functioning. To make our failure model more concrete and to simplify our design, we introduce two timing assumptions about *keep-alive* messages. The first assumption is that live replicas broadcast repeat *keep-alive* messages within a fixed time period, which we will call T_{alive}. Failure to issue a *keep-alive* broadcast within T_{alive} of the last such broadcast is equivalent to a stopping fault (i.e. replicas are assumed not to issue any other messages afterward). The second assumption is that latency of the channels between replicas is bound above by a constant T_l.

These assumptions allow us to use a simple fault tolerance protocol for fail-stop faults. Each replica r waits for a *keep-alive* message from other replicas that it considers live. If $T_{alive} + T_l$ time elapses without a keep alive message from replica r', r immediately puts r' in its set of crashed replicas and then continues execution as if r' never participated in the system. Our timing assumptions guarantee that when this happens, r' has indeed crashed (and was not merely delayed). In a short time after r declares r' crashed, other replicas will also declare r' crashed. This observation leads to an informal argument that this protocol is *safe* with respect to strict operations, as follows.

Suppose (without loss of generality) that replica r is the first to decide that replica r' has crashed. Since r has not received a *keep-alive* message from r' for $T_{alive} + T_l$ time units, by our timing assumptions no other replica can receive a message from r' after r decides that r' has crashed.

Therefore, after a finite period of time all other replicas will also decide that r' has crashed. In the meantime, in the absence of messages from r' the other replicas can exhibit only a subset of the behaviors that they could exhibit had they declared r' crashed simultaneously with r. So safety is preserved.

To add recovery to this mechanism, we require that each replica write to stable storage the location of all other replicas at the time of system invocation (so that it knows how to contact them after a crash and restart). Our recovery protocol for the fail-stop-restart model proceeds as follows. A replica r' restarting after a crash broadcasts a *begin-recovery* message to every other replica. Upon receipt of a *begin-recovery* message from r', replica r sends r' a special "recovery" gossip message. This message has the same structure and content as a regular gossip message, with one difference. A regular gossip message is an incremental update, including only the information that has changed since the previous gossip message between the same replicas. A recovery gossip message contains full (rather than incremental) information, as if this were the first time r gossiped to r'.

After r has sent the recovery gossip message to r', it considers r' as live and acts as if r' was always up. This is the essential step for preserving safety during recovery. It ensures that r and r' never give inconsistent responses to a strict operation[2]. Any strict operation that r replied to *before* sending the recovery gossip message to r' will immediately become stable at r' upon receipt of the recovery gossip message. *After* sending the recovery gossip message to r', r explicitly waits for any strict operation to stabilize at r' before replying to it.

The restarting replica r' now gathers recovery gossip messages from all other replicas. Since together this collection of recovery gossip messages contains all the information present in the system, r' reconstructs its entire volatile state to be consistent with other replicas without further communication. Once the reconstruction is complete, r' resumes normal operation immediately.

Two complications must be taken care of. First, a restarting replica cannot wait indefinitely for recovery gossip messages from all replicas, since some of them might be down. We assume that every live replica sends the recovery gossip message within T_{alive} of the arrival of the *begin-recovery* message. The restarting replica must wait for $T_{alive} + 2T_l$ time units after broadcasting *begin-recovery*, but it declares all replicas whose recovery gossip did not arrive within that time to be down. Second, more than one replica may be recovering simultaneously. So a restarting replica might receive a *begin-recovery* message from another restarting replica. If that happens, the simultaneously recovering replicas signal to each other not to wait for a recovery gossip message from them, but simply to consider each other live.

Remark: The system may lose information about an operation if the subset of replicas that knows about the operation goes down before gossiping about it to other replicas. To tolerate such occurrences, two different approaches may be employed. One approach is for individual replicas to write the information about operations known to them to stable storage, i.e. to implement transaction semantics for locally known operations. Another approach is for clients to re-submit the operation to other replicas after a timeout. The latter approach sacrifices client transparency, but it is easier to implement and can still preserve user transparency. We have not implemented these approaches.

6 ESDS Application: Bank Accounts

The original work on lazy replication and ESDS [1] suggests that directory and information services (and similar applications) are most suitable for ESDS because immediate consistency is not important to users of most such systems. We implemented a simplified application that keeps track of bank accounts to highlight the features of ESDS and show that it could potentially be useful for a wider variety of applications. Using the bank application, we can demonstrate the usage of strict and non-strict operations and timestamp-based dependencies.

[2]Recall that the prefix of a strict operation x, i.e. the operations applied to the data service state before x, must stabilize at every replica before any replica replies to x

The application maintains a customer-account database in a banking environment. Branches of the bank are located at physically different sites. The bank maintains a global database of customer accounts, implemented as an application layer on top of ESDS. At least one replica node resides at each branch. Operations submitted to a particular branch are forwarded to the local frontend. During normal operation, the frontend submits these operations to the local replica. However, if that replica happens to be down, the branch can continue to function by having the front-end submit the operations to remote replicas.

The database maintains a set of data tuples in the form of (NAME, AMOUNT). In addition of opening a new account and closing an established one, we implemented three basic operation which can be carried out on an account. The operations are listed below with the corresponding ESDS specifications:

1. *Withdrawal*: strict = TRUE, *prev-ts* = FULL
2. *Deposit*: strict = FALSE, *prev-ts* = EMPTY
3. *Balance*:
   - Local, Hurried: strict = FALSE, *prev-ts* = EMPTY
   - Local, Quick: strict = FALSE, *prev-ts* = LOCAL-FULL
   - Global, Prompt: strict = FALSE, *prev-ts* = FULL

Assume that the last operation submitted to replica r was received timestamp $(t_1, \ldots, t_n)$. By EMPTY, LOCAL-FULL, and FULL *prev-ts*s we mean the following:

- EMPTY = $(0, \ldots, 0)$
- LOCAL-FULL = $(0, \ldots, 0, t_r, 0, ..., 0)$
- FULL = $(t_1, \ldots, t_n)$

EMPTY prev set contains no operations, a LOCAL-FULL prev set includes all operations previously submitted at replica r (and no others), and a FULL prev set includes all previously submitted operations at all replicas that replica r knows about.

A *Deposit* operation always succeeds, and it is independent of its ordering relative to other operations on the same account. We implemented *Deposit* without any dependency constraints. On the other hand, a *Withdrawal* of the amount X can result in different answers to the client, depending on whether the account has sufficient funds. If it does not, the *Withdrawal* operation does not change the amount on the account and returns an error message. Otherwise, it decreases the amount in the account by X. Permitting two *Withdrawal* operations on the same account to occur concurrently at different replicas would allow the client to withdraw money she does not have. Therefore, we implemented *Withdrawal* as a strict operation.

It is up to the customer to determine what level of inconsistency she can tolerate in a *Balance* operation in exchange for lower latency. Using the Hurried option, there is no guarantee that previously submitted operations for the account will be visible by the *Balance* lookup. With Quick *Balance* lookup, all previously submitted operations at the local branch will be visible, but there is no guarantee with respect to operations submitted at other branches. Using Prompt *Balance* lookup, all operations known at the local replica will be visible, but there is no guarantee that deposit operations carried out at other branches and not yet gossiped to the local branch will be visible.

7 Evaluation

In this section we discuss our correctness testing strategy and present empirical measurements of the impact of our changes on ESDS performance. The figures referred to by this section are at the end of the document.

7.1 Evaluating Correctness

To test correctness of a given extension, we developed a simple database application with specific update operations. The application submits sequences of operations to ESDS and looks for errors in return values and the final state of the database. In case of correct behavior, the database reaches a well defined state, which is testable by the application. Incorrect behavior includes all types of out-of-order executions for strict operations, ignored dependency constraints, duplicate executions of operations, and unexecuted operations.

We modified the system slightly to allow the application some control over the distribution of operations to replicas. Such control is important if the application is to measure the impact of faults on performance accurately.

7.1.1 Multipart Timestamps

Since timestamps implement only dependency constraints between operations in ESDS, we made a working assumption that the switch from prev sets to timestamps did not disturb correctness of the prototype with respect to strict operations[3] Thus, the application tests correctness of the timestamp-based ESDS system with respect to non-strict operations only.

For this phase of testing, we set up executions with N clients and corresponding N replicas. For the purposes of testing we gave clients direct control over which replicas receive the operations.

The clients submit a continuous stream of M operations to their corresponding replica in rounds, with one operation per round per client (a total of $M * N$ operations is submitted). The state of the test application is a two dimensional array, where an element (m, n) corresponds to operation m submitted to replica n. The array is initialized to zeros, and each operation increments the value of its corresponding array element. Clients examine the distribution of executed operations by reading the array at a replica.

The *prev-ts* timestamp of operation m submitted by client n is $(m-1, m-1, ..., m-1)$. That is, each operation has all operations from previous rounds in its prev set. At each round, clients read back the current state of their replica and check for that the array contains a 1 at each position that corresponds to an operation from a previous round. Thus the clients can decide whether all prev set operations were carried out exactly once before the operation from the current round returned.

The final version of the time-stamp extension never failed the described test. We ran the test several times, with different number (1-6) replicas, and with up to 1000 operations at each replica.

7.1.2 Fault Tolerance

Correctness of the fault tolerance extension requires that in the presence of failures the system still maintains consistent state. In the fail-stop mode it means that after a replica died, the rest of the replicas remain consistent and give responses according to ESDS semantics. In the fail-stop-recovery mode, in addition to the above mentioned requirement, the restarting replica(s) should get into a consistent state with the functioning replicas. The recovering replica also should not give responses that are inconsistent with anything other replicas have done while the recovering replica was down. As the correctness requirement for the failure-stop mode is a subset of that of the failure-stop-recovery mode, we tested the correctness of the second model only. We used the newly implemented timestamp prev sets during testing.

After recovery of a replica has completed, We can force the functioning replicas to serialize by submitting a strict operation with a prev set that includes all previously submitted operations to the newly recovered replica. When this operation returns, we can check that all previously submitted operations should have applied at all replicas in the same order.

[3]Dependency constrains among operations are orthogonal to strictness in ESDS. This is reflected in the ESDS prototype design.

The general scenario of the testing is the following: a client submits random operations (strict, non-strict, with different prev sets) to all live replicas in turn. Meanwhile, at random intervals it sends KILL or RESTART messages to one or more randomly selected replicas. After sending a RESTART message, it immediately submits a strict operation with a prev set of all previously submitted operations to the restarted replica[4]. After this operation, all previous operations should be stabilized at all replicas, and the system should be in a consistent state. The client checks this by submitting operations to each live replica which do not modify the state, but simply return it.

In our original test application the client(s) could check whether each of a set of operations were carried out exactly once, but not the order in which they were carried out. To be able to test that the sequence of the prefix operations to a strict operation does not change at any point after the strict operation returns, we extended the application. In the new version the application state includes an additional field C, which keeps track of how many operations have already been carried out on the state. A new operation sets its corresponding field in the state equal to C if the field is currently zero, and equal to -1 otherwise.

With this modification, the client can check the global consistency of the system, i.e. that every replica applies all operations in an identical order. It is also necessary to verify that this order will be stable for the rest of the system execution. The client checks this by keeping a copy of the last consistent state, incrementally updating it at each correctness check step (after each RESTART message), and checking it against the current state of the replicas.

7.2 Evaluating Performance

This section presents performance results for our modifications to ESDS.

7.2.1 Multipart Timestamps vs. Prev Sets

This optimization improves the algorithm along three dimensions. By substituting constant-size timestamps for unbounded *prev* sets, it reduces memory requirements and message sizes. Furthermore, with this protocol the time to check dependencies for one operations is reduced from $\mathcal{O}(n)$ to $\mathcal{O}(1)$. We present empirical evidence for these claims in this section.

Using a slightly modified version of the test application, we measured response latency of the unoptimized and optimized versions of the system, using 1 Pentium II workstation running RedHat Linux 4.2. The measurement run submitted M operations for N (between 1 and 4) replicas, with operation m going to replica $m \ mod \ N$. We measured the latency of the operations experienced by the clients. Each operation m had a prev set of operations $(0 \ldots m-1)$[5].

The collected data confirmed our hypotheses. Figure 2 shows the measured latency for a set of 600 operations[6] submitted to a system consisting of one replica. We ran a set of three measurements with the original prev set and with the timestamp implementation each. From the graph we can see the performance advantage of the timestamp extension. The latency of operations in the timestamp implementation is constant, whereas in the original prev set implementation it is an increasing function of the number of submitted operations.

With more than one replica the response latency in the prev set implementation exhibits even worse characteristics, mainly due to the gossip overhead of prev sets. To demonstrate the behavior of systems with multiple replicas, we plotted the measurements for 2, 3, and 4 replicas in Figure 3 through 5. From the figures we can see that the performance of the original prev set implementation seriously degrades as the number of operations increases. The timestamp implementation outperforms the original prev set implementation in each scenario.

[4] By measuring the response time for this operation, we can effectively measure the length of recovery, as the operation from the newly restarted replica was not return before the replica finished its recovery period. The result of this measurement is shown on Figure 11

[5] To describe this dependency with timestamps, we modified the computation of the timestamp prev sets as follows: $prev\text{-}ts = (t_1, \ldots, x_N)$, where t_i = upper bound($\frac{m}{N}$) if $i < m$; t_i = lower bound($\frac{m}{N}$) otherwise.

[6] We limited the number of operations to 600 in latency measurements due to time constraints.

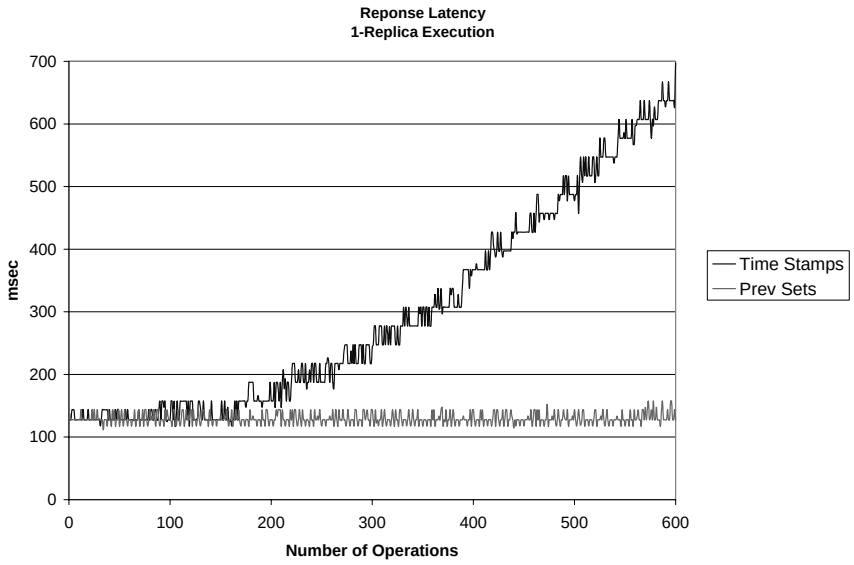

Figure 2: Response Latency - 1 Replica Execution

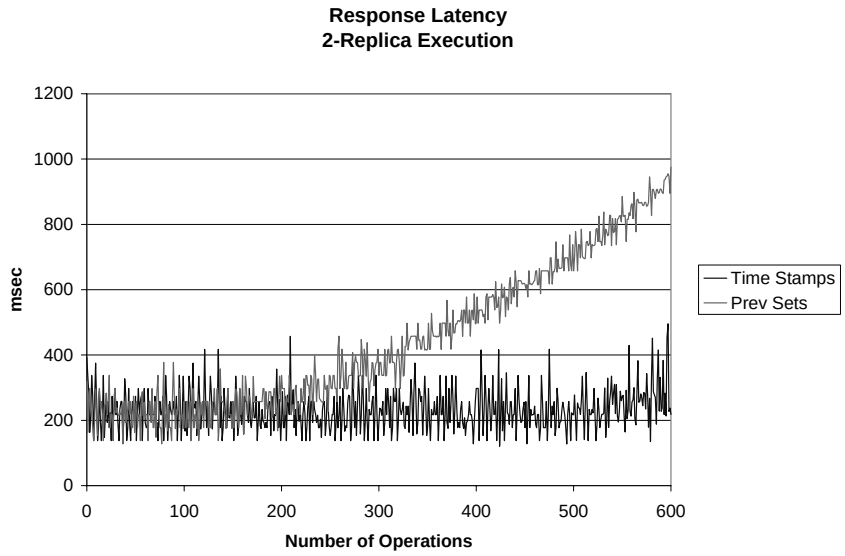

Figure 3: Response Latency - 2 Replica Execution

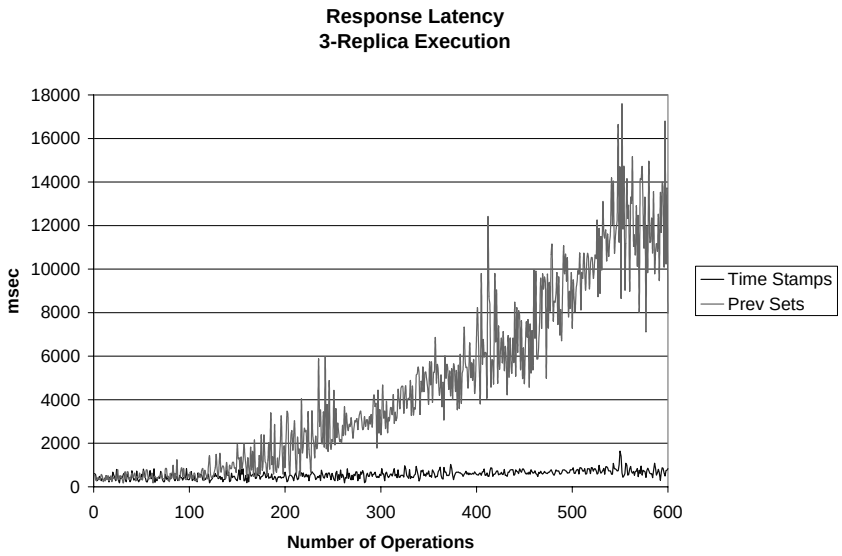

Figure 4: Response Latency - 3 Replica Execution

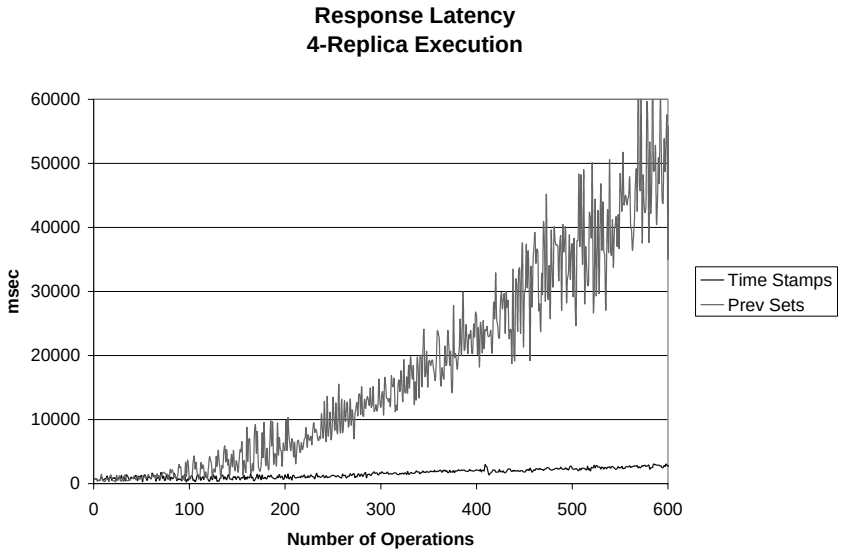

Figure 5: Response Latency - 4 Replica Execution

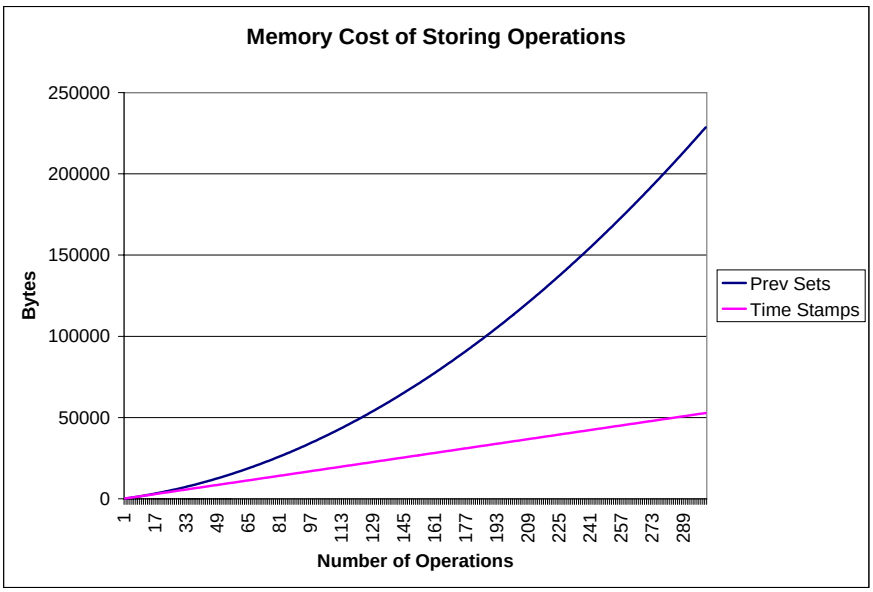

Figure 6: Memory Cost - 1 Replica Execution

Figures 6 and 7 contrast the memory cost of storing ESDS operations in the unoptimized *prev* set-based system and in the timestamp-based system. Each operation submitted to the system had all previously submitted operations in its prev set. Figure 6 plots the amount of memory taken up by operations at one replica as a function of the number of submitted operations in a 1-replica execution. Figure 7 plots the same for a 3-replica execution. In both cases we observed an $\mathcal{O}(n^2)$ total memory cost in the *prev* set system and an $\mathcal{O}(n)$ total memory cost in the multipart timestamp system. This is not surprising, since the size of a given operation's prev set is linear in the number of previously submitted operations, whereas the size of *prev-ts* and *op-ts* is constant.

Figures 8 and 9 plot the total amount of gossip traffic that ESDS replicas have generated by the time of response to each operation. Figure 8 shows the results for a 2-replica execution and Figure 9 shows the results for a 3-replica execution[7]. The communication costs shown in the figures have trends similar to the memory costs: an $\mathcal{O}(n^2)$ total cost for the *prev* set-based system and an $\mathcal{O}(n)$ total cost for the timestamp-based system. Since gossip messages consist entirely of operations, the reasons for the memory cost results apply to communication cost results as well.

7.2.2 Fault Tolerance

With the system extended by fault tolerance, we performed two categories of experiments. First, we measure the effect of replica crashes on latency and throughput of the system. Second, we measured the length of recovery in the fail-stop-restart model, as well as the impact of recovery overhead on performance.

To measure the impact of replica crashes and recovery on performance, we set up the following scenario. Two clients submit a continuous stream of non-strict operations without prev sets to two corresponding replicas. After a certain number n of operations (we used $N = 150$ in our measurement), one of the clients kills its corresponding replica and continues to submit its operations to the remaining replica. After another N new operations, the client restarts its own replica and immediately redirects its operations back to it. This cycle is repeated two times. Meanwhile, the other client keeps submitting its operations to its replica. Using our original test application and

[7]The 1-replica execution case is not interesting since there is no gossip in that configuration.

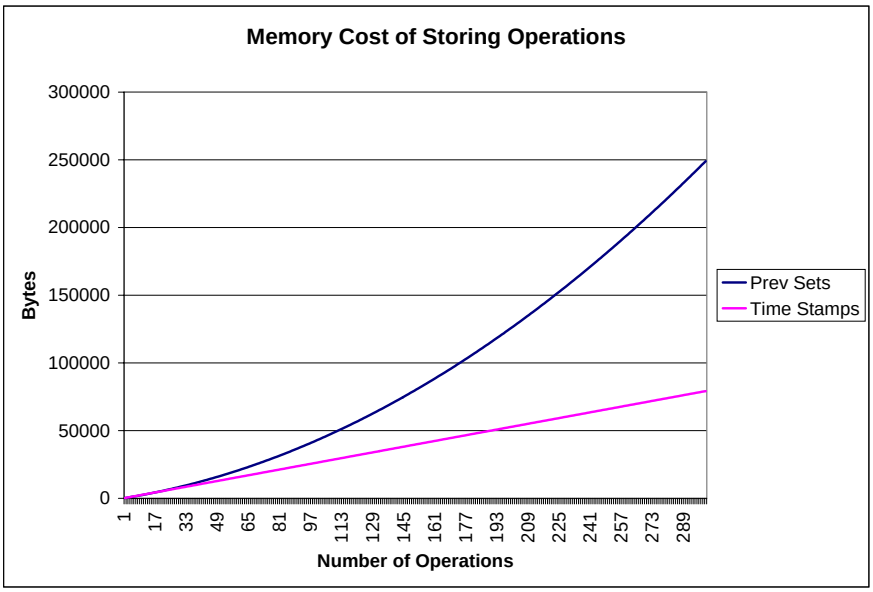

Figure 7: Memory Cost - 3 Replica Execution

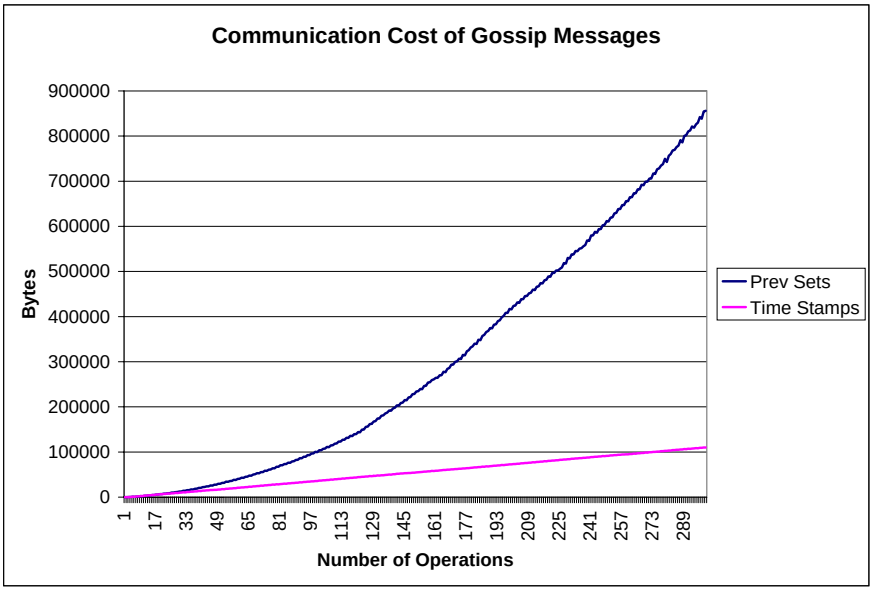

Figure 8: Communication Cost - 2 Replica Execution

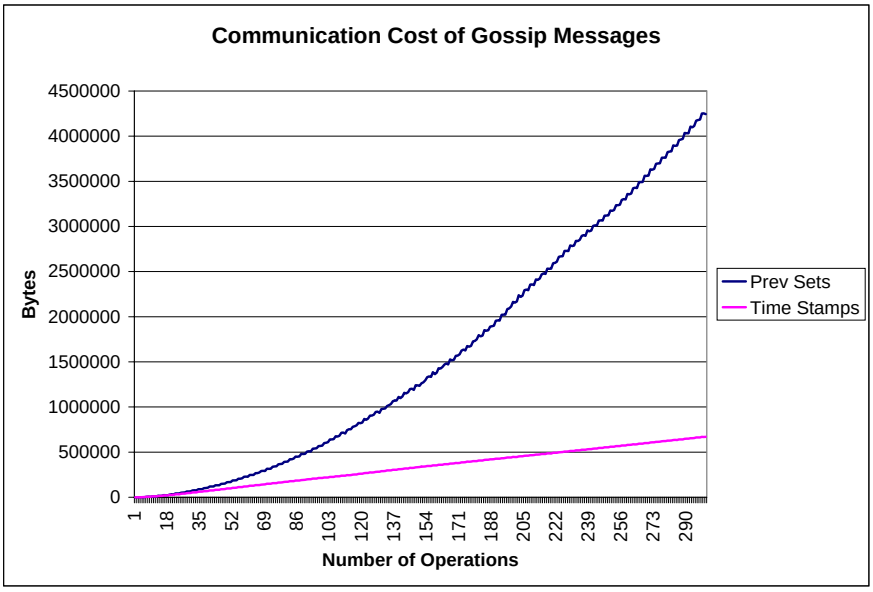

Figure 9: Communication Cost - 3 Replica Execution

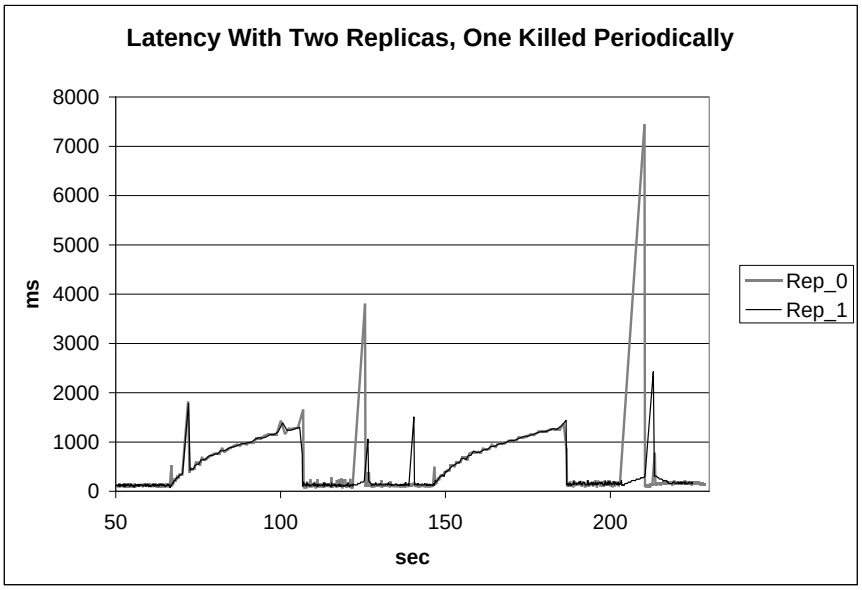

Figure 10: Impact of Crashes on Performance - A Simple Example

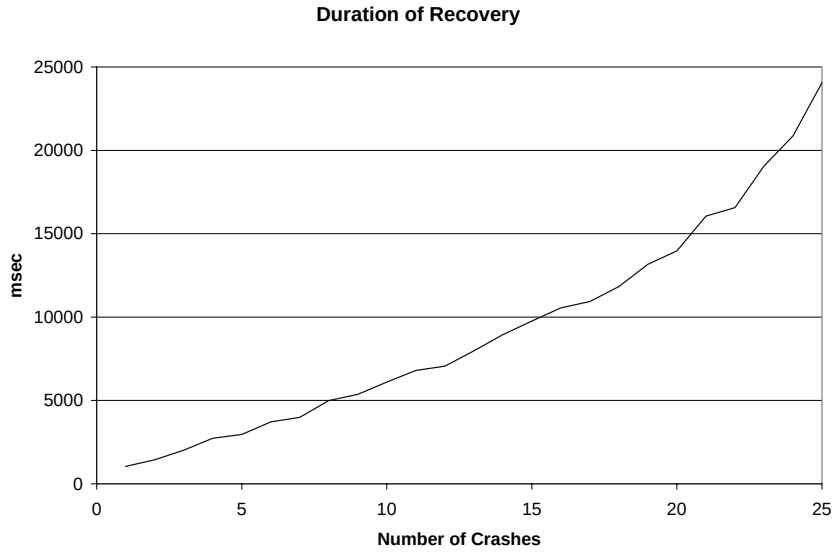

Figure 11: Crash Recovery Time

client code, we measured the latency of all operations in this execution, running the two replicas in parallel on two Pentium II machines. The results are shown in Figure 10.

In addition to a few random peaks (the two biggest ones at around 72 and 140 seconds), several interesting observation can be made from the graph.

1. The latency of the operation submitted immediately after the restart message is also the length of the recovery period of the restarted replica. We can clearly see two big peaks in the latency graph for replica 0. The peaks correspond to the operations submitted to the recovering replica right after the restart message. They show the recovery time (on the Y axis).

2. During the time period when replica 0 is crashed (and client 0 already submits its operations to replica 1), but replica-1 still does not know that replica 0 has crashed, we observe a linear increase in response latency at both clients. This is due to the fact that replica 1 is expecting gossip messages from replica 0 and does not stabilize its operations. This means that the number of done but unstable operations keeps increasing at replica 1. The replica re-applies such operations to the stable state every time it responds to a new operation. The time it takes to do this work grows linearly until the moment when replica 1 realizes that replica 0 has crashed. At this point replica 1 stabilizes all its done operations, and the latency drops back to its normal (pre-crash) level.

3. Although we expected the overall throughput to decrease during the period when only one replica is up, it did not happen. This shows that the remaining replica is not saturated even when it serves two clients at the same time.

   As mentioned in section 7.1.2, we measured recovery time with respect to the number of submitted operations for 2 replicas. The results appear in Figure 11. It is clear that the recovery time is increasing as the number of operations submitted to the system increases. This is not surprising, since restarting replicas rebuild their state from "scratch" by applying all submitted operations to the initial state.

8 Future Work

ESDS allows frontends to submit an operation to more than one replica. Using this feature, frontends can achieve increased performance by dynamic load balancing and make progress when the replicas they normally talk to go down. As we briefly noted earlier, the present implementation of timestamps does not work with this feature, as operations submitted to different replicas are assigned different timestamps. Future work is needed to work out the semantics of a timestamp model that correctly handles these cases.

With the present implementation, frontends do not participate in the failure-recovery mechanisms and do not check whether a replica is down. As frontends keep track of all pending operations at them, it is possible to resubmit operations that have been lost by crashed replicas from the frontend. Implementing this feature is also part of future work.

With the original implementation of prev sets, the system has to keep track of even the stabilized operations, as they can be referred to any time by a prev set of a new operation. For the time being, prev sets and timestamp prev sets coexist in the system, but a system using exclusively timestamps could delete stabilized operations from its memory. The future implementation of this feature is important, as it reduces the memory usage to a long-term bounded value.

Deleting stabilized operations also affects the way the recovery procedure of replicas is implemented. The present implementation relies on having all operations submitted to it available at the time of recovery. If replicas delete stable operations, this method will no longer work. It should be modified so that replicas could recover using a combination of the stable state of other replicas and unstable operations remaining in the system.

Finally, for our fault tolerance work we made strong assumptions about the communication and computation delays in the system. In addition to reliable communication channels, we assumed that a replica can not delay its keep-alive messages for a longer period of time than the length of the time-out. Assuming this, we did not have to handle cases when one (or more) replicas declare another one crashed, when in reality it was simply delayed. In a more realistic environment inconsistency can occur when a replica is still functioning after it has been declared to be crashed. Future work is needed to work out methods by which replicas can synchronize their decisions regarding this question.

References

[1] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-Serializable Data Services. *PODC* 1996, pp. 300-310.

[2] O. Cheiner. Implementation and Evaluation of an Eventually-Serializable Data Service. *Master of Engineering Thesis*, Massachusetts Institute of Technology, August 1997.

[3] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pp. 627-644, Oct. 1976.

[6] M. Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. *IEEE Transaction on Software Engineering*, 5(3):188-194, May 1979.

[7] B. Oki and B. Liskov. Viewstamp Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, August 1988.

[8] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180-209, June 1979.

[9] D. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th ACM Symposium on Principles of Operating Systems Principles*, pp. 150-162, December 1979.

[10] M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*. 4(1):32-53, February 1986.

[11] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690-691, September 1979.

[12] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190-205, March 1992

[13] M. Fischer and A. Michael. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proceedings of the ACM Symposium on Database Systems*, pp. 70-75, March 1982.

[14] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):260-274, 1982.

[15] H. Garcia-Molina, N. Lynch, B. Blaustein, C. Kaufman, and O. Schmueli. Notes on a Reliable Broadcast Protocol. *Technical Memorandum*, Computer Corporation America, October 1985.

[16] L. Lamport. Time, Clocks, and Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, July 1978.

[17] W. Gropp and E. Lusk. User's guide for MPICH, a portable implementation of MPI. *Technical Report ANL-96/6*, Argonne National Laboratory, 1994.

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: Exploiting the semantics of distributed services. *ACM Transactions on Computer Systems*, 10(4):360-391, Nov. 1992.

[19] The Message Passing Interface Forum. *The MPI message-passing interface standard.* http://www.mcs.anl.gov/mpi/standard.html, May 1995.

Design and Evaluation of a Compressed File System
Project Final Report
15-712 Software Systems

Jun Gao, Sanjay Rao, Peter Venable
(jungao, sanjay, pvenable@cs.cmu.edu)

1 Introduction

In this project, we consider issues involved in the design of a compressed file system. We propose algorithms for structuring a compressed file, and analyze the tradeoffs each make with regard to space and time efficiency. We hypothesize that no single algorithm would prove satisfactory for all possible file access patterns. We implement each of these algorithms at the user level, and evaluate their performance with workloads representing different sets of access patterns to validate our hypothesis. We also integrate all these algorithms into a single hybrid system that allows the user to select an appropriate algorithm for the access patterns he expects on each individual file, thus optimizing performance.

2 Algorithms for structuring compressed files

We present a few algorithms for structuring compressed files and discuss the tradeoffs each makes regarding the amount of metadata that needs to be maintained, the amount of space the compressed file occupies, and the time to access the compressed file. The algorithms are presented from the perspective of user-level implementation. They can be modified for better performance in a kernel implementation. For further discussion, see Appendix A.

In the following discussion, *Logical File* refers to the user's view of the file, and *Physical File* refers to the file in the underlying filesystem (eg. ext2fs) in which the compressed representation of the data in the logical file is stored. *Metadata* refers to the metadata that needs to be maintained over and above what a normal Unix filesystem maintains. A *Change* to a logical file refers to an operation that changes its contents without changing its size (e.g. replacing a character).

The algorithms are as follows:

1. The Physical file is obtained by directly compressing the entire logical file. The advantage is that this approach leads to maximum space efficiency, as there is maximum compression, and there is no overhead of maintaining metadata. The disadvantage is that an update, or a random access to a part of a large file requires dealing with the entire file.

2. The Logical file is split into fixed-size *Logical Blocks*. Each of these is compressed, and the compressed chunks are organized *contiguously* in the Physical File. The metadata required is a per-logical-block entry that keeps track of where the compressed chunk corresponding to that logical block is present in the physical file. This approach is efficient for random access of a large file. However, a change made to a logical block of a file may change the size of the corresponding compressed chunk, and this requires explicit copying of the rest of the file.

3. The logical file is split into logical blocks, each of which is compressed into chunks that are initially maintained contiguous in the physical file. Further, each chunk terminates with a "rest

of the logical block" pointer (which is empty initially). When any change to the logical block causes an increase in the size of the corresponding compressed chunk, space corresponding to the increase is allocated at the end of the file, and the "rest of logical-block" pointer is updated to point to this new allocated space. Note that repeated increases in size could lead to a logical block being represented by a chain of discontiguous chunks. When a change to a logical block causes a decrease in the size of the corresponding compressed chunk, the extra space is not freed but could be reused to accommodate future increases of the same chunk. During non-busy hours, the physical file may even be *cleaned offline*, to get rid of fragmentation and to ensure that each logical block is arranged contiguously rather than as a chain of non-contiguous blocks. This approach requires per-logical block metadata overhead that keeps track of where in the physical file the particular logical block was at the start of processing, as well as the current size of the logical block after compression.

4. The Logical file is split into fixed-size Logical Blocks. The Physical file is viewed as consisting of a set of fixed-size *physical blocks*. Each logical block is compressed; a certain amount of *forced space* is added to it and it is mapped onto *consecutive* blocks of the physical file. Note that the extra space reserved for a logical block is more than the minimum forced space added due to physical block alignment. The amount of forced space added is the same for all logical blocks of a given file, but it may vary from file to file depending on an indication given by the user. This algorithm requires a per logical-block metadata overhead that keeps track of where in the physical file the particular logical block is and the number of physical blocks that the logical block occupies. It is not highly space efficient due to internal fragmentation. A change in size to a compressed chunk due to a change made to the corresponding logical block of the file could in most cases be accommodated by the extra space available in the last physical block of that chunk, thus avoiding explicit recopy of the file. However, explicit recopy cannot be avoided when the change of size requires a change in the number of physical blocks to be allocated.

We summarize the trade-offs between individual algorithms in Figure 1.

2.1 Metadata

Each file begins with a generic header identifying the algorithm used and the logical and physical size of the file. Each of algorithms 2,3 and 4 requires a per-logical-block entry. For algorithms 2 and 4, all metadata is at the start of the file. There are a small number of direct entries for any file, and on demand larger blocks of metadata entries are allocated that can accommodate entries for say k logical blocks. An explicit recopy in these algorithms involves updating of all metadata entries, and having the entire metadata at the start is more efficient. The disadvantage of this is that during an append of a large file, a new metadata block needs to be created for every k logical blocks created, which in turn requires recopying of the file. In algorithm 3, we dealt with per-logical-block entries in a similar fashion to how UNIX inodes deal with disk-block addresses. That is, there are slots for a few direct entries, and a few indirect entries. The direct pointers point to real data blocks and the indirect pointers point to a metadata block which contains pointers pointing to real data blocks. These indirect pointer blocks are kept mixed with data blocks, instead of being grouped together at the beginning of the file, as in algorithms two and four.

3 Software Design

3.1 Description

The system in which we compare various algorithms for structuring compressed files is a "transparent" filesystem for linux. By "transparent" we mean that once the filesystem is mounted, it can be treated like any other mounted filesystem. Ordinary UNIX commands, such as ls, cp, cat, or rm, plus most existing software, work as expected, which has the bonus of making the system a prototype

No.	BRIEF DESCRIPTION	METADATA STORAGE	SPACE EFFICIENCY	TIME EFFICIENCY
1.	Compress or Decompress entire file.	No extra storage.	Highly efficient.	Good for small files, and sequential access of large files. Bad for random access, append or change to large files.
2.	Compress or Decompress logical blocks of file, store contiguously.	Per Logical block entry that tracks where the logical block maps to in the compressed physical file.	Less efficient than (1) as compression of smaller chunks.	Good for small files, sequential access , random access and update of large files. Bad for change to large files - recopy.
3.	Compress or Decompress logical blocks of file, store contiguously. "Rest of logical block" pointer is maintained for each compressed chunk. If change in logical block leads to increase in size of compressed chunk, create space at the end corresponding to the size-increase and update the "rest of lb pointer". Any space freed due to a decrease in size is reused to accomodate future increases. Also, clean offline.	Per logical block entry that tracks where the logical block mapped to in the compressed physical file at the start of a run., and the current size of compressed logical block. also, pointer space for each fragment of the compressed chunk corresponding to each logical block.	Slightly worse than (2) offline. Potentially suffers from internal fragmentation online, due to shrinkage, but this is likely to be small.	Similar to (2) for all kinds of accesses, except for change where much more efficient.. Also, may suffer from slight performance loss for reads due to parts of a logical block being discontiguous.
4.	Map each compressed logical block onto integral number of consecutive physical blocks, forcing some space for each logical block. Causes internal fragmentation.	Per logical block entry that tracks where the logical block maps to in the compressed physical file, as well as the number of physical blocks occupied by that logical block.	Worse than (2) due to internal fragmentation. Worse than (3) too.	For changes, usually similar to (3). However may be as bad as (2) if explicit recopy needed.

Figure 1: Algorithms for Structuring Compressed Files.

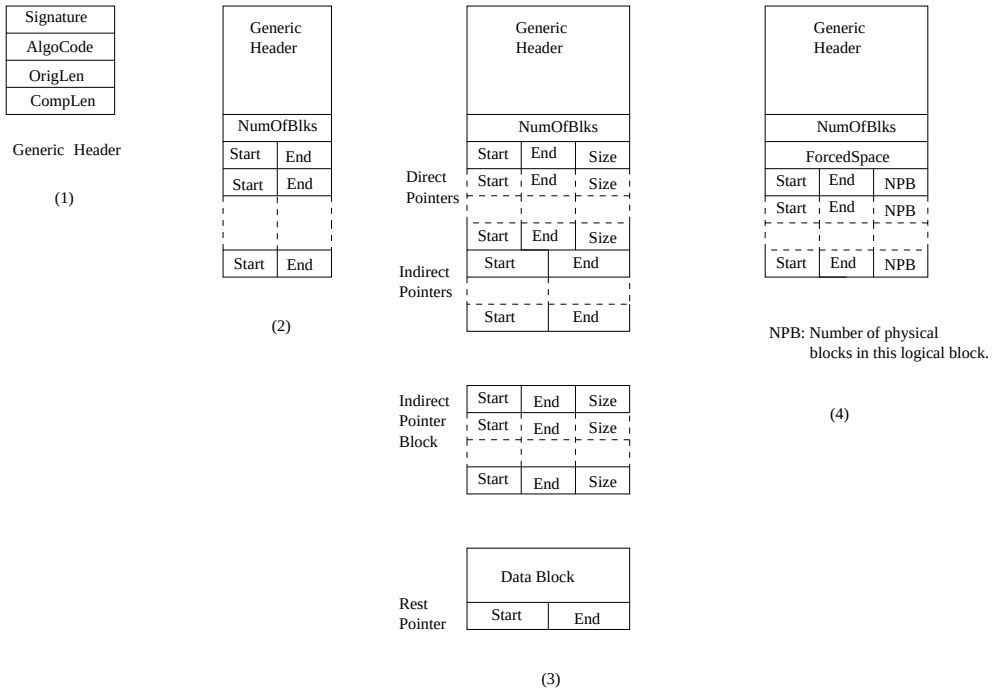

Figure 2: Metadata used in algorithms 1-4.

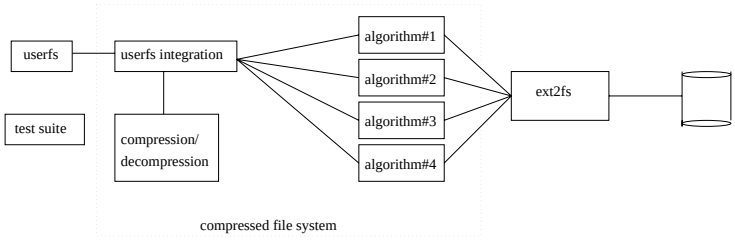

Figure 3: Module Decomposition

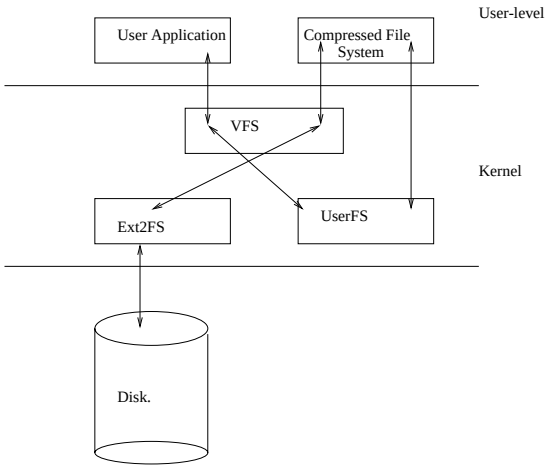

Figure 4: Filesystem Access Data Path

for a truly general-purpose compressed filesystem. Of course, in this experimental system, some less crucial features of transparency are not implemented, but they could easily be added in a production system. Similarly, this prototype is implemented with the crutch of *userfs*; in a production system speed would be greatly enhanced.

3.2 Module Decomposition

Module Descriptions

1. *userfs integration* - Handles filesystems calls which are passed though *userfs*. Provides a clean and small interface to the compression/decompression module and to each algorithm.

2. *compression/decompression* - Compresses and decompresses data. Handles blocks either individually or in a stream.

3. *algorithms* - Each algorithm described above will be implemented separately, conforming to the same interface to the userfs integration module.

4. *test suite* - The test suite will apply the workloads described below and gather cost information for analysis and evaluation.

Figure 3 illustrates the relationships between these modules.

3.3 UserFS

Our implementation is on a Linux platform. We use *userfs 0.9.2*. Userfs is a mechanism that allows a normal linux user process to be a filesystem. This needs to be loaded into the kernel as a module.

Once a userfs module is mounted to a directory, such as /mnt/compressed, any filesystem accesses referring to files within that directory tree are sent to the userfs module. In order to implement a compressed filesystem, we wrote a module to handle all the major filesystem calls, compressing and decompressing along the way. When a system call requests data through userfs, our module looks up the physical file corresponding to the virtual file in the mounted directory, processes metadata to find the correct part of the file, decompresses the data, and passes it through userfs to the application which made the (ordinary) system call. The system works similarly for other operations, such as writing to a file. Figure 4 traces the path of a system call.

3.4 Hybrid System

As an application of this knowledge, we have implemented the compressed filesystem with all four algorithms available in a hybrid fashion. That is, files may be individually specified as to which algorithm they should use, so the best algorithm for each file can be chosen, depending on its specific workload.

The user specifies which algorithm to use by simply typing a prefix to the filename when creating the file. For example, to use algorithm 2 when creating a file named myfile, use a filename like /mnt/compressed/algorithm2:myfile. The forced space for algorithm 4 may be specified in a similar fashion. For example, to use algorithm 4 with 128 bytes of forced space per logical block, use a filename like /mnt/compressed/algorithm4:128:myfile.

This system also makes it very simple to do the "offline cleaning" recommended for files which become fragmented after many updates. To clean a file, simply copy it to a temporary file, delete the original, and recopy the the temporary file back to the old name. This causes the file to be freshly written with no fragmentation.

4 Evaluation Methodology

4.1 Hypothesis

Particular algorithms for structuring compressed files should work better than others for specific file access patterns, and different ones should work better for different workloads. We verify this with empirical results and demonstrate which algorithms are appropriate for which workloads, and investigate how much difference the choice of algorithm makes.

4.2 Test Circumstances

The test software is a set of simulations written by ourselves, and therefore easy to control. It repeatedly runs the same set of tests (including various workloads) using various algorithms. The results of the test consist of time and space measurements. The space measurements are easy to observe by simply inspecting the files.

For time measurements, we use the system real-time clock, purportedly accurate to one microsecond, which is well in excess of the precision needed for our measurements, which are usually in the tens of seconds. We chose real time to insure that all relevant time expenses are included, such as system time, userfs time, I/O time, etc. The downside of this is that other system processes can affect the results. We minimize this by running the final tests on a minimal system, that is, with no other work going on.

Since the test software involves some use of randomization, the random number seeds are regulated so that the exact same test is used for each algorithm. That way the results of each test run are directly comparable across algorithms. In different test runs, different sets of random numbers are used.

4.3 Workloads

The following list of file access patterns has been selected from a wide range of possible patterns as representative workload set. Each workload is annotated with a real-world example it represents. We had considered the possibility of testing a much wider range of workloads, but concluded that it is better to have a small set of realistic workloads representing familiar tasks than a more complete set that includes many loads rarely encountered in practice, which would result in a proliferation of data and obscure the results. Hence, these representative workloads should suffice:

1. Sequential read of a large file (e.g. grep).
2. Many appends of a very large file. For example, consider applications that monitor and log traffic passing a network. The log could grow very large, recommending compression, but frequent updates must be handled efficiently.
3. Random rewrites of records, interleaved with sequential reads (Eg. Database applications).

4.3.1 Files used

A mention of the files that we used is in order, as the compression process depends highly on the nature and content of the file, and could have an important bearing on the results seen. The files used are summarized in the table below. In future, we refer to these files by their file sizes. An important limitation in our evaluation is that the biggest file we used was only 4 MB, and potentially more interesting results could have been obtained with larger files.

File Size	Description
20K	Text File containing mainly alphabets, and a few digits.
130K	Text File containing mainly alphabets, and a few digits.
1MB	A file that contains a list of several IP addresses.
4MB	A file that has a log of several Network packets.

5 Results and Analysis

We present and discuss the results obtained for each of the workloads on every algorithm. Algorithm 0 refers to an algorithm that does no compression, and is used as a baseline for comparison. (It is implemented so as to experience the overhead of userfs, except that it directly accesses an uncompressed version of the file).

5.1 Sequential Test

This test was conducted by sequentially reading a file 100 times. Reported below is the total time for the 100 sequential reads, and the percentage compression obtained ($(Origsize - Compsize) * 100/Origsize$).

Sequential						
	Time(sec)			Compression(%)		
Algorithm	20K	130K	4MB	20K	130K	4MB
0	0.35	2.10	107.21	0	0	0
1	2.95	15.19		50.26	60.38	
2	10.35	73.49	1966.51	44.59	47.66	69.63
3	10.39	73.67	1944.88	42.16	47.42	69.30
4	10.41	73.47	1959.37	41.28	43.58	66.60

As expected, Algorithms 2,3,4 perform very similarly in terms of time-efficiency while Algorithms 2 and 3 do slightly better than Algorithm 4 with regard to space-efficiency. Also as expected is

Algorithm 1's better space efficiency as compared to the other algorithms. While we expected Algorithm 1's time efficiency to be slightly better than that of other algorithms, we were surprised to find the remarkable improvement of a factor of five. This may be explained due to our optimized implementation of Algorithm 1. We wanted to avoid decompressing the entire file for the read of each block during a sequential read of the file. To do this, we maintained enough state with a file being read, so that if it was identified that the file is being sequentially read, then we could carry on the decompression process from where it had stopped in the previous read. However, the previous read call may have caused more data to have been decompressed than needed by that call itself; To ensure that any meaningful optimization is derived, we decided to cache this extra data. Consequently, we were using a cache of $8KB$. Note that this caching is inherently essential for the good performance of the algorithm, and even a modest cache of $8K$ could mean a tremendous performance improvement for the algorithm. No doubt, a good performance improvement could be expected using caching in the other algorithms as well.

5.2 Append Test

In this test, we started off with a file size 0, and appended data constantly till it reached the sizes shown in the Table below. Each append operation itself appended 10 random words from a collection of about 300 English words, as well as some redundant information that was common across all appends. After each append the file was flushed on to disk. The Table below reports the file-sizes created, the compression ratio achieved and the total time for the final file to be created after all the appends.

Append						
	Time(sec)			Compression(%)		
Algorithm	20K	130K	4MB	20K	130K	4MB
0	0.08	0.49	16.86	0	0	0
1	53.41			61.45		
2	10.68	74.85	2573.77	54.06	53.69	55.59
3	10.68	75.78	2613.59	42.96	44.62	46.51
4	10.50	74.60	2486.90	49.90	47.74	49.89

Algorithms 2 and 4 seem to perform better than Algorithm 3 with regard to space efficiency. The reason for this is that repeated appends to Algorithm 3 causes each logical block to be represented as a chain of consecutive fragment, each fragment having the associated overhead of the next fragment pointer. However, we believe that this shortcoming can be overcome in a future implementation by modifying Algorithm 3 so that if there is an increase in size of the compressed chunk corresponding to that logical block which owns the last fragment in the physical file, then, the size of the last fragment is increased, rather than a new fragment created. Algorithm 2 is slightly better than 4 because it has no reserved space at the end of each physical block.

Algorithms 2,3,4 perform similarly in terms of time efficiency. Algorithm 1 is horribly slow in this instance, because each change to the file (each append) requires the decompression, and subsequent recompression of the entire file, resulting in $O(n^2)$ time cost (where n is the number of appends). This is so bad that we didn't even bother measuring algorithm 1 on large files.

5.3 Database Test

Each of the test files was considered to consist of records of size 64 bytes. A change operation was implemented by reading some random record (64 bytes) of the file, and modifying it. The modified record consisted of 64 randomly generated letters of the English Alphabet. Usually the modified record was more random than the initial record, and would increase the size of the compressed file. The Database test itself consisted of 10 runs, each run involving 50 changes

to the file, and 1 sequential read. The file was opened before and closed after each of the 10 runs. The table below presents the total time for the 500(10*50) changes, and total time for the

10 sequential reads. The compression percentage values give the compression percentages after the changes have been made.

Database Test									
	Change(sec)			Read(sec)			Compression(%)		
Algorithm	20K	130K	4MB	20K	130K	4MB	20K	130K	4MB
0	0.16	0.25	0.59	0.03	0.21	7.49	0	0	0
1	270.93			0.33			30.96		
2	53.81	61.94	45.94	1.30	8.20	177.77	23.97	37.31	68.94
3	54.35	61.69	31.89	1.36	8.31	178.59	11.28	34.29	68.52
4	53.66	61.46	34.12	1.31	8.23	198.00	19.54	33.92	65.85

Algorithm 1 has the best compression; Algorithm 2 has better compression than Algorithms 3 and 4. Between Algorithm 3 and 4 however, the space efficiency is better for Algorithm 3 for larger files, while it is better for Algorithm 4 for smaller files. There is wastage of space in Algorithm 4 due to internal fragmentation; On the other hand heavy changes to a small number of logical blocks (as with the 20K file which had 3 logical blocks and 500 changes), would cause each of these logical blocks to consist of a long chain of fragments in Algorithm 3, each fragment having some space wasted for the rest of the chain pointers. For larger files however, it is more unlikely that the same logical block is so heavily changed, and secondly the pointer overhead becomes more insignificant.

When we consider change-times, Algorithm 1 is extremely slow, and we deemed it sufficient to show the poor performance for a small file. Algorithm 3 is almost 33 % faster than Algorithm 2 for changes to the 4 MB file. This is because it avoids explicit recopy of the file, that is needed for every change in Algorithm 2. Algorithm 3 is slightly faster than Algorithm 4, because of the occasional recopies required for Algorithm 4. For smaller files, the change times of the three algorithms is almost indistinguishable; this may due to the low costs of explicit recopy of a small file.

While the main purpose of this test was to analyze the change-times, we also measured the sequential-read times mainly to see if there was any loss of performance in sequential reads with Algorithm 3. We found that the loss of performance is marginal, if at all; more surprisingly, there seems to be a slight deterioration of performance for Algorithm 4. We believe that the good performance for Algorithm 3 is probably explained by the fact that the fragmentation is very light for the larger 4 MB file; for the smaller files good values are seen in spite of heavy fragmentation possibly because of a read-ahead mechanism in the underlying filesystem (ext2fs). We do not have a satisfactory explanation for the poorer performance of Algorithm 4: one possible explanation is that the compressed file is larger in size, and the extra time is due to the reading of a bigger file.

5.3.1 Algorithm 4 Parametrization

All our experiments involving Algorithm 4 so far involved a forced-space of 0. In this section, we study the effect of varying the forced space. Of the operations read,append and change, the only operation where we expect a difference in performance is change and we focus only on this. In this case, the test files were viewed as consisting of records of size 256 bytes. A change operation involved reading a random 256 bytes, and replacing it with a set of random alphabets. The test itself involved 10 runs, each run consisting of 100 change operations. The file was opened before and closed after each run. Presented below is the total time taken for all the (10*100) changes, the compression rates before any change was made, and the compression rates after all the changes were made.

Algorithm 4						
	Time(sec)		Compression(%) Initial		Compression(%)Final	
FS	1MB	4MB	1MB	4MB	1MB	4MB
0	76.33	80.70	74.23	66.59	51.98	60.69
64	76.85	79.56	74.08	65.71	51.92	60.49
256	75.61	73.28	74.08	63.20	51.92	59.90

Our initial expectations were that increasing the forced space would lead to poorer space performance, but better time performance. While our observations seem to indicate the trend, it is not as pronounced as we had expected. For the other files we tried ($< 1MB$), the variation in forced-space caused almost no difference in performance.

6 Conclusions and Future Work

In this project, we considered issues involved in the development of a compressed file-system. We specifically focussed on the problem of structuring a compressed file in this system. We have built a hybrid system, that supports different algorithms for structuring compressed files, and have shown that particular algorithms perform better with particular access patterns. The user has the choice of indicating which algorithm he would like to use based on the access patterns likely to be seen on the file. Based on our evaluations, we make the following recommendations:

1. Sequential Read - Algorithm 1 with minimal caching works very well; Further it gives the best compression ratio. The cache required is small (8K), and is critically essential for good performance of the Algorithm.

2. Appends - Algorithm 2 works well being as time-efficient as Algorithms 3 and 4, yet having a much better space efficiency.

3. Changes - Algorithm 3 seems to be the best candidate. It evidently outperforms Algorithms 1 and 2. Our evaluation shows its performance to be slightly better than Algorithm 4 in terms of time efficiency; It is better than Algorithm 4 with regard to space efficiency (except for small Files); Finally, it has the benefit that it can be cleaned offline, which is not available to Algorithm 4. We had feared a slight loss in performance for sequential access; however such a loss was not revealed in our evaluations. Further, our evaluations revealed only a marginal improvement on forcing free space in Algorithm 4. On the balance, it is fair to choose Algorithm 3 over Algorithm 4 in most cases.

While our system functions as a prototypical compressed filesystem, demonstrating a basic amount of transparency and showing the relative merits of various algorithms, it is not really practical for regular use. It is too slow to compete with other filesystems, and it doesn't support all operations, such as symbolic links. While it would be relatively simple to extend its functionality to include all the regular filesystem operations, the efficiency can only be improved so much while the system resides in user space, passing all calls through the extra userfs layer. A production version of this system should be integrated with VFS inside the kernel instead of using userfs. Not only would this avoid the overhead of multiple redirections of system calls, but it would allow some of the algorithms to be improved through direct access to the underlying disk layout mechanism and would enable metadata to be kept seperate from the data along with the traditional UNIX inode.

A look at the various tables shows us that the time-efficiency of all our algorithms is disappointing when compared to the efficiency of an uncompressed system. Note that this overhead is not due to userfs, as the uncompressed system was also allowed to experience the overhead associated with userfs. However, adopting good caching-mechanisms could help alleviate the overhead associated with compression somewhat; The remarkable performance of Algorithm 1 on sequential accesses with just an 8K cache seems very encouraging in this respect. Caching mechanisms that must be investigated include (i) Caching of Metadata on the opening of a file, as some of our algorithms require frequent updating of metadata entries; (ii) Caching of logical blocks when they are uncompressed, so that a subsequent access to the same logical block does not require the compression overhead.

Finally, in this project, we have not considered issues related to fault tolerance (e.g. If a system crashes while we are in the process of updating metadata, recopying file etc. then the file contents are badly destroyed). However, this question is orthogonal to the design of a compressed file system itself, and methods normally employed in regular file systems could be used.

7 Relationships to Course Material

The main focus of this project was on comparing and improving compressed file layout algorithms, which took place in the context of the development of a compressed filesystem. Clearly, this is directly related to the class topic of filesystems. In addition, at a high level our investigation is directed at identifying situations where each of several competing techniques is effective and trying to support all of them, rather than discover one globally best technique. This approach has been motivated by several class readings (e.g. Munin).

A Kernel Implementation

We discuss how the algorithms described earlier may be implemented at the kernel level one by one:

1. Can be implemented as is.
2. Metadata is added to the usual contents of each UNIX inode. Per logical block entries may be organized in the fashion of direct, indirect and doubly indirect blocks. An optimization is that the per-logical-block entry need only contain the byte in the compressed file where the corresponding compressed chunk starts.
3. Metadata is added to the usual contents of each UNIX inode, adding doubly indirect pointers.
4. Metadata is added to the usual contents of each UNIX inode. Per logical block entries may be organized in the fashion of direct, indirect and doubly indirect blocks. The block size of the physical file may be fixed as the physical disk block size. An explicit recopy of data can be avoided, by having only a recopy of inode pointers.

B Compression Algorithm

We have chosen the Lempel-Ziv-Welch (LZW) algorithm[1] to compress and decompress data. LZW was developed by Terry Welch in 1984 for hardware implementation in high performance disk controllers by refining an earlier algorithm published by Lempel and Ziv in 1978. This algorithm belongs to the category of dictionary methods in data compression, which utilizes the property of many data types containing repeating code sequences. Text files and image files are two good examples of such data type.

The LZW method is very popular in practice (e.g. image GIF format, UNIX compress utility) and its major advantage over other dictionary methods (e.g. LZ77) is its speed, because the number of string comparisons is significantly less.

The LZW algorithm creates a dictionary of the phrases that occur in the input data. When an encountered phrase is already present in the dictionary, only the index number of this phrase will be sent to the output.

This method also does not need to read in the whole input data to perform the compression which gives no limitation on the file size, i.e. the file length can be much larger than the available memory size.

The algorithm itself contains two parts: Encoding and Decoding.
Notation: P = current prefix, C = current character.

B.1 The Encoding Algorithm

```
1. At the start, the dictionary contains all possible roots, and P is empty;
2. C := next character in the charstream;
3. Is the string P+C present in the dictionary?
      a.if it is, P := P+C (extend P with C);
      b.if not,
              i.output the code word which denotes P to the codestream;
             ii.add the string P+C to the dictionary;
             iii.P := C (P now contains only the character C);
4. Are there more characters in the charstream?
         if yes, go back to step 2;
         if not:
              i.output the code word which denotes P to the codestream;
             ii.END
```

[1] LZW algorithm is a patent of Unisys. (US Patent No. 4558302). Free use of the method is allowed except for the producing of commercial software

B.2 The Decoding Algorithm

```
1. At the start the dictionary contains all possible roots;
2. cW := the first code word in the codestream (it denotes a root);
3. output the string.cW to the charstream;
4. pW := cW;
5. cW := next code word in the codestream;
6. Is the string.cW present in the dictionary?
        if it is,
             i.output the string.cW to the charstream;
            ii.P := string.pW;
            iii.C := the first character of the string.cW;
            iv.add the string P+C to the dictionary;
        if not,
             i.P := string.pW;
            ii.C := the first character of the string.pW;
            iii.output the string P+C to the charstream and add it to the
               dictionary (now it corresponds to the cW);
7. Are there more code words in the codestream?
        if yes, go back to step 4;
        if not, END.
```

References

[1] K. Sayood, *Introduction to Data Compression*, pp.97-116, Morgan Kaufmann Publishers, Inc., 1996.

[2] Data Compression Reference Center, *Compression Algorithms*, http://www.rasip.fer.hr/research/compress/index.html, Faculty of Electrical Engineering and Computing, Zagreb, Croatia.

[3] Jeremy Fitzhardinge, *Userfs 0.9.4.2*, http://sunsite.anu.edu.au/archives/linux/ALPHA/userfs/userfs.lsm

Dishrag: Distributed, Shared Objects in Java

Doug Rohde, Rick Romero, Philip Wickline

We have created a system, Dishrag, which supports distributed computing in the Java language. This provides a library of routines for object control and sharing along with a Java pre-processor which tracks object modifications for efficient updating. Dishrag offers ten sharing policies, which differ in their level of concurrency control and are best suited for particular access patterns. We analyze the performance of these policies on various tasks, assess the overhead in each basic operation of the system, and describe several distributed applications that have been implemented using Dishrag.

1.0 Introduction

Traditionally, shared memory packages, such as Munin, have operated at the level of a virtual memory page. However, from the programmer's perspective, code is typically designed around structured memory. This realization has led to the development of object-oriented programming languages which reify this conceptual structure. Therefore, unless one wishes to share a complete address space, it makes little sense to design a sharing system around memory pages when applications are designed around objects. Therefore, we have developed a shared memory system which operates at the level of objects, making it a more natural extension to methods of writing conventional single-processor applications.

Our system, Dishrag (the *di*stributed *shar*ing *ag*ent), is an extension to the Java programming language. It allows the user considerable control over the object locking and distribution for robust and efficient concurrency without significantly altering the object-oriented programming model. In this paper, we outline the interface and implementation of dishrag. We describe and analyze the various sharing policies supported by the system and describe several applications that have been implemented using Dishrag, included parallel sorting and matrix multiplication and a parallel theorem prover.

2.0 Interface Design

This section describes the user interface to Dishrag. Additional details of functionality will be developed in the following sections. Currently, the management of shared objects is handled through an auxiliary control object associated with the user-defined shared object. An alternative design would have been to insert control information into the users' objects during preprocessing. However, this would have made the system less adaptable to the sharing of primitive types and other classes whose source code is unavailable.

A user makes an object shareable by labelling it as implementing the `Shareable` interface. They need not define special methods for the object, other than a basic constructor. The preprocessor will insert the necessary methods into the object prior to compilation.

2.1 Shared Object Creation

Static methods of the `Sharing.Share` class are used to gain access to a control object, which is of type `SharedObject`. The `Publish` procedure creates an object server to handle requests for the object and registers an entry for it accessible to outside processes.

```
SharedObject Object Share.Publish(Object O, String name, int policy,
     int history);
```

Dishrag objects are not migratory, in the sense that the object server remains in the same process for the lifetime of the object. Publish automatically subscribes to the shared object, returning a control object whose use is described below. The `name` argument specifies the name by which subscribes with address this object. `policy` sets the sharing model used for this particular object. Once set, sharing policies cannot be modified. The policies available in Dishrag are described in Section 4.0, on page 4. The `history` field is used by the object server to determine how many old versions of the object will be remembered for clients who have previously accessed the object. This is used to minimize the information necessary to update a client's copy.

The `Unpublish` call will remove the network entry for the named object, preventing further subscriptions. Current outstanding copies of the object, however, remain valid.

```
void Share.Unpublish(String name);
```

`Subscribe` is used by a client to access a shared object. The `name` is as described in the `Publish` call and `host` is the machine on which the object was published.

```
SharedObject Share.Subscribe(String name, String host);
```

2.2 Object and Lock Management

The following methods are implemented by the class `Sharing.SharedObject`:

```
Shareable GetObject();
void AcquireRead();
void ReleaseRead();
void AcquireWrite();
void ReleaseWrite();
```

`GetObject` returns the user's actual object. It might seem desirable to provide a shared object system that hides all aspects of locking from the user. However, this is a severe limitation because it would preclude the user from doing such things as locking two objects simultaneously during a safe dual-update. While Munin only supports a single form of lock, Dishrag offers both read and write locks. This allows interactions between clients to be considerably more efficient, with features such as buffered writes and write locks that allow readers to continue but block other writers.

The exact behavior of the acquire and release procedures is protocol dependent. In non-*request* protocols, the user is guaranteed to have the latest version of the object on return. Provided that the user protects all code that

alters the object with these locks, the object is always guaranteed not to change, aside from user modifications, prior to a release.

Four additional commands are used in some policies:

```
void Commit();
void StopUpdates();
void AcquireUpdatedRead();
void AcquireUpdatedWrite();
```

In *access* policies, changes are buffered and are only written back to the server when another client wishes to access the object. `Commit` forces the changes to be written back immediately. This is necessary, for example, if the client wished to exit. `StopUpdates` is used in *update* policies, in which the client automatically receives all changes. Automatic updates will resume upon the next read or write acquire. In *request* policies, the user is allowed to retain out-of-date objects and must explicitly request updates. This is done using the `AcquireUpdatedRead` and `AcquireUpdatedWrite` methods.

2.3 Remote Threads

In order to facilitate distributed programming, Dishrag provides methods for starting threads on remote machines.

```
void Launch.launch(Runnable r, String where);
```

We do not intend to emulate a sharing mechanism in which the new thread “inherits” all of the class members of the parent thread. All sharing of this sort must be done explicitly by the programmer.

2.4 Synchronization

While the acquire and release methods provide one level of synchronization control, robust concurrent programming requires additional mechanisms for synchronization. Therefore, Dishrag supports barriers, which can be used as traditional barriers which wait for a fixed number of accesses before releasing all waiters or can act as “conditions” which allow direct notification.

```
Barrier BarrierObj.Publish(String name, int limit);
void    BarrierObj.Unpublish(Barrier B);
Barrier BarrierObj.Subscribe(String name, String host);
```

The above procedures are much the same as those for shared objects, though a barrier is not created prior to publishing, as is an object. The `limit` field is used to specify how many waiters are required before the barrier will break. A non-positive value is used for a condition, which has no limit on the number of waiters.

The following calls can be made on a `Barrier`:

```
void Wait();
void Notify();
void NotifyAll();
```

`Wait` will cause the current thread to wait until a notify occurs or the barrier breaks. `Notify` is guaranteed to wake up a single waiter and `NotifyAll` will wake them all up.

3.0 The Preprocessor

In order to minimize the cost of updates, we have taken special measures to avoid shipping entire objects over the network. Using a bit field, we keep track of which fields of an object have been modified since an update, and ship the contents of only those fields. The preprocessor is responsible for annotating the user's code so that the proper bit is set whenever an assignment is made to a field. In addition, the preprocessor inserts methods into the object for generating and interpreting exportable, compact representations of updates to the object. By adding methods instead of relying on Java introspection, we have full access to all fields of the object instead of just the public ones.

Ideally the preprocessor would recognize repeated updates and coalesce the bit field statements into a single update at the end of a block or following a loop. The current version only performs simple tracking, however, and all changes to object fields must be contained within methods defined by that object. More efficient assignment tracking could be implemented with a modification to the Java virtual machine.

4.0 Sharing Policies

In order to produce efficient behavior under a variety of access patterns, Dishrag provides ten sharing protocols. Earlier systems, such as Munin, offered a collection of disparate protocols of seemingly ad hoc design. On the other hand, the Dishrag protocols are based on a simple, coherent framework: each protocol is determined by a reasonable pair of values of just two parameters. Nevertheless, we believe that this set will be able to efficiently handle the specific access patterns for which special Munin protocols were designed. This is partially due to the existence of distinct read and write locks provided by our system.

4.1 Parameters

The first protocol parameter is the *synchronization* mode. There are five possible modes: *read-only, exclusive*, *access, release*, and *multiple-writer.* Exclusive allows only a single reader or writer at a time. Access and release are two forms of single writer/multiple reader modes. In access, a writer blocks other readers and writers when writing begins. The writer need not do anything special upon release of the write lock. In fact, unless there are other clients waiting for the object, changes need not be written back to the server upon release of the write lock.

In release synchronization mode, on the other hand, a working writer blocks other writers but allows readers to continue. This results in a greater level of concurrency and makes it faster to begin writing. However, when the write is released, the changes must immediately be written back to the server and versions held by the other clients either invalidated or updated. Therefore, the trade-off is that buffering is not possible and write releases are more expensive. Multiple-writer mode is essentially equivalent to release mode except that writers treat other writers just like readers.

The second protocol parameter is the *modification* mode. There are three such modes: *invalidate*, *update*, and *request*. Invalidate mode causes all clients to be invalidated by a write. If the synchronization is access, this occurs when a writer acquires the lock. If the synchronization is release, this occurs when the writer releases the lock. Update mode is similar except that other clients automatically receive updates rather than invalidates. Note that in update mode each client will always have the previous version when a write occurs so the same update may be sent to all of them and no modification history needs to be stored in the server.

In request mode, users are allowed to retain out-of-date copies. No invalidates or updates are ever sent and clients must explicitly request any updates. This might be desirable in a situation similar to a web page in which a single resource is potentially read by very many clients, the server does not wish to keep track of the clients as it must with an invalidate or update policy, and it is not detrimental for a client to be reading a slightly out-of-date copy. Munin does not appear to support such a policy.

Although there are five synchronization modes and three modification modes, not all pairs define reasonable policies. Read-only mode is only consistent with a request policy. Exclusive control is only consistent with invalidate mode. Finally, access synchronization does not make sense with an update policy because access relies on requests by other clients to cause a write-back by the last writer and read requests are only made the first time a client locks the object in update mode. What remains are ten policies, described individually below for clarification.

4.2 Protocols

Read-only: Although users may write their local copies of such objects if they wish, read-only objects have no mechanism for sending changes back to the server. The object is sent to the client by the server on creation and at the user's request thereafter.

Exclusive: The exclusive policy allows only one readers or writer at a time. Any read or write attempt will cause the current owner, if there is one, to write back any changes they have made once they release their lock. The new owner is then sent any necessary changes as is allowed to proceed. Repeated writes by the same client will be buffered and only written back when another client wishes to obtain the lock. Munin's *migratory* policy is like this, although it does not use a fixed server.

Access-invalidate: This and the next four policies allow multiple readers but only one writer. In this policy, invalidates are sent to all readers before a writer may begin to write. Therefore, readers are not allowed to hold out-of-date copies. As in *exclusive*, the writer only sends changes back to the server when the server requests them. The invalidate and request policies use version numbers to allow the server to keep track of which pieces of the object have changed since a client's previous request so that only necessary updates are transmitted. This is equivalent to Munin's *conventional* policy.

Access-request: As in all request policies, no updates or invalidates are sent in this policy. If a writer is working, read or write requests will be blocked until the writer is done. When a writer releases the lock, updates are only returned to the server if another client is waiting.

Release-invalidate: In this and other release policies, readers are allowed to continue reading when a writer begins. Once the writer is done, the changes will be written back to the server immediately, without waiting for a request. When the changes arrive at the server, the current readers are invalidated.

Release-update: This policy is similar to the previous one except that, when a writer sends changes back to the server, the changes are forwarded to each of the readers rather than sending invalidation messages. If a client

wishes to stop receiving updates, they must explicitly notify the server. No version numbers are required in this policy. *Release-update* reduces the number of messages required, provided that the object has a relatively stable sharing pattern, particularly if a frequent reader, rather than a frequent writer, forms the critical path. Munin's *reduction* policies may essentially be special cases of this policy.

Release-request: This is identical to *access-request* except that update requests are not blocked if there is a write occurring. The reader requesting the update will simply be given the last version of the object. This allows quicker responses to update requests but may be undesirable in some circumstances.

Multiple-writer-invalidate: In the multiple writer policies it is up to the user to keep writers from stepping on one another's toes. These policies might be thought of as release rather than access policies as writers will not block anyone when they begin and writers will write-back their changes as soon as they are done writing without waiting for a request from the server.

Multiple-writer-update: This is the same as the previous policy except that updates are sent to everyone rather than invalidates. As in *release-update*, clients must explicitly tell the server when they no longer want updates. Munin's *producer-consumer* and *write-shared* policies are similar to this, although the latter appears to perform updates immediately upon each write, rather than waiting for a release of the write lock.

Multiple-writer-request: This allows anyone to retain out of date copies. All clients must explicitly request updates. Munin's *result* policy appears to be a special case of this in which only the reader

4.3 Multi-threaded Clients

In writing distributed programs, it may be desirable to have multiple threads in a single client process group which share the same local copy of an object. Therefore, Dishrag offers concurrency protection mechanisms within the client process to ensure that multiple threads don't violate the lock guarantees. However, implementing the same protocol interactions at the client level as occur at the server level would be exceedingly complex and would require that multiple copies of an object be made within the client process. This would change the user's programming model more than we would like because the user could no longer retain references to their shared object except through the control object.

Therefore, only access synchronization is supported by the client-level code. If a user wishes to have multiple threads in the same process interact with an object using exclusive, release, or multiple-writer synchronization, they must have those threads separately subscribe to the object. The threads will then have their own copies and will interact through the server. When the server sends a client an update, that update must wait until all client threads release their locks on the object before it can perform. Because the server need not wait for this to occur, a new thread is spawned on the client to perform an update and the server may continue after only a short delay.

5.0 Testing and Evaluation

We have evaluated Dishrag in a number of ways. We first present timing measurements of each of the basic network operations used by the system. Then we will describe several illustrative sharing patterns that might occur in a Dishrag application and compare the performance of each of the protocols on the patterns, analyzing their strengths and weaknesses in practice. Finally, we describe three useful applications that have been implemented using Dishrag.

TABLE 1. Time in milliseconds to perform simple Dishrag operations

Operation	Time
Minimal RMI call	2.7
Publish an object	75.6
Subscribe to an object	18.5
Obtain an empty update	21.4
Obtain a 1KB update	32.1
Obtain a 2KB update	41.7
Obtain a 3KB update	52.3
Send an invalidate	4.6
Send an empty update	9.8
Publish a barrier	31.1
Subscribe to a barrier	9.0
Wait and break a barrier	3.4

5.1 Component Benchmarks

Table 1 shows the time in milliseconds required to perform various operations. Timing was evaluated on an Ultrasparc 1 client communicating with 266 MHz Pentium II server attached to different ethernets within the same building. Dishrag uses Java's built-in remote method invocation (RMI) procedures for all communication. Although each RMI call appears to create and destroy a TCP connection and spawn a new thread in the receiver, it is still relatively quick compared to the overhead of the other basic procedures.

Subscribing to an object is no more costly than obtaining an update, although this subscribe does not include the cost of the necessary first update. The row labelled "obtain an empty update" is the cost for the client to send a read request to the server and receive back the object updates when there are actually no updates to be returned. There is quite a bit of overhead in obtaining an empty update. This is partially due to the cost of grabbing locks in the server and partially due to the fact that, in the current implementation, the server returns an "updater" object even when that object contains no update information. This could be optimized by returning nothing when an update is unnecessary. However, this would have relatively little effect in practice as an empty update will only actually be sent if the user acquires and releases a write lock without writing to the object.

It appears that each additional 1KB of data in an update message costs roughly 10 milliseconds per update. The next two rows of the table list the cost for the server to push either an invalidate or an update to a busy client. The invalidate is less than twice the cost of a basic RMI call. The update is a bit more costly because it must wait for the client to spawn a new thread before it can return. The time necessary to wait on a barrier (where no waiting is actually done because the barrier breaks immediately) is only slightly greater than an RMI call. This is important because barrier waits are quite frequent in many applications and it is often the thread on the critical path that breaks the barrier.

5.2 Sharing Patterns

In order to evaluate the sharing protocols and illustrate their performance characteristics in relatively controlled situations, we implemented four small testing programs. Each program contained a server which spawned four

client threads on different machines. We measured the average time to complete each program using each of the protocols other than read-only.

In Program 1, a single 1KB object was used. One client acted as the writer, repeatedly acquiring a write lock, writing to the object, and releasing the lock. The other four clients acted as intermittent readers. Each one would read the object an average of once for every ten writes. Barriers were used to synchronize the clients so that readers were aware of the number of writes that had occurred. Figure 1 shows the average time required to complete 100 iterations of the loop using each of the protocols.

Of primary interest in reading these charts is the comparison between access-invalidate, release-invalidate, multiple-writer-invalidate, and release-update. This first sharing pattern is best suited for access policies. Because reads only occur following approximately 1/3 of the writes, there is potential for the writer to buffer multiple writes. This is taken advantage of by the exclusive and access-policies. Access is slightly better than exclusive as readers do not interfere on those occasions when they both wish to access the object. Although it doesn't do write buffering, multiple-writer-invalidate is surprisingly good here. Although the number of messages sent in this sharing pattern is the same for it and release-invalidate, multiple-writer-invalidate is faster because there is less overhead in the server. In the release policy, all read and write requests that reach the server must get on a queue before they can complete. Because multiple-writer provides less synchronization control, it lets these requests go straight through.

The update policies are particularly bad for this sharing pattern because most of the updates are unnecessary. Two versions of each request policy are shown. The eager version requests an update on every read and the lazy version only requests an update on 10% of the reads. Clearly the lazy versions perform well because they require few updates, with multiple-writer-request the fastest overall. Interestingly, the eager versions are still quite good. In fact, for this sharing pattern, each eager request policy should be faster than the corresponding invalidate policy because they do exactly the same number of operations, but the request policies avoid sending any invalidates. The performance of eager request policies would quickly decline if the sharing pattern involved clients performing multiple reads between each write.

Program 2 was identical to Program 1 except that readers performed a read following every write, not just 10% of the writes. The results are shown in Figure 2. Now the benefit of the access mode disappears because there is never an opportunity for buffered writes. Because it doesn't allow concurrency between the writer and the readers, access-invalidate is now worse than release-invalidate. Although exclusive did quite well on Program 1 because the sparse reads caused little interference, that is not the case in this program. The multiple-writer policies continue to outperform release because of their lower overhead. Although the update policies are not performing extraneous updates, they still do poorly in this situation because the updates are performed on the critical path while the writer is waiting to release its lock. Again the eager request policies did quite well because they avoid invalidates, but multiple-writer-invalidate turned out best overall, aside from the lazy patterns.

Rather than one writer with multiple readers, Program 3 involves multiple writers with a single reader. On each iteration of the loop, each of the writers writes to a separate object while the reader sits idle and then the reader reads from all of the objects. In this case, the exclusive and access policies do poorly because there is no opportunity for buffering and the writers block out the reader unnecessarily. The release and multiple-writer invalidate and request policies are better because they allow the writers to begin the next iteration while the reader is still reading. However, the benefit of the update policies is finally revealed. The cost of performing the updates is now mostly paid in parallel by the writers rather than on the reader's critical path. The update policies even outperformed the lazy request policies in which the reader was only accessing each object 10% of the time.

Like Program 3, the final illustrative program had four writers and a single reader. However, in this case the writers were all writing to the same object. As expected, the results, shown in Figure 4, indicate that the multiple-

writers invalidate and request strategies were superior. Of note is the poor performance of the release consistency policies. In access-invalidate, each writer must send a single message invalidating the previous writer and obtaining their updates. However, in release-invalidate each writer pushes their updates to the server and then the next writer must still send them an invalidate. The update policies result in pathological behavior on this sharing pattern. Not only is each writer unnecessarily sending updates to the reader, they are sending them to other writers as well. Multiple-writer-update takes a bit under a third the time of release-update because most of this occurs in parallel, but it is still four times more costly than multiple-writer-invalidate.

Aside from confirming most of our initial expectations, these analyses have lead to a number of interesting realizations. Access-request is a good, safe choice for a single-writer/multiple-reader policy. Its ability to buffer writes means that it will rarely be doing detrimental amounts of extra work, as a release policy might with many repeated writes. On the opposite end is the update policies. There can be situations in which update policies are optimal, but the benefit over an invalidate policy is at best small. The cost of using an update policy in the wrong situation, however, can be considerable.

An unintuitive finding is that a multiple-writer policy may actually be desirable even when only a single writer is ever active. Often, the user will be using barriers or other control mechanisms to enforce a particular pattern of access. In this case, the safeguards provided by single-writer policies are unnecessary. Because of their low overhead, the multiple-writers policies may be more efficient alternatives. Similarly, although the request policies were intended for relatively rare situations in which out-of-date copies are acceptable, our tests revealed that it may be beneficial to replace an invalidate policy with a greedy request policy that always seeks an update. Provided that the sharing pattern is known not to contain two reads in a row, this will incur no extraneous updates but will avoid any invalidate calls.

5.3 Practical Applications

5.3.1 Parallel Matrix Multiplication

In order to examine the usefulness of Dishrag for computational sub-problems, we implemented a simple matrix multiply program that will read in two matrices, decide on an optimal way to split them into two sub-problems, and then will run the sub-problems as new threads on arbitrary machines. This application does not require any amount of information sharing, in the sense that the sub-problems are independent and the parent process is just waiting for the spawned children to complete. The depth of problem splitting is a settable parameter. For purposes of the analysis, it is easiest to just examine the case of depth one, a single split, compared to no splitting. The objective in the splitting is to minimize the amount of data being transferred over the network while keeping the problem sizes for each part equal. The three ways to split the problem and their costs are shown below for matrices X and Y of size (x, y) and (y, z).

TABLE 2. Matrix Splitting and associated network cost of sub-problem splitting

Split Pattern	Size of Sub-problem Input	Size of Sub-problem Output
$\left[X_l \mid X_r\right]\left[\frac{Y_t}{Y_b}\right] = \left[X_l Y_t + X_r Y_b\right]$	$(xy)/2 + (yz)/2$	xz
$\left[\frac{X_t}{X_b}\right] Y = \left[\frac{X_t Y}{X_b Y}\right]$	$(xy)/2 + yz$	$(xz)/2$
$X\left[Y_l \mid Y_r\right] = \left[XY_l \mid XY_r\right]$	$xy + (yz)/2$	$(xz)/2$

Basically, the first split will be used for roughly square matrices, but the bottom two methods will be used for splitting rectangular matrices. The following table presents the timing for several various runs of this program. All times are reported as the average of five runs. The Null problem is one that reads in the files and outputs a result file, to mimic all I/O operations done except for those explicitly involved in the matrix multiply. In the Single Machine, Split once condition, three threads are run on a single machine, but each is running with its own copy of the shared objects and no actual memory sharing occurs. In the Two Machines, Split Once condition, the first sub-problem is run on the local machine but as before it obtains its own copy of the object from the server. In the Three Machines, Split once, the server reads in the data and launches worker threads on two separate, machines, waiting for them to complete and then forming the final result. The second table immediately has removed the time for the Null problem from the remaining.

TABLE 3. Timing for various size matrix multiplies and sub-problem splitting

Matrix Sizes	Time for Null Problem	Single Machine, no Splitting	Single Machine, Split once	Two Machines, Split once	Three Machines, Split once
(40, 80, 30)	5.99	8.51	9.95	10.39	10.92
(50, 40.75)	6.35	9.48	11.15	11.5	10.7
(100, 100, 100)	18.39	27.35	31.28	27.05	27.76
(200, 180, 200)	60.85	98.48	103.4	94.53	93.51

Matrix Sizes	Single Machine, no Splitting	Single Machine, Split once	Two Machines, Split once	Three Machines, Split once
(40, 80, 30)	2.52	3.96	4.40	4.93
(50, 40.75)	3.13	4.8	5.15	4.35
(100, 100, 100)	8.96	12.89	8.66	9.37
(200, 180, 200)	37.63	42.55	33.68	32.66

The most immediately observation to make it that for small problems, we pay too much of a penalty in creating sub-problems and launching new threads to improve total elapsed time measures. For big problems, we do see an improvement, but it is clear that we are paying a big penalty for splitting the problem onto multiple machines. It appears that the cost of preparing and moving the objects across the network is actually prohibitive for anything other than large problems. Given the time required by the application code just to read in the matrices from disk, which was typically twice as long as it took to compute the answer, this is not terribly surprising.

One final result that shows large sub-problems do induce a savings if run in parallel was done on a pair of (300, 300) matrices. The time to run the problem on a single machine was 271s, and the time to run it on three machines was 124s, without including the Null problem time of 150s.

5.3.2 Parallel Sort

We also implemented a parallel sorting program, which would read in a file to wort, split the work up among an arbitrary number of machines. Each machine would perform a merge sort, and the results would then be merged back at the initial server. Unfortunately, we have not gotten this program to work on large lists, and it appears to be a bug in the Java Virtual Machine that is causing object serialization to fail. The strings to sort are implemented as linked lists, whereas the above matrices were actually arrays. Transforming this problem into one which uses arrays, however, would incur a extra copying costs above and beyond everything introduced by our system. So, although it is implemented, we were unable to obtain any meaningful numbers for this application. On the bright side, it took only about an hour and a half to write and debug from scratch.

5.3.3 Parallel Theorem Proving

We also implemented a simple distributed propositional theorem prover using Dishrag. This system uses goal directed search to attempt to prove queries from a database propositional hereditary harrop formulas.

The theorem proving program, called gds (Goal Directed Search), parallelizes its computation by maintaining a queue of subgoals to be proven. These subgoals may be dequeued and solved by a pool of worker threads, which simply repeatedly dequeue a goal from the queue and attempt to solve it, writing back the result to a shared object in which the creator of the subgoal can monitor. In the process of solving a subgoal more subgoals may be found which are in turn placed upon the work queue.

Figure 5 summaries the results of our tests of gds on Dishrag. We asked gds to solve large theorems (approximately 200 connectives and base propositions) from a large database (approximately 200 connectives and base propositions). We tested gds proving the same theorems using one, two, three, and four machines. As the table shows, the fastest times were achieved using only a single processor. This is most likely due to the fact that gds does not encode its database or queries in such a way that sharing can be realized between different subgoals, forcing Dishrag to constantly retransmit the same data. Also gds makes no attempt to choose the appropriate

granularity of subgoals to submit to the work queue. Instead, anytime that it finds a conjunction of goals to be solved it makes one of them into a task to be put on the work queue. A more successful approach might be to only submit those subgoals which are sufficiently large, thereby amortizing communication cost.

6.0 Conclusions and Future Work

Dishrag implements a useful set of object sharing and synchronization cross-platform primitives that extend the Java core language. The idea of sharing entire objects as opposed to memory segments is needed in Java in order to implement useful parallel applications. We have also shown that the primitives for locking and synchronization are relatively inexpensive, but the cost associated with updating an object over the network is expensive. At the earliest design stages, we had chosen to use Java's serialization primitives for data shipping, and it appears as if the serialization of an object is a bottleneck in our tested applications. The synchronization and locking methods also use serialization, but in those instances, the actual amount of data being shipped over the network using serialization is minimal, and in fact quite similar to RPC. We were wary of relying on the network too much, but that ended up being relatively small cost compared to the overhead in preparing to send an object over the network.

Currently, our pre-processor that tags updates on an object's fields is very simple, and several optimizations could be made to it. Consecutive updates should be coalesced when possible into a single update, and updates should be moved outside of loops. Tt is also possible to do away with the pre-processor entirely through modifications of the underlying virtual machine. Considering the current cost we are paying to serialize the object, it is not clear that the cost introduced by the tagging of updates as they occur is prohibitive in practice.

A second improvement would be to reduce our reliance on the serialization primitives. However, this would also require changes to the Java virtual machine. Currently, object serialization occurs to an object that is a TCP stream, but there are still several layers of indirection between the code that actually can serialize an object and the code that writes to the stream. To make the shipping of an object faster, a more monolithic approach might be necessary.

7.0 Related Articles

The Charlotte project has implemented distributed shared objects for java as part of a metacomputing package for the web. In contrast to Dishrag, they track changes to objects by requiring users to access fields throw special get and set methods. This means that users cannot use the language in the familiar way, using =, ++, +=, etc.

Charlotte does not attempt to track changes to individual fields in an object; instead, once any one field is set, the entire object is marked dirty and must be shipped to update other users copies. Dishrag has potential to be more efficient in cases where large object are shared, but only small portions of them are modified. Charlotte also only implements one simple sharing policy, where Dishrag provides a range of policies, similar to Munin.

Munin provides multi-protocol distributed shared memory. Unlike Dishrag, it shares data on a page level, and not on an individual object level. This means that performance can be hindered by "false sharing", that is, data which is treated as dirty simply because it is on the same memory page as data which has been modified, even though it has not itself been modified. On the other hand, in order to avoid ever incorrectly marking code as dirty, Dishrag is forced to insert code to track changes to each field. This may impose an unacceptable performance overhead in some situations.

Fighting Fire with Truth: a Concurrent Transactional Truth Maintenance System

Michael Mateas and Kamal Nigam

1 Introduction

A Justification-based Truth Maintenance System (TMS) is an AI component used by a problem solver to cache inferences. Traditionally, TMSs are tightly coupled to a single problem solver and are single-threaded. The problem solver issues one request at a time to the TMS and is the only process issuing such requests. In this paper we describe a concurrent, transactional TMS, which supports multiple simultaneous requests from multiple sources.

This project intersects with system issues discussed in the course, specifically threads (concurrency) and transaction processing. Throughout this paper, we will reference specific ideas from the course as they arise.

This paper will first describe TMSs and motivate the usefulness of a TMS that supports concurrency. Then we will describe the design and implementation of our transactional TMS. Next we'll present our evaluation methodology, including the design of a fire model to generate test data. Finally, we'll present and analyze the evaluation results.

2 Truth Maintenance Systems

A TMS caches inferences made by an inference engine and uses the cached inferences to make conclusions more quickly than the inference engine could if it had to remake the conclusion from scratch. The TMS is able to process cached inferences quickly because it only handles propositional knowledge, while the inference engine is handling some arbitrary reasoning model (e.g. first order logic, model based reasoning). In the case of first order logic, the inferences cached in the TMS refer to specific variable bindings.

For example, an inference engine may have the rule

$$\text{On\_Fire}(X) \wedge \text{Combustibles}(X) \wedge \text{Next\_To}(X, Y) \wedge \text{Next\_To}(Y, Z) \quad \Rightarrow \quad \text{At\_Risk}(Z) \tag{1}$$

meaning that if a location is ever on fire, and there are combustibles (e.g. greasy rags) at that location, then any location a distance of two locations away from this one is at risk. This rule applied to the facts On_Fire(a), Combustibles(a), Next_To(a, b), and Next_To(b, c) would be used to derive the fact At_Risk(c).

The problem solver would then ask the TMS to create nodes representing these facts and to connect these nodes together in a justification network. The justification network for this particular inference would consist of a DAG with directed arcs leading from On_Fire(a), Combustibles(a), Next_To(a, b), and Next_To(b, c) to At_Risk(c). Now, if in the future one of the antecedents (say On_Fire(a)) became "out" (TMS jargon for becomes false or unknown) then At_Risk(c) would automatically become out without the inference engine having to do any work. As the chains of inference become longer, the TMS saves more inference work (assuming that the truth values of facts used as justifications by cached inferences change, allowing an inference to be reused).

Traditional TMSs are not concurrent. A single problem solver interacts with the TMS. Only one operation (e.g. changing an assumption, adding a justification structure) is active in the TMS at any one time. A concurrent TMS would allow multiple processes to simultaneously update truth values in the TMS. Assuming the justification graph is bushy, such concurrency would be a win, since conflicts would be relatively rare. Yet these updates must be made transactionally in order to maintain consistency in the TMS. For example, a process should never see the state where a precondition is out (e.g. On_Fire(a)) but the antecedent is still in (e.g At_Risk(c)).

3 Design and Implementation

This section describes the design and implementation of the components making up our system.

3.1 TMS

The TMS maintains a directed graph of nodes, where each node represents a propositional fact. There are four node types: premise, assumption, and "regular". Premise nodes are always in; assumption nodes can be set in or out by an external process; and regular nodes have their value set by justification structures.

The primary functions provided by the TMS are:

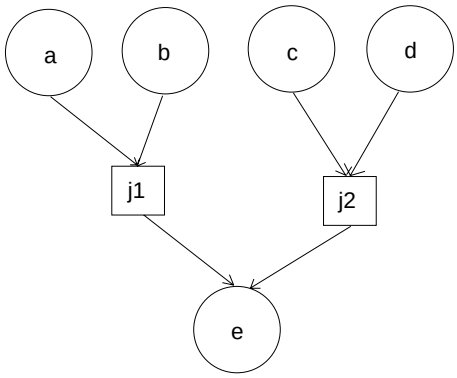

Figure 1: TMS justification

- `enable-assumption(node)`: Make an assumption node in.
- `retract-assumption(node)`: Make an assumption node out.
- `in-node?(node)`: Returns true if a node is in.
- `out-node?(node)`: Returns true if a node is out.
- `tms-create-node(datum, assumptionp, contradictoryp)`: The basic operation to create a new node in the TMS. The arguments assumptionp and contradictoryp are set appropriately to create an assumption or contradiction node. If both of these arguments are false, then a normal node is created.
- `justify-node(informant, consequent, antecedents)`: Creates a justification structure in the TMS (draws in the arcs of the dag). Informant is the node corresponding to the rule justifying an inference. The consequent is the node representing the fact concluded by an inference. The antecedents is a list of the nodes representing the facts used by the rule to infer the consequent.
- `supporting-justification-for-node(node)`: Returns a description of the nodes which justify (are the antecedents of) of a node.

The TMS and transaction manager (described below) follow an object oriented design. The primary objects in the TMS are nodes, justifications, and the TMS object itself. Justifications are the objects that connect antecedent nodes to a consequent node. The TMS object stores global lists (e.g. list of all nodes, list of all assumptions), which support dumping global state.

3.2 Transaction Manager

The transaction manager provides atomicity (commit and abort), isolation (two phase locking) and consistency, but does not provide durability (recreation of state in the face of system failure).

3.2.1 Impossibility of avoiding deadlock

The transaction manager provides deadlock detection in the form of timeouts. Transactions which timeout on a lock are aborted. We are not able to avoid deadlocks through a lock ordering scheme because of the the inherent function of TMSs. Consider the justification graph in Figure 1.

Node `e` has two justifications, `j1` and `j2`. Suppose nodes `a` and `b` are currently in and `j1` is the current justification of `e`. If `a` is set out, then `e` becomes out (its current justification is no longer satisfied). At this point, the process that set `a` out would have two writelocks on `a` and `e`. After setting `e` out, the TMS begins looking for an alternative support for `e`. In this case, there is a second justification `j2`, which, if its antecedents are satisfied (are in), would provide a support to make `e` back in. To check whether the antecedents `c` and `d` are in, the process has to readlock them. If they were both in, `e` would be set back in and the transaction would complete. Notice that even in this simple example, a transaction which looked like a single operation at the requester level (make the assumption node `a` out), resulted in four operations in the TMS and the acquisition of two writelocks and two readlocks. Now suppose that after the

process had acquired a writelock on `a` but before it had acquired a readlock (checking the alternative justification) on `c`, another process writelocks `c` to change its value to out. The outness begins propagating down towards `e`. Process 2 needs to acquire a writelock on `e` to finish the propagation but process 1 holds a writelock `e`. Process 1 is waiting to acquire a readlock on `c` to check if `j1` is able to serve as support for `e`, but process 2 holds a writelock on `c`. The two processes are deadlocked. In general, a process changing an assumption node will acquire locks "down" from the assumptions as it propagates assumption node changes and back "up" towards the assumptions as each node that went out looks for alternative support. This pattern makes it impossible to establish a partial order on lock acquisition.

3.2.2 Locking scheme

We employ standard readlocks (shared) and writelocks (exclusive). In addition, we allow the holder of a readlock to promote it to a writelock. Our promotion policy is adapted from [Gray79] and [Bayer76]. When the holder of a readlock asks to promote it, if there are no other holders of readlocks they are granted an immediate promotion. If there are other readlocks, the promoter is placed at the front of the lock grant queue. Once the current readers finish, the promoter will be granted their writelock before any other waiting readers or writers receive the lock. When a process requests a lock promotion, if there is already someone waiting to promote on the lock, the requesting process aborts. If promote waiter queues longer than one were allowed, then, when the first promoter acquired a writelock, all the other promote waiters would have effectively released their readlocks in the middle of their transactions, breaking two phase locking, and thus losing serializability.

We support lock promotion because of the hidden pattern of lock acquisition that occurs in the TMS. A process may be issuing a transaction consisting of only two assumption updates (to different nodes). From the point of view of the transaction issuer, it is only acquiring locks on two different nodes. But because the TMS does truth propagation and searches for alternative support, the transaction is actually acquiring many read and writelocks. Readlocked nodes acquired during the first update (searching for support), may need to be written during the second update (value propagation). At this point the transaction needs to promote locks.

An alternative design we considered is to eliminate readlocks entirely; only writelocks are supported. This simplifies locking logic, and possibly lowers the number of aborts (no aborts due to multiple promoters). But it has the obvious drawbacks of decreasing the amount of concurrency and increasing the amount of copying between transaction scratch space (shadow nodes; see below). A minor optimization is to acquire read and writelocks if a transaction consists of only a single TMS operation but to acquire only writelocks if it consists of multiple operations. Because of the decrease in concurrency, we decided to implement the lock promotion strategy.

The listener, which looks at incoming requests and dispatches them to the TMS (see below), uses a hash table to map names of nodes (which is how requester refer to nodes) into node pointers (which is how the TMS refers to nodes). Since the hash table needs to be updated whenever a new node is created, the hash table must also be read and writelocked. The granularity of locking in the hash table is individual hash buckets. Since hash buckets are meta-data (used only to get to the real data), the lock strategy on buckets doesn't have to obey two phase locking [Bayer76]. Whenever a readlock is acquired on a bucket (to map a name to a pointer), the lock is immediately released when the node is found. This decreases spurious contention between readers and writers on hash buckets.

The lock logic is localized in a class called Tlockable. If an entity wants to be lockable, it inherits from this class. In our system, TMS nodes, hash buckets, and global node and justification lists (in the TMS proper) are all lockable entities. We adapted the locking algorithm from [Birrell93] to support lock promotion.

3.2.3 Transaction state

When a transaction begins, a transaction record is allocated and a transaction ID (a pointer to this record) is returned to the caller. The transaction record holds transaction state for the duration of the transaction. This state includes:

- `thread`: Identifier of the thread processing the transaction.
- `readlocked_nodes`: List of TMS nodes readlocked by the current transaction.
- `shadow_nodes`: List of records associating a shadow node with a TMS node (list of writelocked nodes).
- `new_nodes`: List of new nodes created during this transaction.
- `new_justs`: List of new justifications created during this transaction.
- `jtms_node_lists_to_change`: List of write intentions to node lists maintained in the TMS.
- `jtms_justs_to_change`: List of write intentions to justification lists maintained in the TMS.

- `shadow_buckets`: List records associating a shadow hash bucket with a hash bucket (list of writelocked buckets).
- `readlocked_bucket`: The bucket currently readlocked by this transaction (a transaction can only have one bucket readlocked at a time).

3.2.4 Shadow objects

In order to support atomicity, all write changes are written to shadow objects. On successful commit, shadow objects are copied back onto real objects and the shadows are deallocated. On abort, shadow objects are deallocated with no copy.

An instance of a shadow of an object is a member of the class of that object. For example shadow hash buckets are members of class hash bucket and shadow TMS nodes are members of class TMSnode. The fact that shadow and real objects are the same class means that the same code that manipulates a real object can manipulate a shadow. This design choice made it easy to convert our non-transactional TMS (which we wrote first) into a transactional TMS. For example, consider the following two code fragments. The first code fragment is the non-transactional TMS node method to enable an assumption. All references to identifiers not declared in the method are to class members.

```
// Makes an assumption in.
void tmsnode::enable_assumption()
{
  if (assumption == F) throw TMSNODE_enabled_non_assumption();
  if (label == OUT) {
    make_node_in(NULL);
    if (my_jtms->debugging == T)
      printf("Propagating IN from %s\n", name);
    propagate_inness();
  }
  else if (support != NULL) support = NULL;
}
```

The second code fragment shows the same method modified to be transactional.

```
// Makes an assumption in.
void tmsnode::enable_assumption(tid_t tid)
{
  tmsnode* shadow_node = Twritelock_node(this, tid);
  if (shadow_node->assumption == F) throw TMSNODE_enabled_non_assumption();
  if (shadow_node->label == OUT) {
    make_node_in(NULL, tid);
    if (my_jtms->debugging == T)
      printf("Propagating IN from %s\n", name);
    propagate_inness(tid);
  }
  else if (shadow_node->support != NULL) shadow_node->support = NULL;
}
```

Notice that now all TMS node methods take a transaction ID as an argument. At the beginning of the routine, the node is writelocked. Twritelock_node returns a pointer to a shadow node. All references to data members now occur through this pointer. Other than that, the code stays the same. If the method was readlocking instead of writelocking, readlock would return a pointer to the real node (assuming that the node hadn't been previously writelocked in the same transaction). The code that references data members doesn't care if the pointer is to a real or shadow node, since shadows and reals are the same type.

Updates to the TMS global lists are not handled via shadows. Use of shadows for these lists would involve an inordinate amount of copying. For the global lists, a list of write intentions are maintained instead. Since deletion o renaming of TMS nodes is not a supported functionality, this implementation is not problematic.

3.2.5 Use of transaction state

The transaction state is used every time an object is readlocked or writelocked, every time a new TMS node or justification is created, and on commit and abort. In this section we will describe how the transaction state is used for each of these operations.

When an object is readlocked, the transaction manager first checks whether the object has been previously readlocked in this transaction. If so, readlock returns a pointer to the object itself. Next, the transaction manager checks whether this transaction already holds a writelock for this object. If so, a pointer to the shadow object is returned. If the object is neither read nor writelocked, then the object is readlocked, the object is added to the readlock list, and a pointer to the object is returned.

When an object is writelocked, the transaction manager checks whether the transaction already holds a writelock on this node and if so returns the shadow pointer. Next it checks if the object is readlocked by this transaction. If so, it promotes the lock, removes the object from the readlocked list, creates a shadow copy of the object, adds the real object and associated shadow to the shadow list, and returns the shadow pointer. If the object is neither read nor write locked, the object is writelocked and the same shadow creation as in the case of promotion is done.

When a new TMS node or new justification is created, the new object is registered with the transaction. The newly registered object is placed in the new object list. These lists are used for abort cleanup.

On commit, the contents of all shadow objects are copied onto real objects, the locks are released, and the transaction record is cleaned up.

On abort, the locks are released and the transaction record is cleaned up (including deleting newly created objects that have been registered with the transaction). Finally, the thread associated with the transaction is killed.

3.3 Listener

The listener module handles concurrent incoming requests and manages a thread pool to perform operations inside the TMS. The primary thread listens on the main socket for new sensor connections. Upon connecting, it submits a request to the thread pool to handle the new connection. Some thread from the pool will then read the stream of transactions from the connection. It will request TMS actions to be performed by the thread pool. Thus, every sensor uses two threads in the TMS – one to process the request, and one to do the actual work. We use this duality approach to handle aborted transactions. When a transaction is aborted because of potential deadlock, the worker thread is killed after all its locks are released to stop it from progressing further on its work. The killed thread is replaced with a new thread, so the pool does not dry up. To maintain communication with the sensor, the thread that works cannot be the thread that communicates. To synchronize, they use signals to pass back return values of successful and unsucessful work.

3.4 Sensor

The sensor module is a simple program that reads in a trace file generated by our simulator and sends it to the TMS in transaction-sized chunks. If a transaction succeeds, it proceeds normally. In the event of an abort, the process delays for a random amount of time on the order of tens of milliseconds to allow the conflicting process to complete, and tries again. On successive aborts, the potential delay length grows linearly. The sensor tracks transaction latency, total throughput and abort statistics over the course of a run. These numbers are used in the evaluation process.

4 Evaluation Methodology

4.1 Evaluation Criteria

Our primary goal in evaluating the concurrent TMS is to demonstrate that the addition of concurrency is beneficial to the performance of the system. Our primary measure of performance will be throughput, measured in terms of sensor updates processed per second. Sensor updates form the core and bulk of operation that drives the system, and thus, we hope to minimize the latency of these operations while providing high concurrency.

We hypothesize that as we increase the number of concurrently running sensor threads, throughput will increase up to a point. Slack in the system due to network overhead, paging of virtual memory to and from disk, and other sources is time lost to a single sensor. In concurrent implementations, this time can be recaptured by other sensors, especially if data locality prevails, and each sensor can work independently. Eventually, however, too much concurrency will slow the system. With more sensors, the chances of locking conflicts grows, and threads will be increasingly required to wait for one another. Overhead costs of managing multiple threads will also slow the system. We hypothesize that

there will be dramatic gains in throughput as we increase the number of sensors, and then a slow decay in throughput as we push on system limits and data locality.

As explained in section 3.2.1, deadlock can occur in a concurrent TMS if nodes that share consequences (under transitive closure of justifications) are simultaneously updated. In the fire scenario, facts associated with the same location will share many consequences. Facts associated with locations next to each other will share some consequences. As noted in section 4.3, as the distance between locations grows, it becomes more and more unlikely that they will share any consequences. We measure the effect of locality of updates on TMS performance by measuring throughput and the number of aborts for a number of traces with different locality. We hypothesize that as concurrent TMS updates become more spatially separated (relative to the fire model), peformance (both in throughput and number of aborts) should improve.

We also measure the cost of transactions. There is significant overhead in providing atomicity and consistency, because partially completed work is kept separate from committed work. We will quantify the cost to the system for a single sensor updating the TMS. It is not relevant to test the cost of transactions in a multiple-sensor scenario, because isolation is lost without transactions.

4.2 Fire Model

To motivate the necessity of a real-time, concurrent, consistent TMS, we have simulated the problem domain of fighting fires on a ship. We include diagnosis, prevention, and proactive recovery from fires. Consider a ship with a multitude of remote sensors in different locations. These sensors are heterogeneous in type, such as smoke detectors, airflow detectors, temperature sensors, and voice and motion sensors to detect the presence of people. These sensors, or their agents update the TMS in real time with their values. Working in parallel, consider the existence of automatic inference engines that combine evidence from sensor values to determine higher-level states of the world. For, example, if the smoke detector goes off in the bathroom, it may deduce that someone is sneaking an illicit cigarette, but if the smoke detector goes off and the heat detector is high, then there is a fire in the bathroom. These inference engines supply conclusions to the TMS, with their antecedents, and the TMS takes responsibility for updating the conclusions as the sensors change.

In addition to fire diagnosis for the ship, the inference engines recognize situations where it is appropriate to take action to combat the fire and prevent its spread. For example, if there is a fire, we automatically turn on the sprinkler system. These actions are also the result of inferences made, and the TMS is responsible for ordering their execution whenever such action-level concepts become valid. As another action example, the TMS can also take irreversible actions, such as shutting unopenable fire doors. It is imperative that we do not needlessly lock people into locations from which they cannot escape. Because of this, it is necessary that the TMS maintain consistent state between the sensors, higher-level analysis, and actions it orders in the world. Yet, it must provide this consistency in the face of real-time sensor updates and high concurrency.

Our domain model of the first-order inference rules used are shown in the figure below. To take these inference rules and apply them to a fire scenario, we built a simulator to generate sensor readings appropriate for a fire, and to generate propositional instantiations of the inference rules as appropriate. The simulator uses a probabilistic model of fire progression. At each discrete time step, the simulator updates the state of each location. Fire spreads to adjacent locations probabilistically, based on the presence of combustibles, airflow, and the activation of sprinklers, and the open or closed status of firedoors between locations. The duration of a fire at a location depends probabilistically on the presence of combustibles, and the duration of sprinkler activation in the location.

$$\text{Smoke}(X) \wedge \text{Temperature\_Med}(X) \Rightarrow \text{On\_Fire}(X) \quad (2)$$

$$\text{Temperature\_Hi}(X) \Rightarrow \text{On\_Fire}(X) \quad (3)$$

$$\text{Temperature\_Med}(X) \wedge \text{Combustibles}(X) \Rightarrow \text{On\_Fire}(X) \quad (4)$$

$$\text{Smoke}(X) \wedge \text{Combustibles}(X) \Rightarrow \text{On\_Fire}(X) \quad (5)$$

$$\text{Smoke}(X) \Rightarrow \text{At\_Risk}(X) \quad (6)$$

$$\text{Temperature\_Med}(X) \Rightarrow \text{At\_Risk}(X) \quad (7)$$

$$\text{On\_Fire}(X) \Rightarrow \text{At\_Risk}(X) \quad (8)$$

$$\text{On\_Fire}(X) \wedge \text{Next\_To}(X, Y) \Rightarrow \text{At\_Risk}(Y) \quad (9)$$

$$\text{At\_Risk}(X) \wedge \text{Airflow}(X, Y) \Rightarrow \text{At\_Risk}(Y) \quad (10)$$

$$\text{On\_Fire}(X) \wedge \text{Combustibles}(X) \wedge \text{Next\_To}(X, Y) \wedge \text{Next\_To}(Y, Z) \Rightarrow \text{At\_Risk}(Z) \quad (11)$$

$$\text{Motion\_Sensor}(X) \Rightarrow \text{People\_At}(X) \quad (12)$$

$$\text{Voice\_Sensor}(X) \Rightarrow \text{People\_At}(X) \quad (13)$$

$$\neg\text{Motion\_Sensor}(X) \wedge \neg\text{Voice\_Sensor}(X) \Rightarrow \neg\text{People\_At}(X) \quad (14)$$

$$\text{At\_Risk}(X) \wedge \text{People\_At}(X) \Rightarrow \text{Sound\_Alarm}(X) \quad (15)$$

$$\text{On\_Fire}(X) \wedge \text{Sprinkler\_At}(X) \Rightarrow \text{Activate\_Sprinkler}(X) \quad (16)$$

$$\text{At\_Risk}(X) \wedge \text{Sprinkler\_At}(X) \wedge \text{Combustibles}(X) \Rightarrow \text{Activate\_Sprinkler}(X) \quad (17)$$

$$\text{Airflow}(X, Y) \wedge \text{At\_Risk}(X) \wedge \text{Ventilation}(X) \wedge \neg\text{People\_At}(X) \Rightarrow \text{Close\_Ventilation}(X) \quad (18)$$

$$\text{On\_Fire}(X) \wedge \text{Ventilation}(X) \wedge \text{Airflow}(X, Y) \Rightarrow \text{Close\_Ventilation}(X) \quad (19)$$

$$\text{Fire\_Door}(X) \wedge \text{At\_Risk}(X) \wedge \neg\text{People\_At}(X) \Rightarrow \text{Close\_Fire\_Door}(X) \quad (20)$$

$$\text{Fire\_Door}(X) \wedge \text{On\_Fire}(X) \Rightarrow \text{Close\_Fire\_Door}(X) \quad (21)$$

From an updated state of the world, we generate a sensor trace by creating sensor values based on the state of the world, and adding to the trace whenever the values differ from the previous turn. For example, a smoke sensor will be activated 60% of the time if there was smoke on the previous turn. If there is a fire at a location, there is an 80% chance of smoke for obvious causal reasons. Also, smoke will travel to neighboring locations, especially if there is airflow in that direction. This is also modeled probabilistically. Thus, given the state of the world as the fire progresses, it was straightforward to generate sensor traces.

To generate the inference trace, it is necessary to model how the inference engine works, and how it discovers new rule instantiations. We posit that rule instantiations will only be discovered by an inference engine after sensor values in the TMS reflect that a rule is valid and necessary to discover a new analysis, or take a new action. In other words, if the TMS could use the rule, after a certain time of inference engine work, it will be inserted into the TMS. To this end, we fully preseeding a TMS with all propositional instantiations of all of our rules. The simulator updated the sensors on a turn-by-turn basis in the TMS. When a propositional rule was first used in the TMS to make a node IN, a callback was issued to the simulator. The simulator modeled the time for inferencing work to discover this newly used rule, and after the appropriate delay, added the rule to the inference trace. In this way, the sensor trace and the inference trace are tightly coupled with respect to realism.

In our model of the world, the TMS orders actions to be taken to fight and prevent the fire. Some of these actions are irreversible (like closing fire doors) and others are used as appropriate (like turning on and off sprinklers). The simulator accepts callbacks from the TMS on these action nodes when building these traces. On a callback, if a node would have already been created by the inference engine, we take appropriate action by updating the state of the world, after a simulated delay as a time cost for taking real-world actions. This further increases the simulated synchronization between the TMS and the ship on fire.

4.3 Trace Generation

Using the simulator described above, we modeled a fire on a ship with 150 locations over 1,000 timesteps. We generated a sensor trace of 84,185 transactions with 220,831 sensor updates and an inference trace of 2,250 rule discoveries for 1,096 analysis and action nodes, with a total TMS size of 6796 nodes. It is appropriate that the inference trace is considerably smaller than the sensor trace, because once a rule is discovered, the TMS automatically maintains its values. The sensors, however, must be constantly updated when their values change.

To support the different methods of evaluation, it is necessary to take the sensor trace, and divide it differently, so tests by concurrency and locality can be performed. To test locality, we divided the trace into five traces. To vary the data locality of the updates, we relied on the underlying physical model of the fire scenario. Intuitively, sensor updates to one location will effect strongly effect actions and deductions for that location. It will also somewhat effect its neighbors, but the effect on locations far away will be nearly non-existent. Looking in more depth at our inference rules, most of them pertain to a single location, or a location, and its near neighbors. Only a rare chain of specific inference rules will allow a sensor update in one location to effect another location far away. Thus, in dividing up the sensor trace by locality, we mapped sensor updates from consecutive locational regions into a single sensor to provide high locality. For very low locality, we took the sensor updates for each location, and distributed them among all five thread updaters. We also made intermediate splits along this spectrum by by varying the number of consecutive locations in a region, and by varying the evenness of distribution of a single location among sensor updaters.

To create different sensor threads to measure throughput on varying amounts of concurrency, we chose a fixed locality policy, of using seven consecutive locations per sensor thread. Then, we divided the sensor trace using this policy among any number of sensor updaters that we chose. We tested divisions of between one and ten concurrent sensor updaters. In each of our runs, we also provided a single inference updater that handled inserting all the new inferences.

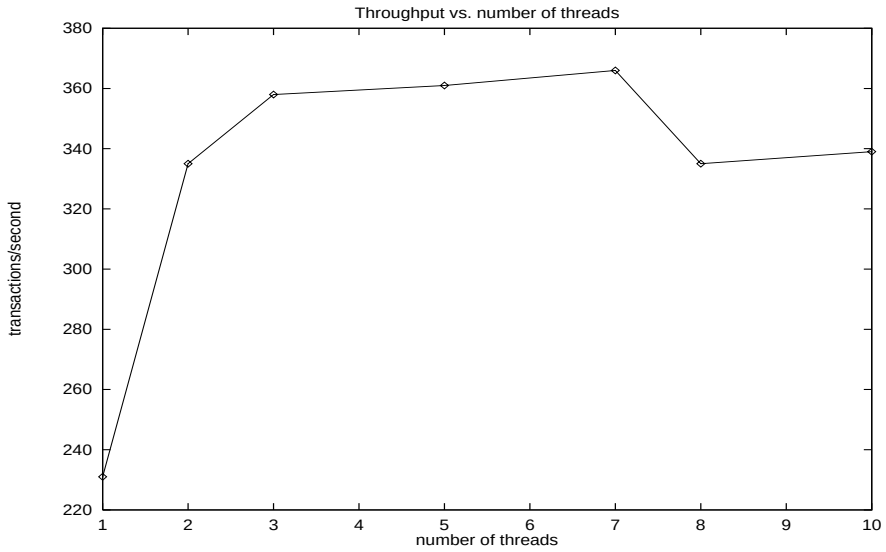

Figure 2: Throughput for various levels of concurrency. Notice that as we increase the number of sensor threads, throughput increases significantly, leveraging the slack in the system. Eventually, though, the overhead of the multiple threads, and the loss of data locality provides a slower decay in throughput.

5 Results and Analysis

All our experiments were run on an older 100 MHz DEC Alpha with 25 Meg RAM. All the sensor updaters ran on a different machine running over a local ethernet connection. External network traffic was minimal, and no other load on the machine running the TMS was present other than regular system maintenance processes.

Figure 2 shows the effect of concurrency on throughput. Notice that the effect of adding even a second sensor updater increases throughput by about 50%. This indicates that there was significant slack in the system to be leveraged. This slack lay both in the network and on the machine. About half of the image size of the TMS was resident in the machine's memory, and active paging was noticed. Maximal concurrency was reached around five or seven sensor update threads with a throughput of about 365 transactions per second. Each transaction varied in size from one to four sensor updates.

Figure 3 provides evidence that as concurrency grows, the loss of data locality becomes a more serious issue. The number of aborts due to timeouts on lock acquisition grows more than linearly with the number of concurrent threads. It is reassuring to see that the drop in throughput in Figure 2 corresponds nicely with the increase in aborts in Figure 3. This suggests that the maximum concurrency is declining due to these effects of data locality under high concurrency.

Figure 4 presents throughput figures for increasing amounts of locality when there are five sensor threads running concurrently. Surprisingly, there is no dominant trend suggesting that poor locality effects throughput. The presence of multiple updaters working in the same region doesn't have a strong impact on performance. This means that the unavoidable potential for deadlock among nodes that share consequences is not an impediment to the use of a concurrent TMS at this level of concurrency. It is likely, though that with larger amounts of concurrency, these locality issues would have a real effect. Note that the number of aborts in Figure 3 for ten concurrent updaters is an order of magnitude higher than for five updaters. This would seem to indicate that the real issues due to data locality are not felt at this level of concurrency, even though we are getting optimal throughput at five sensor updaters.

Figure 5 show the number of aborted transactions with our different locality policies. This shows that indeed, our policies do have an effect on locality, as the number of conflicts decreases with our increase in locality. However, given that there are 84,000 transactions in the run, the percentage of aborts changes from 0.29% on the high end to 0.14% on the low end. These numbers are small enough that it indicates again that conflicts due to locality are not a driving factor for performance at this level of concurrency.

In our measurement of the cost of transactions, we compared the throughput when running a single sensor updater with and without transactions. With transactions turned on, we got a throughput of 241 transactions per second. With transactions turned off, we got a throughput of 397 transactions per second. This indicates that we are paying a significant penalty for using transactions. In fact, the maximal throughput we got in the concurrent case was about equal to using a single case with no transactions. However, this result does not lead us to believe that transactions are

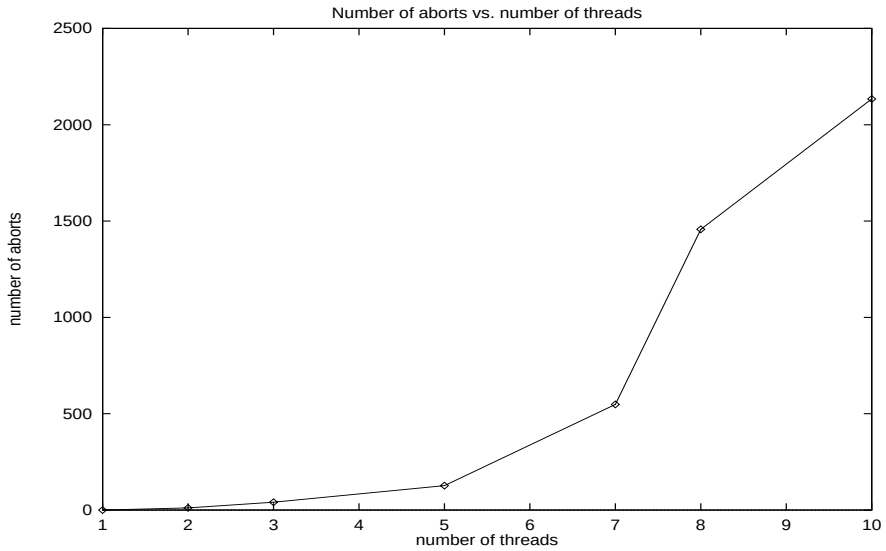

Figure 3: The number of aborted transactions as a function of concurrency. Note the natural increase in the number of aborted transactions as concurrency increases. This graph supports our hypothesis that the loss of throughput with large amounts of concurrency is due in part of a loss of data locality. The number of aborted transactions is a strong measure of data locality. The sudden increase in aborted transaction corresponds to the loss in throughput.

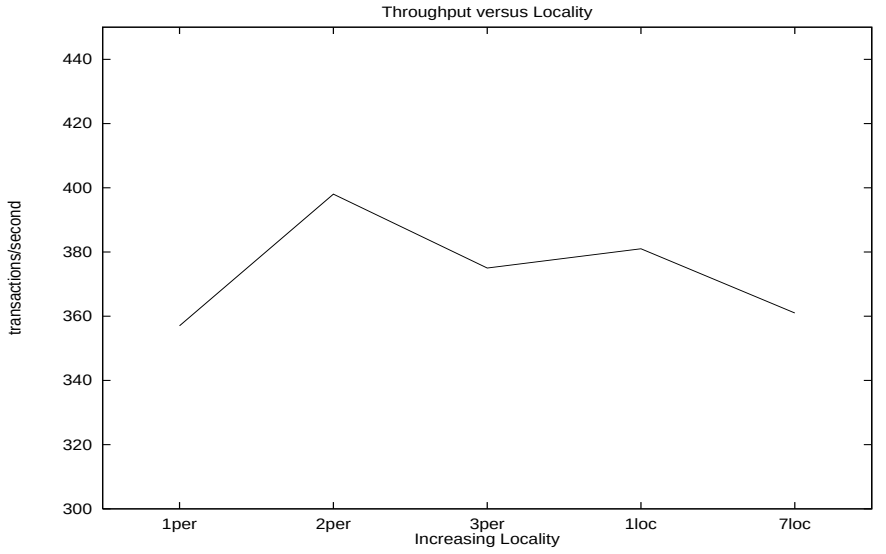

Figure 4: The throughput as a function of locality. The labels on the x axis refer to different ways of dividing up updates among threads. As we move to the right, the local region owned by a thread (meaning it is the only updater for facts in the region) increases. Note that there is no strong trend in the figure.

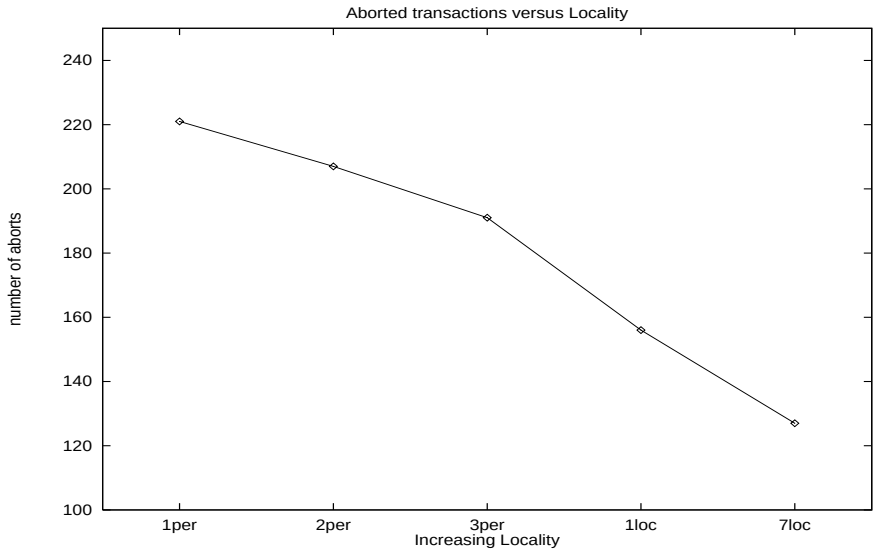

Figure 5: The number of aborts as a function of locality. As the region owned by a single updater increases, the number of aborts decreases.

of no use. First, many of our transaction operations made heavy use of list copying to maintain correct state. Many of these list copies could be avoided by adding extra logic to our list iterators to be knowledgeable about shadow objects. This would increase the performance of the transaction module. Secondly, the amount of stall time in our system was not particularly large. Once could easily imagine significantly larger TMSs that require heavy disk access to maintain persistent storage that would involve much more stall time. Such a system would benefit even more with concurrency, and the cost of transactions in such a system might be small in comparison to its benefits. In conclusion, we have shown that a transactional, concurrent TMS is feasible, and demonstrates expected properties of standard concurrent systems.

6 Bibliography

- Bayer, R., and Schkolnick, M. *Concurrency of Operations on B-Trees*. 1976.
- Birrell. *An Introduction to Programming with Threads*. 1989.
- Gray, J. N. Notes on Data Base Operating Systems, in *Operating Systems: An advanced Course*. Springer-Verlag, 1979.

A User-Level File Service Based on Watchdogs

Mihai Budiu, Raluca Budiu {mihaib+,ralucav+}@cs.cmu.edu

December 15, 1997

Abstract

This paper describes a user-level extension to the file mechanism offered by the Linux kernel. Unprivileged user processes (called henceforth "watchdogs") can control the access to files and manage file contents transparently for the processes which use these files. The flexibility of this mechanism is exploited in the implementation of a simple user-level remote-file service which has very good performance on some benchmarks, due to an enhanced cache manager.

1 Motivation

Our thesis is the following: "sometimes increased flexibility may be preferred to a rigid (but generally fast) scheme."[1]

To prove this assertion we move a service (remote file + caching) from the kernel to the user space. We gain much in flexibility, at the expense of some overhead. Our caching service is moved in user space because it does not fit well the internal interfaces of a Unix kernel, using vastly different paradigms (i.e. caching on variable-sized intervals). However, in order to make this service available to previously written applications (without having to modify them), we build a system call interception layer which permits cooperation of the kernel and the caching service.

2 Terminology and Organization

We will use the term "watchdog" both to refer to a process which "supervises" a file by filtering accesses to it, and to refer to the data structures of the kernel used for this purpose.

We will refer to the supervised files as "clients" of the watchdog. We also occasionally blur the distinction between the processes which act on client files, calling them "clients" as well (although this position is entirely transparent to them).

This document is structured like follows: section 3 shows how the new kernel mechanism can be used, section 4 describes how the kernel intercepts system calls and passes them to a user-level "watchdog", section 5 estimates the performance of the watchdog mechanism theoretically and empirically. Section 6 discusses the new filing service we have implemented and how it uses the watchdog services; the performance is measured in a variety of ways. Section 7 concludes.

[1] This is basically a variant of the end-to-end argument in [Saltz84]: only the application can decide what kind of semantics it needs, and no general-purpose mechanism can satisfy all the needs that may arise.

3 User Level Interaction

We begin by describing the kernel part of out project. We have a mechanism by which a user process (the watchdog) can register to observe the system calls done on a set of files. The kernel intercepts the system calls of other processes (clients) and sends a description of the intended action to the watchdog. The watchdog replies to the kernel indicating how the system call should be executed. There are three possibilities:

- The system call should fail and an error should be reported to the user;
- The system call should be executed using the normal execution path;
- The system call should be handled by the watchdog on behalf of the kernel ("faked"); the results will be passed from the watchdog to the kernel and from the kernel to the client process.

The best way to understand how a watchdog interacts with the kernel is to see a very simple example. Here is a how an (unprivileged!) user can write a simple watchdog, which just traces opens, fakes reads and denies writes on a file (we elliminated error processing from the code):

```
#include "watchdog.h"

int main()
{
   int fd, /* file to supervise */
       wd; /* watchdog interface */
   char buf[] = "Pass this data to the client on read";

   fd = open("file_name", O_RDWR); /* need R/W permission */
   wd = watchdog(fd);   /* sys call: register as watchdog. wd=file descriptor */

   while (1) {
      struct wd_request wq;   /* kernel will describe syscall here */
      struct wd_reply   wr;   /* watchdog will give answer here */

      read(wd, (char*)&wq, sizeof(wq));     /* This blocks the watchdog until
                                               the kernel has info to report */
      switch (wq.operation) {
        case WD_OPEN:
                printf("Process %d opens 'file_name', mode %d\n", wq.mode);
                wr.action = WD_ALLOW;  /* let the process perform
                                          the requested operation */
                break;
        case WD_READ:
                printf("Process %d reads from 'file_name', offset %ld,"
                        "size %ld\n", wq.pid, wq.offset, wq.size);
                wr.action = WD_FAKE;  /* the watchdog will supply the
                                         read data */
                if (wq.size < sizeof(buf)) wr.size = wq.size;
                else wr.size = sizeof(buf);  /* How much data supplied */
                wr.buffer = buf;      /* This is the supplied data. */
                break;
```

```
        case WD_WRITE:
                printf("Process %d wants to write in 'file_name', offset %ld,"
                        "size %ld\n", wq.pid, wq.offset, wq.size);
                wr.action = WD_DENY;  /* Block the requested action */
                break;
      }
      write(wd, (char *)&wr, sizeof(wr));    /* Send the answer to the kernel */
      if (wq.operation == WD_CLOSE) break;   /* If a client closed file, stop */
   }
   close(fd);  /* stop watching this file */
   close(wd);  /* close the watchdog channel */
}
```

4 Internal Architecture

Figure 1 shows the internal architecture and functioning of the kernel part of our system.

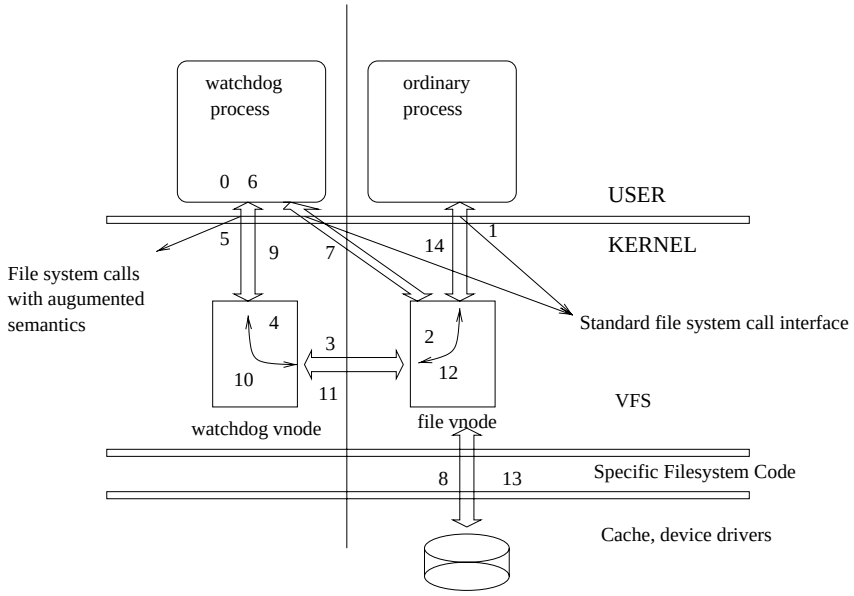

Figure 1: Architecture of the system

Here is a step-by-step description of a typical interaction to a supervised file, which shows how a run through the code in section 3 is handled internally.

We assume that the watchdog has opened the supervised file previously and that it has created the watchdog inode. The rest of the processing goes like follows:

0. The watchdog does a `read()` call on the watchdog inode to wait for information from the kernel. This suspends the watchdog process;
1. A process makes a system call on the supervised file;
2. The VFS level intercepts the system call;

3. The client file inode builds a data structure inside the watchdog inode describing the attempted system call on the file;
4. The process is suspended and the watchdog process is woken up;
5. This data structure describing the system call is passed to the watchdog as a result of the watchdog `read()` system call in step 0; the watchdog is woken up;
6. The watchdog does some processing based on the received information;
7. The watchdog may decide to directly access the file;
8. The watchdog's file access goes directly to the true file, without being intercepted by the VFS layer;
9. The watchdog does a `write()` system call on the watchdog inode, sending to the kernel its decision; it may decide to send a piece of data which is to be "served" to the process as the file contents;
10. The kernel decodes the decision;
11. The kernel resumes the suspended process;
12. The VFS layer carries the action indicated by the watchdog; data sent from the watchdog is received by this inode like being file data;
13. The suspended process may be let to access the file directly itself if the watchdog says so;
14. The system call of the client on the file terminates.

Everybody is ready for a new cycle of operations starting from step 0.

4.1 Architectural Details

Everything is handled entirely in the VFS layer [Klei86], so no file-system dependent code had to be changed. This makes the watchdogs applicable on any type of object which is represented in the filesystem (e.g. sockets, directories, pipes).

The watchdog file is basically represented in the kernel by a vnode[2] of a special type. A process can have only one watchdog file, which is used to multiplex all the information about the system calls on the "client" files.[3] Unlike [Bersh88], which defines a special message-passing interface for kernel-watchdog communication, we use a file abstraction mechanism for this purpose.

The bulk of the kernel code consisted in implementing the proper methods for this new type of file: `read()`, `write()` and `close()`. There is also some amount of code for system call interception; the intercepted system calls are right now `read()`, `write()`, `lseek()`, `open()` and `close()`.

Data movement between the watchdog process and its clients is accomplished with the same mechanism used for (unnamed) pipes (i.e. copy-in to a kernel buffer associated with the watchdog inode and copy-out to the user space).

[2] Linux uses the terminology "inode" for the objects traditionally called vnodes.

[3] As a possible extension, if watchdog processes will be multithreaded, the kernel could maintain one watchdog vnode per thread.

5 Performance Evaluation

In all the experiments that follow we measured the timing repeatedly using very lightly loaded machines both for the client and the server. We could not control the load on the AFS servers, though.

5.1 The Critical Path: An Estimation

Unfortunately all of the above steps except step 0 are on the critical path for any client system call. A standard system call (i.e. on non supervised files) would contain only steps 1, 2, 13, 14. All the other are pure overhead which has to be rather low to make this new feature attractive.

The bottom line of the performance estimation as it depends on our machine, is summarized in table 1. The data is for a PentiumII at 266Mhz, with 64M RAM, running Linux 2.0.30.

Operation	Cost/operation
2 extra system calls	2μs/call
2-tens extra context switches	7.5μs/switch
2 additional data copies	20-40ms/Mb[4]

Table 1: Overhead Estimation (compared to a standard file system call).

5.2 Performance Evaluation Experiments

To assess the true overheads we performed some micro-tests. We measured 5 ways to perform some operations on files on the *local* disk (i.e. the watchdog performs the client operation on another file on the local disk). For this test the data is actually in the buffer cache, because the test is repeated many times to measure precisely:

- Simple operation using the operating system (labeled "OS");
- Operation denied by the watchdog (denied);
- Operation allowed by the watchdog and performed by the OS (allowed);
- The operation is performed by the watchdog on behalf of the process (faked);
- The same operation using a named pipe instead of a file (pipe) and an ordinary process instead of a watchdog at the other end of the pipe.

The tests are for the following operations:

- Make a `lseek()` system call (which may fail, in the case of pipes);
- Make a `lseek()` followed by a `read()` system call for one byte;
- Make a `lseek()` followed by a `read()` system call for one page (4Kbytes);
- Make a `lseek()` followed by a `read()` system call for 10 pages (40Kbytes)
- Make a `lseek()` followed by a `read()` system call for 1 Megabyte.

[4]This depends on cache parameters, hit rate and other factors; the figure is true for large transfers (comparable to the L2 cache size, of 512K).

The table 2 gives the wall-clock time in microseconds. The precision of the measurement is within 1%.

Operation	seek	read(1)	read(4096)	read(40K)	read(1M)
OS	2.1	5.1	8.6	220	21406
denied	17	33	33	33	33
allowed	17	35	42	270	22968
faked	20	44	88	634	50300
pipe	2.2	11	66	720	57812

Table 2: Basic operation cost (μs) — cache hits.

It is worth noticing that the cost of this kind of IPC is on a par with pipes, and even better for large transfers (this is due to the fact that we allocate more buffer space in the kernel than a pipe so we imply fewer context switches, and because we can allow `read()` system calls to return large amounts of data, reducing the number of kernel crossings necessary to fetch it).

To evaluate the cost of an operation which does *not* hit in the cache is much more difficult, because of the low timer resolution which cannot be used to measure non-repetitive events. A coarse measurement estimated the cost of reading 1 Mb sequentially from the disk at 150ms.

5.3 Conclusions Based on Overhead Estimation

The bottom line for overheads of the watchdog scheme is in table 3.

Data	Disk accessed	Watchdog overhead
little	no	factor of 10
much	no	factor of 2
any	yes	within 20%

Table 3: Watchdog Overhead: a Summary.

This gives us hints about the types of operations where watchdogs may be effective[5]. These are:

- Operations with poor locality (demanding thus frequent disk access);
- Operations on large pieces of data which can benefit from application-specific optimizations[6].

[5] Assuming that the functionality of the watchdog could be implemented in the filesystem itself; watchdogs have however an unlimited range of behaviours, many of which could not be easily fitted inside a kernel.

[6] There is actually an extra cost due to the pageable nature of the watchdog buffers (being in user space). This can be somewhat controlled with a system call like `vadvise()`. This does not have measurable impact on any of our measurements in this paper because we have a very large core memory and we use a lightly loaded machine.

6 Application — Remote File Service with Smart Caching

We have implemented a "poor man's NFS[7]" on top of a watchdog. This very simple filesystem is configured to fetch the files from various servers using TCP/IP.

We use in the watchdog the interval-cache library developed by Mihai during the summer for the PDL. This gives us flexibility in accommodating highly variable access patterns and lets us tune the behaviour of the cache to enhance throughput for the cases illustrated previously.

6.1 Application Architecture

The figure 2 shows the architecture of the remote file service. For this implementation of the watchdog caching service we have used two pieces of code that we have developed previously: a simple user-level threads package and the smart interval-cache manager. We rewrote the bottom-half of the cache manager and we have written a watchdog to use the services of the top-half. We had to carry minor modifications to the cache itself, to accommodate for variable-sized files (the cache was designed to handle fixed-size partitions).

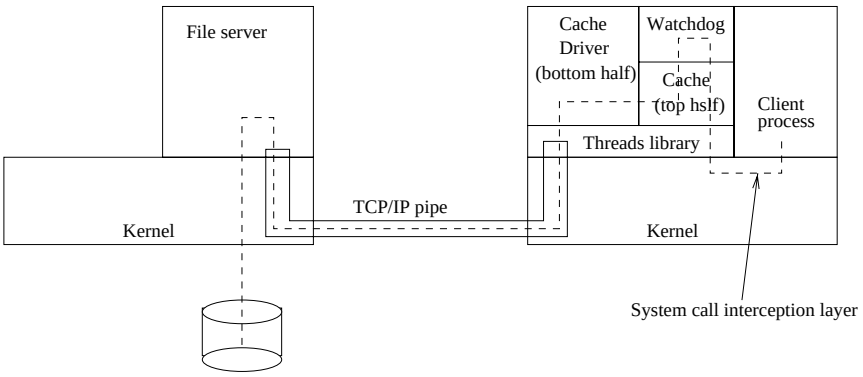

Figure 2: Architecture of the Remote File Service

Table 4 presents an overview of the parts we implemented/used.

Code	Description	Developed
Client	Accesses the files transparently	Unmodified
Watchdog	Receives system calls and decides action	From scratch
Interception	Kernel layer to redirect system calls	From scratch
Cache (top)	Caches variable sized intervals	Adapted
Cache (bottom)	Interfaces with the remote server	From scratch
Threads	Very simple user-level threads	Unmodified
File server	Handles remote requests	From scratch
Benchmarks	Various tests	From scratch

Table 4: Project code breakdown.

[7]This filesystem provides right now only little of the NFS functionality (e.g. no directory operations). There's no reason why this functionality could not be extended.

6.2 Performance for the Application

As mentioned in the performance section, the interval cache can give increased performance for some types of accesses; the ones that would benefit most are:

- Very small scattered writes which this cache can perform asynchronously, without having previously to write-allocate (i.e. read) the containing block;
- Very large sequential operations.

6.2.1 Sequential Tests

We compared our implementation with NFS on transferring a very large file between two computers in the CMU LAN. The server was running in both cases on a SUN SPARC4, gs20, and the watchdog and the client were running on the same Linux workstation on which we carried the previous measurements.

We have set the block size of our interval-cache to be 64Kb for this test. The interval-cache is using a rather small cache (compared to the OS which can use the whole 64M RAM) of 640Kb.

The test consists in copying a very large file (2M) between the two ends. The command running the test is `cp vmunix file`[8]. In all tests the file was *not cached* locally at the beginning of the unique transfer. The four filesystems compared are:

- UFS: the local Unix Filesystem;
- NFS: the Sun Network File System;
- AFS: the Andrew File System;
- WD: our watchdog-based file system.

The times are accurate within 20% (observed from repeated measurements); variations are due to poor timer resolution and unpredictable environment (network).

Protocol	Source	Destination	Time (s)
UFS	local disk	local disk	0.2
NFS	NFS partition	local disk	8.4
WD	remote partition	local disk	6.9
AFS	AFS partition	local disk	5.3
NFS	local disk	NFS partition	61
WD	local disk	remote partition	11.3
AFS	local disk	AFS partition	9.5

Table 5: Large sequential writes performance; WD cache capacity is 640KB, WD cache block is 64K. Time is given in seconds.

Note that NFS can benefit from the read-ahead, while our simple file server/watchdog has no such facility (yet). This is why the read performance of NFS is so much better than the write performance. The write performance of NFS is very bad.

[8] It is important to note that cp reads the file in 4K sized blocks. However, this does not coincide with the unit of transfer from the remote file server.

6.2.2 Sequential Performance as Function of Buffer Sizes

The performance of the copy test depends on two parameters:

- The amount of data transfered by the copy program in a system call (i.e. the `read()` buffer size);
- The size of the cache block.

Table 6 displays this dependency, which we measured by re-implementing the `cp` program to use larger buffers.

Syscall block	1K	4K	16K	64K
Cache block	To local disk			
64K	8	7	7	8
32K	13	13	13	9
16K	27	27	28	8
8K	54	54	27	8
	To remote disk[9]			
64K	10	10	10	4
32K	16	16	16	4
16K	30	23	4	4
8K	48	57	4	4

Table 6: Sequential WD copy time (in seconds) function of the cache block size and of the request size. The cache size is constant at 640K.

Observe that the performance is very good even if the cache uses small blocks (1K), but the client makes accesses in large chunks. This happens because the cache is an interval cache, and manipulates sets of cache blocks as units. The cache initiates simultaneously operations for many blocks which form a contiguous range, using scatter-gather (the `readv()/writev()` system calls), and these transfers are viewed by the transport protocol as a single large unit.

The write performance is even better because our cache handles writes smaller than the cache block size asynchronously, and thus the timing that we see comes mainly because the cache has been filled and has to be flushed. A very interesting phenomenon occurs in column 3 of the write test (16K syscall block), where a smaller block size for transfer gives actually better throughput! This is due to the way our cache handles incomplete blocks. If a syscall buffer size is a precise multiple of the cache block size this is handled more efficiently than if it is a divisor[10]. Another abnormal value is the 57; we conjecture that this is due to a transient overloading of the network (we should re-do this measurement).

[9] The `close()` system call terminates in our implementation after data is flushed from the local cache, but does not wait for confirmation from the remote end; this is why write times are sometimes shorter. This is one advantage of the reliable transport protocol which we have chosen.

[10] A more detailed knowledge of the internals of our cache is necessary to understand this behaviour. Basically our cache can accept writes to a quarter of a cache block without doing any read, because it stores the data and a descriptor. This descriptor takes space in the cache, so four quarters of a block do not actually fit in a block; writing the fourth one triggers a write-allocate: the block is read (unnecessarily!) and the quarters are overwritten. The cache could be improved in this respect.

6.2.3 Random Access Tests

We used two artificial tests; we believe that such access patterns are plausible for some scientific applications, of course, at some larger scale. One could certainly argue that there is something contrived in these tests, as the right way to solve this problem is to use the same caching technique in a user-level library. Remember though that we are not touching the application!

- Matrix multiplication: a file is initialized to contain two randomly generated integer matrices of 50*50 elements, and next their product is computed in a third matrix at the end of the file. Each 4-byte element is accessed directly with a `read()` of `write()` system call.
- Selection sort: a file is initialized to contain 200 random integer values (on 4 bytes each); a sorting program orders the file "in situ".

In all the tests the data is in a remote file. We used a 4Kb block for our cache for this test. The data fits entirely in the local cache in all cases, so the cost we see is mainly given by the cache management.

Table 7 gives the performance measurement:

Test	Protocol	Duration (s)
Matrix	UFS	1.6
	NFS	301
	WD	22.5
	AFS	4.69
Sort	UFS	0.2
	NFS	13.6
	WD	1.7
	AFS	0.4

Table 7: Random access performance.

The performance of NFS on these tests is abysmal. It is more than 12 times slower than our caching method.

The multiplication test is doing $O(n^2)$ writes and $O(n^3)$ reads. To see which of these operations is the bottleneck we carried measurements for several matrix sizes. The NFS time grows more like n^2 for small values of n, so we conjecture that small writes (even with good locality) have very poor performance on NFS. A further investigation would dump the traffic between the client and the server to observe the cache behaviour. We haven't done that yet.

6.2.4 A Hard Test for AFS

AFS beats our method for the previous tests because it practically performs all data accesses locally, so it has the advantage of lower overhead. We devised another test on which our scheme outperforms AFS on cold-cache start, and almost equals AFS performance on warm-cache start[11]. The test consists of a series of 20 bytes writes strided at variable distance of each other in an

[11] We could control the AFS cache behaviour by logging onto another client and creating/deleting the file from there, relying on the coherent cache protocol of AFS.

existing 2Mb file. Our cache used 64K blocks, the same as the AFS caching unit. Here is the minimum[12] time as measured by repeating the experiment 4 times (the measured time includes the `close()` system call, which synchronizes caches in both AFS and WD):

Protocol	Cache	Stride	Intervals written	Time (s)
AFS	cold	1K	64*25	9.8
AFS	warm	1K	64*25	5.5
AFS	cold	64K	25	10.0
AFS	warm	64K	25	5.6
WD	cold	1K	64*25	6.3
WD	warm	1K	64*25	6.4
WD	cold	64K	25	6.4
WD	warm	64K	25	6.4

The results show indeed that the number of writes in AFS is not as important as the number of 64K intervals spanned; crossing such a boundary incurs a very high cost in AFS. This proves that the AFS caching scheme is prone to inefficiencies for some access patterns, which can be successfully avoided by the watchdog.

7 Conclusions

This project proves that the cost of the flexibility obtained by implementing services in user space can sometimes be recovered if the services fit what the client applications need. The kernel offers a single service, customized to give good performance for a class of applications (e.g. the cache is effective when locality is good). However, using our scheme we can implement *simultaneously different* cache schemes for different applications, tuning the behaviour for independent optimizations.

7.1 Course Relevance

This project draws from the following topics treated in the course:

- Concurrency and threads: programming in the kernel is writing monitor code; the watchdog itself is multithreaded;
- File system design and implementation; caching services;
- Delegating services to the user space (idea discussed at the memory management in the context of Mach);
- Networking.

7.2 Future Work

The project suggests some interesting development paths. Some simply intend to enhance functionality, while other may exploit other applications of the watchdog scheme. Here are some ideas:

[12] We had big variability in the results at the measurement, which was probably due to a loaded network. We think that the important one is the *minimum* time to complete, as this one cannot be artificially diminished by any experimental circumstances.

Functionality enhancement: our project has some limitations which can be easily overcome with more implementation time. Some of these are:

- The watchdog has to be able to answer to the requests in any order;
- Right now the watchdog `read()` system call is non-interruptible;
- Client system calls on supervised files are also non-interruptible;
- The watchdog has to have all supervised files open; this puts a limit on the number of files it can simultaneously service;
- More system calls should be intercepted; first candidates: directory operations, `mmap()`, `getpage()`, `putpage()`, `fcntl()`, etc.

Other applications:

- It would be interesting to evaluate the remote file service for some large scientific applications with large data sets which strain AFS to make it non-local;
- Some other applications of watchdogs seem worth implementing. Many ideas are given in [Bersh88]. To these we can add: distributed shared memory, cgi-bin for web servers, etc.

References

[Bersh88] B. B. Bershad, C. B. Pinkerton. Watchdogs — Extending the UNIX File System. *Computing Systems, 1, 1988*, pp. 169.

[Saltz84] J. Saltzer, D. Reed, D Clark — The End-to-End Argument in System Design. *ACM Transactions on Computer Systems, vol 2, nr. 4, 1984*, pp 277–288.

[Faul91] R. Faulkner, R. Gomes. The Process File System and Process Model in UNIX System V. *Proceedings of the 1991 Winter USENIX Conference*, Jan. 1991, pp.243–252.

[Klei86] S. R. Kleiman, 1986. Vnodes: An Architecture for Multiple File System Types in Sun Unix. *USENIX Summer Conference Proceedings 1986*, pp. 238–247.

[Vaha96] Uresh Vahalia. Chapters 9, 10, 11 in *UNIX Internals.*, Prentice Hall, 1996.