CITI Technical Report 95–3

Peephole Log Optimization

L.B. Huston
lhuston@citi.umich.edu

P. Honeyman
honey@citi.umich.edu

Center for Information Technology Integration
University of Michigan
Ann Arbor

ABSTRACT

The log files generated while operating a file system in disconnected mode grow to substantial sizes. Eliminating redundant or useless operations in these logs can free up scarce disk space on laptops, reduce replay times, and reduce the frequency of data conflict. Our approach uses a rule-based portable peephole optimizer for compilers. This work suggests a general method of optimization for any system that performs logging at the vnode layer.

January 26, 1995

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Peephole Log Optimization

L.B. Huston
lhuston@citi.umich.edu

P. Honeyman
honey@citi.umich.edu

Center for Information Technology Integration
University of Michigan
Ann Arbor

1. Introduction

One goal of mobile computing is to provide users with a system that faithfully emulates the desktop environment. The mobile environment makes this goal a difficult one to reach because of limitations such as sporadic, low-bandwidth networks. Moreover, mobile computers tend to be resource poor, so that consumption of resources, such as disk space, demands careful control.

The conventional paradigm for modern computing systems is distributed computing. An essential service in a distributed computing environment is the (distributed) file system. One successful technique of providing mobile clients with a distributed file system is disconnected operation [5, 6], a form of optimistic replication in which the client can continue working with cached data when file servers become unavailable. As part of disconnected operation, any updates need to be logged so they can be propagated to the server at a later time.

If each operation is logged as it occurs, many log entries will be redundant. For example, if a user is editing a file called `paper.ms` portions of the log will typically contain the following entries.

```
   .
   .
   .
setattr(paper.ms)
write(paper.ms)
   .
   .
   .
setattr(paper.ms)
write(paper.ms)
   .
   .
   .
```

In this sequence, each `setattr` and `write` pair corresponds to each time the user saves the file she is editing.[1]

The object of replaying the log is to achieve the file system state that would have been realized were the client connected. In the absence of conflict, the naive, but incontrovertibly correct approach to achieving this state is to replay each deferred operation in the log. This approach is wasteful if it performs operations that do not affect the final state. For example, in the log segment shown above, it is possible to eliminate the first `setattr/write` pair, as these operations are made superfluous by the the second `setattr/write` pair.

Many other operations can be removed from a log without affecting the final state of the file system. For example, if file `foo` is created and subsequently deleted during a period of disconnection, the `create` and `remove` operations, along with any other operations on `foo`, can be deleted from the log.

Removing unnecessary operations has several advantages. The amount of time needed to replay the log depends directly on the number of operations being replayed; this is especially important when replaying over a dialup network, as we often do. Furthermore, by reducing the log size, disk space — a scarce

[1] The `setattr` operation is characteristic of UNIX text editors, which truncate before writing new file data.

resource on many laptops — is freed. Additionally, Kistler reports that removing unneeded operations reduces the chance of a conflict during replay [7].

To reduce the size of the log, transformations must be applied so that the optimized log is equivalent to the original log, *i.e.*, replaying either log results in identical final states of the file system. In the remainder of this paper, we describe related work, followed by a detailed description of our use of a peephole optimizer to reduce the size of logs produced during disconnected operation of AFS. We conclude with measurements of the effectiveness and performance of the optimizer.

2. Related Work

Early work on log transformation was in the context of optimistic replication for databases [1]. Log transformations use semantic properties to convert a given log into an equivalent, shorter log, which can reduce the cost of synchronizing replication sites after a network partition.

The Coda file system also uses log transformations to reduce the space needed for the log, and to reduce the time necessary for reintegration [7].

A problem related to optimization of file system logs is compiler peephole optimization [2, 9, 10], used to remove redundant instructions from assembly code when a program written in a high level language is compiled.

3. Design

The system we have built is designed to be used with a version of AFS [4] that supports disconnected operation [5]. Logging in disconnected AFS is performed at the vnode layer; each log entry corresponds to a vnode operation [8]. The operations that concern us are the mutating ones — those that cause data in the file system to be modified. For a standard vnode interface these operations are:

```
store, setattr, create, remove, rename, link, symlink, mkdir, rmdir
```

The `store` operation is not a standard vnode operation, but represents the operation that stores modified file data to the file server. In AFS this is `close`, but for other distributed file systems, *e.g.* NFS [11], this would correspond to the `write` and `putpage` operations.

To optimize our disconnected AFS logs we have elected to use a machine-independent peephole optimizer designed by Fraser [3], and modify it to support our needs. Fraser's peephole optimizer takes as input a list of rules, each of which consists of a set of *source* operations, followed by a set of *target* operations equivalent to the source set. The optimizer applies these rules by searching for a set of operations in the input that matches a source set, and replacing them with the corresponding target set. The optimizer iterates until no more rules can be applied.

To optimize the log we have built three components: a preprocessor, an optimizer, and a postprocesser. Although the preprocessor and postprocessor are specific to our AFS work, and embed detailed knowledge of the log format, the optimizer itself is independent of file system particulars.

3.1. The Preprocessor

Because the peephole optimizer operates on character strings, the log of disconnected operations — stored as binary data — must be converted into a text equivalent. Some information in the log is unnecessary for transformation, but is required for replay. The preprocessor elides this information, but provides a means for the postprocessor to recover it.

To increase the efficacy of the optimizer, we have broken the `setattr` vnode operation into three operations. The first component, `set_len`, sets the length of the file. The second component, `set_mode`, sets mode, owner, and group information on a file or directory. The third operation, `set_time`, sets the modification or access time. A given `setattr` might combine more than one of these operations; during optimization we treat each independently, and recombine them in the postprocessing stage.

We also elaborate the semantics of the `rename` and `create` operations. For a `rename` that overwrites an existing file, we precede the operation by a `remove` on the target name. Similarly, we replace a `create` operation on an existing file by a truncate operation, *i.e.,* `set_len` to zero. By expanding these operations during the preprocessing stage, we are able to simplify the optimizer because it need not detect these special cases. The postprocessing stage converts these operations back into a single operation.

The preprocessor reads the log file and emits an output string for each entry in the log. Each output string is one of the following.

```
Opid   create     PDid    Fid     name
Opid   mkdir      PDid    Did     name
Opid   remove     PDid    Fid     name
Opid   rmdir      PDid    Did     name
Opid   link       PDid    Fid     name
Opid   symlink    PDid    Fid     name
Opid   rename     SPDid   DPDid   Fid    oname   nname
Opid   store      Fid     dist
Opid   set_len    Fid     len     dist
Opid   set_mode   Fid     dist
Opid   set_time   Fid     dist
```

The `Opid` field is an index of the operation in the disconnected log. The postprocessor uses this field to reconstruct the log from the optimized output. The `Opid` is also used to sort the optimized operations so that when the operations are replayed, any file timestamps will have the same relative ordering as they would have had in the unoptimized case.

The `PDid` uniquely identifies the parent directory of the operation. In the case of `rename`, both the source parent directory and the destination parent directory are included.

The `Fid` and the `Did` fields serve as unique identifiers for a file or directory in the cache. Some operations include a `name` field to identify one of several names linked to a `Fid`; the `name` field unambiguously identifies a name in the file system by encoding a numeric representation of the file name and its parent directory (`PDid`). The `dist`, or *distinguished*, field indicates whether a file has exactly one name. Unless the preprocessor has direct knowledge to the contrary, all files are assumed to have multiple names.

For clarity, the remainder of the paper elides fields not directly relevant to a given example.

3.2. The Optimizer

The operation of Fraser's optimizer is simple. The user specifies a set of substitution rules. A rule is a sequence of lines containing a search pattern and a sequence of replacement lines. Any search pattern found is replaced.

We made some small modifications to the peephole optimizer. The largest of these adds support for conditional events that can trigger substitution. These conditionals are needed for the ordering operations described in Section 3.2.2.

We provide a set of rules that describe the correct optimizations. The rules can be classified into two categories: *replacement* rules, which replace a set of adjacent operations with an equivalent set; and *ordering* rules, which re-order adjacent operations so that further replacement rules can be applied.

We perform the optimization in two phases: one that operates on data operations, and one that operates on namespace operations. Each phase uses a different set of ordering and replacement rules.

3.2.1. Replacement Rules

The replacement rules remove redundant operations from the log. A typical example of a replacement rule is:

$$\mathtt{Opid}_1\ \mathtt{create}\ \mathtt{PDid}_1\ \mathtt{Fid}_1\ \mathtt{name}_1$$
$$\mathtt{Opid}_2\ \mathtt{remove}\ \mathtt{PDid}_2\ \mathtt{Fid}_2\ \mathtt{name}_2$$

are removed if

$$\mathtt{Fid}_1 = \mathtt{Fid}_2 \text{ and } \mathtt{name}_1 = \mathtt{name}_2$$ [2]

[2] The `name` field embeds both the file name and parent directory ID in the encoding, so comparing $\mathtt{PDid}_1$ and $\mathtt{PDid}_2$ would be redundant.

When this pattern of consecutive `create/remove` operations is found in the log, both operations are elided: they do not affect the final state of the file system.

Another example of a replacement rule is:

> $\texttt{Opid}_1$ `create` $\texttt{Fid}_1$ $\texttt{name}_1$
> $\texttt{Opid}_2$ `rename` $\texttt{Fid}_2$ $\texttt{oname}_2$ $\texttt{nname}_2$

is replaced by

> $\texttt{Opid}_1\texttt{:Opid}_2$ `create` $\texttt{Fid}_1$ $\texttt{nname}_2$

if

> $\texttt{Fid}_1 = \texttt{Fid}_2$ and $\texttt{name}_1 = \texttt{oname}_2$

This rule represents a file that is created and subsequently renamed. The transformation allows the file to be created with the final destination name, eliminating the `rename` operation. Note that the new `Opid` is the concatenation of $\texttt{Opid}_1$ and $\texttt{Opid}_2$ separated by a ‘‘:’’. This allows the post processor to identify the operations that were combined.

3.2.2. Ordering Rules

It is often the case that potentially canceling operations are *not* adjacent in the log. For example, when a user creates a file, she usually performs some other operations before deleting the file, so the `create` and `remove` operations are separated by intervening operations, which causes the pattern matching to fail. To address this problem we provide rules that group related operations together.

Two operations can be reordered if all replacement rules applied to the reordered operations preserve the log’s semantics. Thus the legal ordering operations depend on the defined replacement operations. This issue is discussed further in sections 3.2.3 and 3.2.4.

An example of an ordering rule is:

> $\texttt{Opid}_1$ `set_mode` $\texttt{Fid}_1$ $\texttt{dist}_1$
> $\texttt{Opid}_2$ `set_mode` $\texttt{Fid}_2$ $\texttt{dist}_2$

are replaced by

> $\texttt{Opid}_2$ `set_mode` $\texttt{Fid}_2$ $\texttt{dist}_2$
> $\texttt{Opid}_1$ `set_mode` $\texttt{Fid}_1$ $\texttt{dist}_1$

if

> $\texttt{Fid}_2 < \texttt{Fid}_1$

These two operations can be reordered: they affect metadata in distinct files, so there is no dependence between the operations. The ordering rules group operations on similar files in a manner comparable to a bubble sort.

3.2.3. Data Optimization

The vnode operations can be divided into two different types: *data* operations, which modify file data or metadata; and *naming* operations, which modify the file system namespace.[3] Accordingly, we perform the optimization in two phases, the first for data operations, and the second for namespace operations. This allows us to take advantage of the differences between these types of operations.

The primary difference between the two phases lies in the ordering rules. In the first phase, the ordering operations group all data operations based on the `Fid` of the affected files. Because each data operation affects a single file, all data operations on the same file can be grouped together without violating the semantics of the file system. The ordering rules also separate out namespace operations, so that they do not interfere with data optimizations.

During this phase we don’t consider any replacement rules that modify the namespace. This restriction is

[3] Some operations such as `remove` can be both namespace and data operations. In each phase we treat these operations in the appropriate manner.

necessary because the ordering rules may allow a transformation in which such replacement rules could generate an invalid log.

The replacement rules, which remove redundant data operations, are given in Table 1.

Old Operations	New Operations	Conditional
Id_1 set_len Fid_1 $dist_1$ len_1 Id_2 set_len Fid_2 $dist_2$ len_2	Id_2 set_len Fid_2 $dist_2$ len_2	$Fid_1 = Fid_2$ && $len_2 = 0$ [4]
Id_1 set_mode Fid_1 $dist_1$ Id_2 set_mode Fid_2 $dist_2$	Id_1:Id_2 set_mode Fid_2 $dist_2$	$Fid_1 = Fid_2$
Id_1 set_time Fid_1 $dist_1$ Id_2 set_time Fid_2 $dist_2$	Id_1:Id_2 set_time Fid_2 $dist_2$	$Fid_1 = Fid_2$
Id_1 store Fid_1 $dist_1$ Id_2 store Fid_2 $dist_2$	Id_2 store Fid_2 $dist_2$	$Fid_1 = Fid_2$
Id_1 store Fid_1 $dist_1$ Id_2 set_len Fid_2 $dist_2$ len_2	Id_2 set_len Fid_2 $dist_1$ len_2	$Fid_1 = Fid_2$ && $len_2 = 0$
Id_1 store Fid_1 $dist_1$ Id_2 remove Fid_2	Id_2 remove Fid_2 $dist_1$	$Fid_1 = Fid_2$ && $dist_1 = 0$
Id_1 set_len Fid_1 $dist_1$ len_1 Id_2 remove Fid_2	Id_2 remove Fid_2 $dist_1$	$Fid_1 = Fid_2$ && $dist_1 = 0$
Id_1 set_mode Fid_1 $dist_1$ Id_2 remove Fid_2	Id_2 remove Fid_2 $dist_1$	$Fid_1 = Fid_2$ && $dist_1 = 0$
Id_1 set_time Fid_1 $dist_1$ Id_2 remove Fid_2	Id_2 remove Fid_2	$Fid_1 = Fid_2$ && $dist_1 = 0$

Table 1: Replacement rules for data optimization. This table lists replacement rules applied during optimization of data operations.

3.2.4. Naming Optimization

The second phase optimizes the namespace operations. Prior to this phase, the output from first phase must be sorted on `Opid` to place the remaining operations in their original order. This ensures the log is a valid replay log.

As with the data optimization phase, we use reordering operations to group related operations. Extra care must be taken to preserve semantic validity, *i.e.*, we may not perform any reordering that would lead to an invalid log. For example, consider the operations:

$Opid_1$ mkdir $PDid_1$ Did_1 $name_1$
$Opid_2$ create $PDid_2$ Fid_2 $name_2$

The ordering rules should not allow these operations to be reordered if $PDid_2$ is identical to Did_1 — if the `mkdir` operation is canceled by a `rmdir` operation, then the `create` operation applies to a non-existent directory.

[4] In general we could perform replacement when $len_2 <= len_1$, but in all cases we have seen, files are truncated to length 0.

An example of a valid ordering rule is:

$Opid_1$ mkdir $PDid_1$ Did_1 $name_1$
$Opid_2$ create $PDid_2$ Fid_2 $name_2$

are replaced by

$Opid_2$ create $PDid_2$ Fid_2 $name_2$
$Opid_1$ mkdir $Pfid_1$ Did_1 $name_1$

if

$Fid_2 < Did_1$ and $Did_1 \neq PDid_2$

It is possible to combine the data and naming optimizations into a single step, but the stricter ordering requirements of the naming optimizations may prevent the optimizer from finding some valid data transformations. The replacement rules for the namespace optimization are given in Table 2.

Old Operations	New Operations	Conditional
Id_1 create Fid_1 $name_1$ Id_2 remove Fid_2 $name_2$		$Fid_1 = Fid_2$ && $name_1 = name_2$
Id_1 link Fid_1 $name_1$ Id_2 remove Fid_2 $name_2$		$Fid_1 = Fid_2$ && $name_1 = name_2$
Id_1 symlink Fid_1 $name_1$ Id_2 remove Fid_2 $name_2$		$Fid_1 = Fid_2$ && $name_1 = name_2$
Id_1 mkdir Did_1 $name_1$ Id_2 rmdir Did_2 $name_2$		$Did_1 = Did_2$ && $name_1 = name_2$
Id_1 create Fid_1 $name_1$ Id_2 rename Fid_2 $oname_2$ $nname_2$	Id_1:Id_2 create Fid_1 $nname_2$	$Fid_1 = Fid_2$ && $name_1 = oname_2$
Id_1 link Fid_1 $name_1$ Id_2 rename Fid_2 $oname_2$ $nname_2$	Id_1:Id_2 link Fid_1 $nname_2$	$Fid_1 = Fid_2$ && $name_1 = oname_2$
Id_1 symlink Fid_1 $name_1$ Id_2 rename Fid_2 $oname_2$ $nname_2$	Id_1:Id_2 symlink Fid_1 $nname_2$	$Fid_1 = Fid_2$ && $name_1 = oname_2$
Id_1 mkdir Did_1 $name_1$ Id_2 rename Fid_2 $oname_2$ $nname_2$	Id_1:Id_2 mkdir Did_1 $nname_2$	$Did_1 = Fid_2$ && $name_1 = oname_2$
Id_1 rename Fid_1 $oname_1$ $nname_1$ Id_2 rename Fid_2 $oname_2$ $nname_2$	Id_1 rename Fid_1 $oname_1$ $nname_2$	$Fid_1 = Fid_2$ && $nname_1 = oname_2$

Table 2: Replacement rules for namespace optimization. This table lists the replacement rules applied during the optimization of the naming operations.

3.3. The Postprocessor

The final component of our system is the postprocessor, which restores the output from the optimizer to the original log format. This procedure includes reordering the log to correspond to its original order, and collapsing expanded operations into their original form. The former step ensures that all file modifications show the same relative time ordering, so that tools such as `make` and `rcs` work properly.

4. Performance

Applied to logs gathered from our own use of disconnected AFS, the peephole optimizer removed almost three-quarters of the logged operations. Measurements were made using 13 log files collected during our travels, each containing between 173 and 4727 mutating operations. (The overwhelming majority of the vnode operations logged are non-mutating ones — over 95%.) The long logs represent disconnected operation intervals of a day or more. The average log contained 1334 operations before optimization, 360 operations afterward.

Table 3 summarizes the results of our measurements. While replay time tends to be dominated by RPC latency, `store` operations also require significant network transit time, especially over slow links.

Operation	Total Operations	% Removed
`create`	1612	44
`mkdir`	11	0
`remove`	1699	36
`rmdir`	3	0
`link`	9	100
`symlink`	121	100
`rename`	1172	33
`store`	6671	84
`setattr`	6046	85
`Total`	17344	27

Table 3: Fraction of operations removed. This table lists each mutating operation along with the fraction of operations removed during a typical optimization. The numbers following each operation give the percentage of these operations that are removed during optimization. The `store` and `setattr` operations are removed during the data optimization phase, all other operations are removed during namespace optimization.

Measurements made on a 33 Mhz Intel 486 running Mach 2.5 show that it takes an average of 33 seconds to optimize a log, with 9% of the time spent in preprocessing, 18% of the time in data optimization, 64% in namespace optimization, and 9% of the time in the postprocessor. The optimizer is CPU bound: on a 20 Mhz Intel 386SX running identical software, the same optimizations take almost five times as long, with approximately the same time distribution for the different phases.

5. Conclusions

We have described a way to implement log optimization for a disconnected file system that performs logging at the vnode layer. Our mechanism builds on the well understood principles of compiler peephole optimization, and uses an off-the-shelf optimizer with minimal changes. The aspects of our approach that are implementation specific are the pre- and postprocessor, which are easy to construct when the log structure is known.

While it would be possible to perform log transformations ‘‘on the fly,’’ *i.e.*, when appending to the log, this would add overhead to each operation. With our approach, the optimizer runs while the file system is quiescent. It could be run immediately before invoking the replay code, to reduce the amount of time needed for replay, or at any time that disk space becomes scarce, to free up more space.

6. References

1. B. Blaustein, H. Garcia-Molina, D. Ries, R. Chilenskas, and C. Kaufman, ‘‘Maintaining Replicated Databases Even in the Presence of Network Partitions,’’ *Proc. of the IEEE EASCON Conf.* (September 1983).

2. Jack W. Davidson and Christopher W. Fraser, ‘‘The Design and Application of a Retargetable Peephole Optimizer,’’ *ACM TOPLAS* **2**(2), pp. 191–202 (April 1980).

3. Christopher W. Fraser, ‘‘A Compact, Machine-Independent Peephole Optimizer,’’ pp. 1–6 in *Proc. of the 6th Ann. ACM Symp. on POPL*, San Antonio (January 1979).

4. John H. Howard, ‘‘An Overview of the Andrew File System,’’ pp. 23-26 in *USENIX Conf. Proc.*, Dallas (Winter 1988).

5. L. B. Huston and P. Honeyman, ‘‘Disconnected Operation for AFS,’’ pp. 1–10 in *Proc. of the 1993*

USENIX Symp. on Mobile and Location-Independent Computing (August 1993).

6. J.J. Kistler and M. Satyanarayanan, ‘‘Disconnected Operation in the Coda File System,’’ *ACM TOCS* **10**(1), pp. 213–225 (February 1992).
7. James J. Kistler, ‘‘Disconnected Operation in a Distributed File System,’’ Ph.D. Thesis, Carnegie Mellon University (May 1993).
8. S.R. Kleiman, ‘‘Vnodes: An Architecture for Multiple File System Types in Sun UNIX,’’ pp. 238–247 in *USENIX Conf. Proc.*, Atlanta (Summer 1986).
9. David Alex Lamb, ‘‘Construction of a Peephole Optimizer,’’ *Software — Practice and Experience* **11**(6) (June 1981).
10. W. M. McKeeman, ‘‘Peephole Optimization,’’ *CACM* **8**(7), pp. 443–444 (July 1965).
11. D. Walsh, B. Lyon, G. Sager, J.M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss, ‘‘Overview of the Sun Network Filesystem,’’ in *USENIX Conf. Proc.*, Dallas (Winter 1985).