An Implementation of an Applicative File System* Technical Report 354**

Brian C. Heck and David S. Wise

Computer Science Department
Indiana University
Bloomington, IN 47405–4101 USA
Fax: +1 (812) 855-4829
Email: heckb@cs.indiana.edu

Abstract. A purely functional file system has been built on top of pure Scheme. It provides persistent structures and massive storage expected of file systems, without explicit side-effects like read and write. The file system becomes an additional, lazy argument to programs that would read from it, and an additional result from functions that would alter it.
Functional programming on lazy structures replaces in-place side-effects with a significant storage management problem, handled by conjoining the heap to the file system. A hardware implementation of reference counting is extended out to manage sectors, as well as the primary heap. Backing it is a garbage collector of heap and of disk (i.e. UNIX's **fsck**), needed only at reboot.

CR categories and Subject Descriptors:
D.4.2 [**Storage Management**]: Storage hierarchies; D.1.1 [**Applicative (Functional) Programming**]; E.2 [**Data Storage Representations**]: Linked representations; H.0 [**Information Systems**].
General Term: Design.

Additional Key Words and Phrases: Reference counting heap, mark/sweep garbage collection, hardware, Scheme, functional programming.

1 Motivation

The acceptance of functional, or applicative, programming languages has been encouraging. However, as often as they are taught and used, their scope of application has been restricted to formalism, to toy algorithms, and to simple systems. True, "purity" has been constrained at higher levels, to yield success stories like Lisp's (with side effects), ML's [11] (without laziness), and maybe Haskell's [5] (under UNIX I/O). With such constraints, however, they are unlikely to fulfill their acknowledged promise for parallel processing.

** To appear in *Proc. Intl. Workshop on Memory Management*, Springer Lecture Notes in Computer Science (1992).
* Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number CCR 90-02797.

In a formal context applicative languages are often used as the foundation for semantics, for program analysis, and for rigorous documentation or proof. Although one can argue that important, non-"toy" algorithms (like divide-and-conquer tree searching or Strassen's matrix multiplication) were first invented and are best taught using functional style; nevertheless, even these are used in production from C or Fortran source code.

Similarly, it has long been understood that Landin's streams [9], or lazy evaluation, allows a simple system to be expressed as an applicative program [4, 6] whose input is the stream `stdin` and whose output is the stream `stdout`. However, two things essential to a full operating system are still missing from such models: some kind of indeterminism, *e.g.* to interleave multiple, asynchronous inputs; and a persistent file system to cushion users against failures. A reliable file system is essential to allow recovery from a crash without rehearsing all of history, beginning when the system was first installed. Designing and implementing the file system is the goal of this project.

We expect two things from an Applicative File System (AFS). The first, already mentioned, is the ability to establish critical data structures in persistent media—commonly on a magnetic disk. Then all of ephemeral, primary memory might be lost, yet the system can rapidly be restarted from data on disk, recovering the system to a persistent configuration that recently preceded the catastrophe.

The second seems incidental: that files are larger than structures in main memory. Sometimes they are stored in secondary memory because of sheer bulk; often they are static over long periods. These properties derive from physical properties of the storage medium, and our own habits in using it.

However, AFS must behave within the constraints of pure functional programming: that the only operation is applying a function to arguments. Side effects are not available; thus, the programmer can neither "read" nor "write" a file. He can, however, traverse one stream-like parameter, and generate another one as a result. Either may be "bound" to a name in a distinguished environment that is commonly called a file directory.

The remainder of this paper is in six parts. The next section briefly contrasts common serial structures with, in particular, linked trees, setting up the importance of reference-counting hardware in Section 3. Similarly, Section 4 deals with streamed input and output, setting up the file system described in Sections 5 and 6, formally in the former, operationally in the latter. Finally, Section 7 presents a simple running example, and Section 8 contrasts this with past work and offers conclusions.

2 Side-effected Aggregates vs. Recopied Trees.

The only memory model we use is Lisp's (actually Scheme's) heap. Every data structure is built from binary nodes, a member of the recursive domain:

$$S = E + S \times S,$$

where E is a flat domain of elementary items. Conventional vectors, and their conventional in-place updates are not used. If static, they are easily mimicked with

linked lists or (better yet) trees; so they are unnecessary. Moreover, it is possible to recopy a perfect tree of n nodes, incorporating one change, by creating only $\lg n$ new nodes. Therefore, side-effect-free updates are simulated cheaply and we can prohibit side-effects, consistent with functional programming and lazy evaluation.

Since the only non-trivial structure is a list, the only file structure is a persistent list. Because file updates, similarly, cannot be done by means of side-effects, it is possible to sustain two, perhaps several, successive incarnations of a single file simultaneously—merely as different lists, likely with shared substructure.

From the perspective of the database manager we have simplified the file system; we need only to keep the most current of several surviving incarnations, even while older ones are still bound and traversed elsewhere in the system. Later we shall describe how lists in memory migrate off to disk as files, and vice versa.

The inital perspective is that the file consistency problem has simply been traded for for a massive storage management problem. The manager or garbage collector needs to handle both binary nodes in primary (ephemeral) memory and sector-objects in secondary (persistent) memory. The remainder of this paper describes how that system was built and how it runs.

3 Reference Counting vs. Garbage Collection

We have built a system that has a hybrid storage manager; it has reference counting machinery both in main memory as well as on disk, and it is backed up with garbage collectors in both places.

A foundation to the system is the hardware implementation of Reference Counting Memory (RCM) [17, 18]. RCM is reported elsewhere, and the interesting story here is how AFS was laid over it. However, a brief overview of the hardware is necessary first.

3.1 Reference Counting in Hardware

RCM has been implemented as a device on a NeXT computer. Although its design would support full memory speed, the first prototype appears as eight megabytes of microsecond memory. It is configured as a half-meganode heap and an equal amount of serial memory that "roots" RCM. A second version is being designed for parallel processing; the description below presumes that there are several RCMs.

Every write of a pointer in RCM is a read-modify-write. That is, a new pointer is overwritten at a memory location only with removal of a former reference in that location during the same memory cycle. The algorithm to write a pointer is dispatched from a processor to memory where the following C code is executed uninterruptibly (as a remote procedure local to `destination`):

```
struct node
{
   integral RefCt;
   node *left, *right;
};
```

```
void store(pointer, destination)
  node *pointer, **destination;
{
    dispatch incrementCount(pointer);
    dispatch decrementCount(*destination);
    *destination = pointer;
}
```

All these operations occur essentially in parallel, subject to two constraints: the increment is dispatched before the decrement, and the former content at the destination is used just before it is overwritten. Both the fetch from *destination* and the store there occur during the same memory cycle (read-modify-write). The sequentiality of these three steps in uniprocessor C satisfies the constraints of a uniprocessor, although we intend them to be nearly simultaneous in hardware. Again, the increment and decrement are dispatched on-line, but they complete off-line.

Reference-counting transactions can be interleaved with similar ones dispatched from other memories to the same *destination*, as long as increments/decrements arrive at the targeted reference count as some merging of the orders in which they were dispatched. A unique, non-caching path between any source-destination pair, as on a bus or a banyan net meets this constraint.

At the destination address, both increments

```
void incrementCount(p)
  node *p;
{
    if Sticky(p->RefCt) ;else p->RefCt++ ;
}
```

and decrements occur as atomic transactions.

```
void decrementCount(p)
  node *p;
{
    if (Sticky(p->RefCt) || --(p->RefCt)) ;else FREE(p) ;
}
```

The use of FREE above indicates return to the local available space list. *Each of these three operations requires only finite time;* a node can return to available space still containing live, yet-counted pointers [15]. Thus, one memory location can, on one hand, handle a store and, on the other, act on a couple increments or decrements during one memory cycle.

RCM is controlled by reading or writing to special memory registers. Notably, new nodes of two types are allocated by reading from distinguished addresses. Because of this and because increments, particularly, must not be deferred, RCM is written-through the cache and read without caching.

In addition, RCM has on-board support for the rotations required in Deutsch-Schorr-Waite marking and has an on-line sweeper so that memory cycles for garbage collection can beat stop-and-copy. Early benchmarks show it running MachScheme (hobbled to use a recursive stack) faster than equivalent code using a RAM heap [18].

MachScheme is MacScheme [13] ported to the Mach operating system on the NeXT computer, and subsequently revised to use RCM for its heap of binary nodes. We acknowledge Lightship for granting source-code access to MacScheme.

3.2 AFS over MachScheme over RCM

AFS is then implemented over MachScheme which uses MacScheme's tagged pointers, extended for RCM hardware. RCM provides two types of nodes in its randomly allocated heap space. Nodes having addresses of the form $8n$ are terminal nodes (floating-point numbers); nonterminal binary nodes have addresses of the form $8n+4$.

MachScheme's tagging system and RCM design allows us 18-bit addresses into our file system. With 1024-byte sectors this yields a quarter-gigabyte file system. However, the effect of internal fragmentation reduces this. Moreover, the present tests were run on a prototype file-system of 4000 sectors, so to demonstrate a space-constrained configuration.

AFS's directory structure is modeled after UNIX [12, 14]. All files have an associated data node (dnode) similar to a UNIX inode except that there are no indirect pointers. Dnodes contain a single pointer to the unique first sector of the file.

Initially, we wanted to have files collected by reference counting on disk, like those in UNIX. When we examined our design for sharing data we discovered that we really needed to maintain reference counts on each sector. So we began to use typed pointers for references from heap space to sectors, but this required reference counting to interfere, dispatching additional disk transactions on every write or overwrite of a sector reference; they would be prohibitively expensive. What we needed was a second RCM dedicated to keeping reference counts for sectors with count information dispatched from the RCM MachScheme was using. We, therefore, stole enough nodes from the existing RCM to dedicate one per sector for the prototype AFS.

As a result, all the reference counts both for nodes in memory, and for sectors on disk are maintained by the same circuits in RCM. The difference is that a node whose count drops to zero implicitly returns to available space. However, sector reference counts are "nailed down" so that they cannot return to zero (and be handled like an RCM binary node, instead of as a whole sector); thus, their counts must periodically be scanned to find one-counts—to be returned to the sector pool. (This scan will be eliminated under the next version of RCM.)

3.3 Garbage Collection on Disk

It is desirable that any garbage collection be deferred as much as possible because such a lengthy traversal is slow, particularly under multiprocessing and on disk. Garbage collection on disk parallels UNIX's **fsck**; it is slow but it can be necessary—especially for recovery after a catastrophe.

Moreover, MachScheme has its own garbage collector, using RCM's mark/sweep hardware. In order that these two collectors not interfere with each other, the RCM-resident counts on disk sectors are stored in two parts corresponding to the RCM-sourced and to the on-disk references. Thus, MachScheme's internal collector recomputes the former, but does not traverse the disk. **fsck** roots from the current disk directory, and traverses only the persistent memory; it requires a "quiet system," just like UNIX.

A sector on disk is composed of two parts: most of it is a compressed representation of the Scheme expression. A preorder-sequential representation is used, with tags indicating types immediately following. Circularities are detected and represented appropriately. Ten-bit pointers refer to positions in structure within that sector (uncounted), or to expanded 24-bit (counted) off-sector references. They form the second part of the sector—compressed at the end.

Thus, **fsck** need not traverse the first part of any sector, but must traverse and count the references at the end. And when a sector is condemned because its reference count drops to zero, all those forwarding references must be dereferenced.

3.4 Directories and Write-only-memory

The "file system" we discuss here is not built free-standing. In fact, it is nothing more than a permanent UNIX file of four kilosectors, which models a small, private random-access disk. UNIX utilities see it as a jumble of bits, and its UNIX-directory entry is irrelevant to the description below.

Within it is at least one—-possibly several successive—directories that we have generated. The most recent one of those represents the "current" file system. As discussed in Section 2, "creating," "deleting," or "changing" the binding of a file's name is effected by recopying the directory (itself a file) to include that alteration. Thenceforth, the `file_system` is bound to the newer directory.

After a file binding is created, the bound structure soon migrates onto disk. Because any file binding must now refer to a structure entirely disk-resident, huge files no longer need to consume heap in main memory. Manipulation of these files remains the same as if they were resident in that heap—except for the delays associated with file access, needed to copy sectors from it back into the heap.

The remaining problem is how to assure recovery of the file system after a system crash. The entire file system is still rooted in some ephemeral register of the computer, even though its entire content is resident on disk. However, the persistent information there is useless unless its root can be found.

One word of permanent memory (or disk) is set aside as *write-only-memory*, to receive a copy of the root of the file system after every update. (This contrasts with Section 2 protocols.) Operationally, this seems to be a side effect, but this binding is *completely invisible to a running system,* which uses its own ephemeral register as its root of the file system. In effect, the memory-resident root of the file system is copied into permanent memory, but nothing in this lifetime can use that copy; therefore, it may as well "not exist."

Whenever a conventional operating system is rebooted, it uses this distinguished address into permanent storage in order to root and to restore the file system as it

was when last stable. The system is restored to a configuration from the not-too-distant-past, and comparatively little is lost, just as in UNIX.

Of course, the former streams, `stdin` and `stdout`, are lost during a catastrophe, and the reboot establishes a new `stdout'` and provides a new `stdin'`, presumably initiated by a user aware that the crash occurred and likely inquisitive of what files survive in the aftermath.

4 `Stdin` and `Stdout`

`Stdin` and `stdout` are classically treated as special streams/files with read-once and write-only privileges respectively. As discussed in Section 1, many researchers have suggested using streams for I/O. Merging these two approaches under AFS provides an elegant solution. (These ideas are not implemented in the current AFS due to the eager nature of Scheme.)

As in any file in the system, `stdin` should become manifest in main memory when a read operation is attempted upon it. The data structure representing `stdin` should be a lazy list. The tail of the list would be a suspension which, when thawed, creates a list with the head being a character read from the real input device (or a `port-not-ready` token) and the tail being a copy of the original tail suspension. This new pair is placed in the tail position of the manifest portion of the lazy list and control returns. To get a character from the `stdin` file one simply takes the head of the list. The user is responsible for keeping track of where she is in the `stdin` data structure. Each computational thread has an incarnation of the file system, so each may have a different opinion of what the next character to be read is. There is no restriction that keeps a thread from rebinding its own `stdin` to a different device or file.

No side-effects are necessary to maintain `stdin` because only the main thread can write to the master file system. Until the main thread updates its version of `stdin`, the entire stream of characters up to the last actually read from the device can remain memory resident (or swapped to disk). Updating the main file system to reflect the current consumption of `stdin` by the main thread would be achieved by the user installing what she perceives to be the tail of the current `stdin` as `stdin'` in a new file system. An ideal time to perform this update would be just before spawning a new thread. Due to reference counting, the list of characters would be automatically collected as soon as all references to the older `stdin` no longer exist.

`Stdout` may be handled by the classic `write-only-file` viewpoint with the file system piping the `stdout` file to a logical device (perhaps unique for each file system which is active). The operating system may map each of the logical devices to an actual output device performing any merge operations necessary.

5 Functionality

Notation from formal semantics is used to specify the types of AFS primitives.

Finite Sets

$\pi \in$P	Persistent memory addresses
$\mu \in$M	Main memory addresses
$\iota \in$ I	Identifiers as file names

Domains

$S = E + (S \times S)$	S-expressions
$\alpha \in A = P + M$	Addresses
$\delta \in D = I \rightarrow P^{+}_{\perp}$	Directories
$\rho \in R = A \rightarrow \mathcal{N}$	Reference Count
$\chi \in C = P \rightarrow M_{\perp}$	Cache

An interface to the file system has been provided through the following commands. All commands take an implicit file system and return a new system implicitly.

mkf : $D \rightarrow I \rightarrow S \rightarrow D$
Makes S persistent and associates I with that persistent structure in the new file system.

rmf : $D \rightarrow I \rightarrow D$
Returns a new file system with no directory entry for I

getf : $D \rightarrow I \rightarrow S \times D$
Returns the data structure associated with I in D plus a new D.

mkdir : $D \rightarrow I \rightarrow D$
Returns a new file system with an entry for the directory I.

lnh : $D \rightarrow I \rightarrow I \rightarrow D$
Returns a new file system with an association between the existing file (the second identifier) with the first identifier via a hard link.

lns : $D \rightarrow I \rightarrow I \rightarrow D$
Returns a new file system with an association between the existing file (the second identifier) with the first identifier via a soft link.

fsck : $D \rightarrow D$
Returns a new file system after a disk garbage collection. Intended to be run only by the main thread.

createfs : $I \rightarrow \perp$
Used to build a new, empty file system in the UNIX file I. The user must install the system to use it.

load-fs : $I \rightarrow D$
Installs the file system contained in the UNIX file I into the current Scheme session.

close-fs : $D \rightarrow \perp$
Stores the current file system into the UNIX file it originated from and removes all file systems from the Scheme session.

6 System Operations

AFS users have the ability to create hard and soft links in much the same manner as in UNIX. (*Cf.* UNIX commands `ln` and `ln -s` respectively.) As in UNIX, hard links are counted references and soft links are uncounted. One interesting change, due to the elimination of side-effects, is a modification in the behavior of hard links. (Soft links behave exactly the same as their UNIX counterparts.) Following a hard link in UNIX returns the most current version of the file; this behavior stems from UNIX overwriting the file to install the new contents. However, AFS does not install the new data structure into the existing file system. Figure 1 contains before and after diagrams resulting from writing new "contents" to an existing hard link.

Initially, in the directory δ_1, hard links ι_1 and ι_2 correlate to the disk resident structure beginning with sector π_1. Issuing the command "(`mkf` ι_1 (`cons 6 15`))" creates a new file system in which δ_2, ι_2 still points to its old contents, but ι_1 does not. In δ_1 (the root of the old file system), both links still point to the old data structure, π_1.

6.1 File Creation

A request for migration of a data structure α to disk initiates a preorder-sequential compression [7] of α into a set of sectors P^+. At some time (undefined but "soon") after a request to make a data structure a file, the data structure will become persistent. Currently, AFS has eager behavior inherited from Scheme, but implicit laziness could be inserted. Lazy file write operations are consistent with most current operating systems' views of files and buffers; until all buffers are flushed and the file committed, no guarantees can be made about the state of the file. Laziness should be transparent to the user except after a catastrophic system failure after which the file system is guaranteed to come back up in some stable, pre-existing state; we just cannot say exactly what state. (The user will learn to program confirmations to `stdout` that depend on successful commits to migration.)

Graphs migrated to disk may produce arbitrary graphs of sectors. In the trivial case of a flat list of nonterminal nodes, all having unary reference counts, the file becomes a flat list of sectors. One consideration for migration of graphs to disk is that multiply referenced nodes provide opportunities for data sharing and introduce a possibility of circularity. We assert that circularity in manifest graphs can be detected and sector reference counts corrected to allow reference counting to collect the disk structures [3].

As for suspensions, we would install them into the file system *verbatim*, that is, unexpanded. The suspensions would remain persistent on disk, but heap resident versions could be thawed and allowed to continue their work. Because Scheme is eager, however, we don't provide for suspensions yet.

Figure 2 shows the data structure rooted at μ_1 before and after it becomes persistent. The content of μ_1 is copied into a new heap node μ_2. This is immediately followed by adding an entry to the resident sector cache associating the address of the file's first sector, π_1 (a preallocated sector) with μ_2. (The resident sector cache which maps resident sector addresses to RCM addresses ($\chi : P \rightarrow M$) reduces the likelihood of having multiple copies of sectors heap resident, but more importantly,

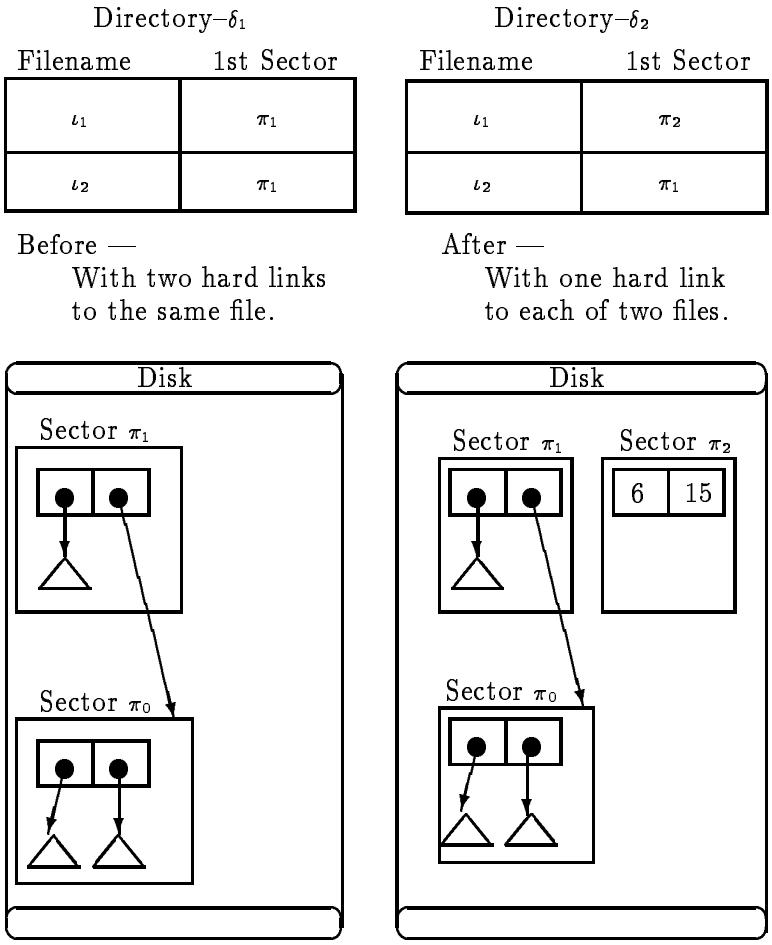

Fig. 1. Result of requesting (`mkf` ι_1 (`cons` 6 15)).

it prevents costly disk accesses if a needed sector is found to have a cache entry.) Next, the contents of μ_1's `car` and `cdr` fields are overwritten by forwarding pointers to the file's first sector, π_1.

The modifications to the original data structure to insert forwarding pointers are side-effects only at the level of implementation, not with respect to language semantics. Further, we assert there is no net consumption of heap space by the migration process. The recovery of the nodes containing forwarding pointers is completed during the next garbage-collection cycle. Thus the space needed for the new node μ_2 is offset by recovery (in the future) of the pre-existing node μ_1.

During compression, any multiply referenced, nonterminal nodes (μ_k) encountered are treated as separate trees and are rooted at the head of a new sector π_k (and have their contents replaced by forwarding pointers as μ_1's was in the above

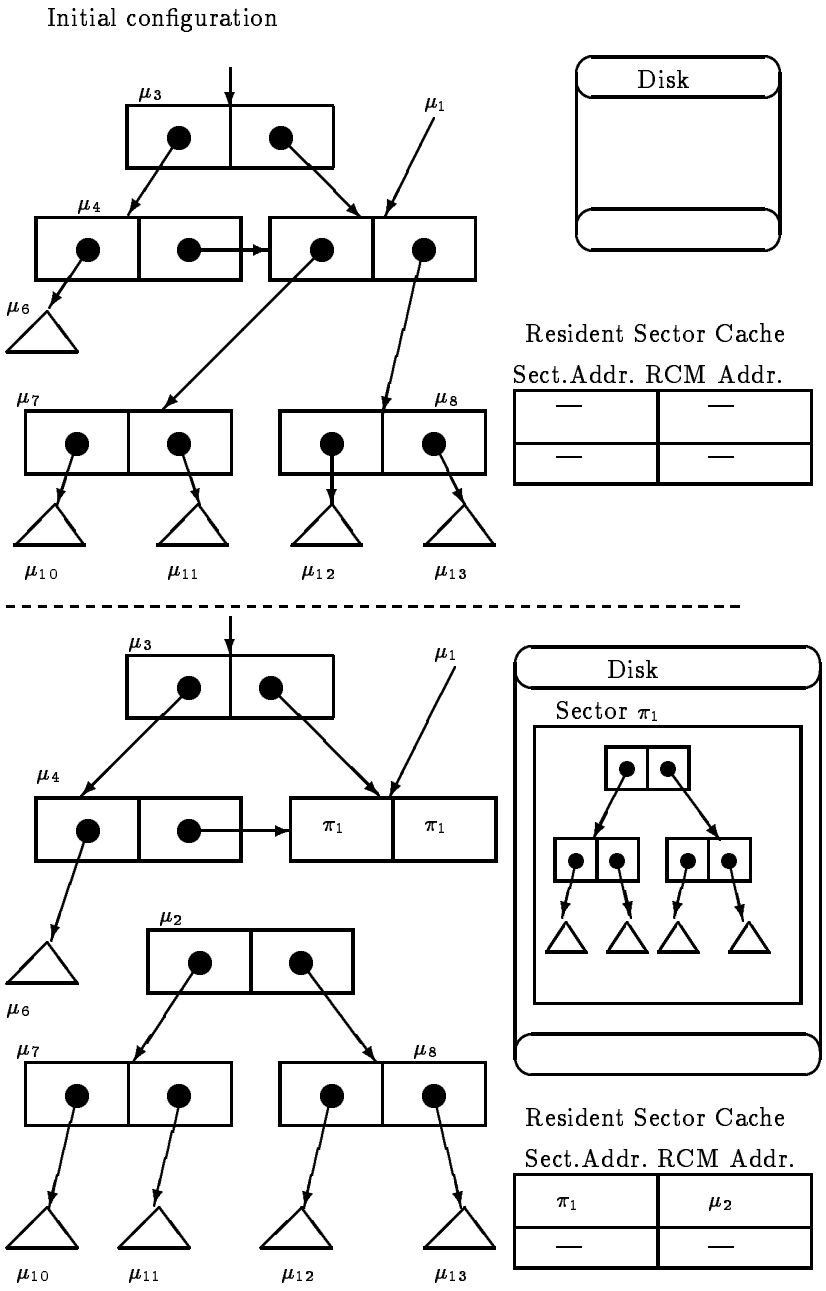

Fig. 2. Example of Data Migration: making μ_1 persistent.

example). A pointer to π_k is written into the current, compression buffer and the μ_k is marked as disk resident (via a forwarding pointer as discussed below). Next, μ_k is stacked for later compression onto disk (rooted at π_k). The placement of multiply referenced, nonterminal nodes at sector heads allows the sharing of that single data structure by all its referents.

While traversing the data structure, data from all uniquely referenced nodes and multiply referenced terminal nodes is copied into the current compression buffer. (If the buffer is full, a pointer to an empty sector is added and the buffer is written to the sector preallocated for it. Processing continues with the new, empty buffer.) Multiply referenced, terminal nodes are not assigned to unique sectors (to reduce internal fragmentation).

Sector pointers encountered in the heap during compression are added to the current sector as normal data along with one additional entry. At the end of every sector, a set of off-sector references, as mentioned in Section 3.3, is maintained. This list enables the disk's garbage collector to recalculate sector reference counts without scanning the sector.

6.2 File and Sector Migration from Disk to Heap

A getf command is treated as a request to load the root sector of the file into the heap. (The same mechanism is used to render sector addresses encountered in the heap into heap addresses.) The sector address is clashed against the resident sector cache. In the case of a hit ($\chi\pi \neq \perp$), the address of the heap resident copy of the data structure is returned. A cache miss ($\chi\pi = \perp$) forces a load of the desired sector into the heap and the addition of an entry in the cache to reflect that action. A traversal of the entire graph stored in the file will cause the file to be loaded into the heap one sector at a time. This is not to say that the entire file will be heap resident at any given time; if the file is sufficiently large that may not be possible).

7 Examples

The first example is a file editor taken from Friedman's and Wise's [4] paper. The example editor consumes a list of commands (stored in the file "commands"), applies the commands to the contents of another file "rhyme", and returns a list. We then install the list as a new version of the file "rhyme" in a new file system. The old version of "rhyme" is never deleted; it is automatically reclaimed when no longer in use.
The editor has six basic commands:

type	–Prints characters up to the next newline.
repos	–Repositions the cursor at the beginning of the text.
dele	–Deletes the character to the right of the cursor.
ins	–Inserts the given string to the left of the cursor.
find	–Finds the first occurrence of the given string to the right of the cursor. Success is reported if the string is found. Otherwise the cursor is advanced to the end of the text

and failure is reported.

subst –Locates the target string just as the find command and replaces it with the replacement string. Reports success or failure in the same manner as find.

```
MacScheme* Top Level Version 1.9 (Development)
>>> (createfs "fsys")
#t
>>> (load-fs "fsys")
#t
>>> (mkf "rhyme"  (rcmlist #\t #\h #\e #\space  #\q #\u #\i #\c #\k
         #\space #\b #\r #\o #\w #\n #\newline #\f #\o #\x))
#t
>>> (mkf "commands"
     (rcmlist
      (rcmlist find (rcmlist #\q #\u #\i #\c #\k #\space))
      (rcmlist type)
      (rcmlist dele)
      (rcmlist dele)
      (rcmlist ins (rcmlist #\d))
      (rcmlist subst (rcmlist #\newline) nil)
      (rcmlist repos)
      (rcmlist subst (rcmlist #\q #\u) (rcmlist #\s))
      ))
#t
>>> (mkf "rhyme" (editor (getf "commands") (getf "rhyme")))

#\:"find"(#\q #\u #\i #\c #\k #\space)
"Found: "(#\q #\u #\i #\c #\k #\space)
#\:"type"#\b#\r#\o#\w#\n
#\:"dele"
#\:"dele"
#\:"ins"(#\d)
#\:"subst"(#\newline)()
"Found"(#\newline)
```

```
#\:"repos"
#\:"subst"(#\q #\u)(#\s)
"Found"(#\q #\u)
#\:#t
>>> (getf "rhyme")
(#\t #\h #\e #\space #\s #\i #\c #\k #\space #\d #\o #\w #\n #\f
#\o #\x)
>>>
```

7.1 Example of Catastrophic Failure

When the system fails, AFS attempts to return to a previous, stable state. It will attempt to bootstrap the last version installed in the write-only memory discussed in Section 3.4. In the example below a system is created with thirty files (each with a unique copy of the same S-expression for convenience). While executing we cause the system to fail (via an interrupting control–C). To see how the system fared under a failure we list the files to determine which file was last created. According to the directory, "seed24" is the last file created intact. File "seed23" was probably in the process of being written to disk, but since the new file system (with "seed23" installed) was not written to the write-only-memory before the failure, the file is lost. This behavior is consistent with most current file systems.

```
MacScheme* Top Level Version 1.9 (Development)
>>> (createfs "fsys")
#t
>>> (load-fs "fsys")
#t
>>> (mk-manyfiles "seed" (rcmlist 1 (rcmlist 4 5.4)) 30)
^C
Program received signal 2, Interrupt
(gdb) q
prototype: ffsrnrcmsch
MacScheme* Top Level Version 1.9 (Development)
>>> (load-fs "fsys")
#t
>>> (ls)
("f" "seed24" "Fri Mar 27 21:10:07 1992
" "f" "seed25" "Fri Mar 27 21:10:07 1992
```

```
" "f" "seed26" "Fri Mar 27 21:10:07 1992
" "f" "seed27" "Fri Mar 27 21:10:07 1992
" "f" "seed28" "Fri Mar 27 21:10:07 1992
" "f" "seed29" "Fri Mar 27 21:10:06 1992
" "d" ".." "Fri Mar 27 21:09:49 1992
" "d" "." "Fri Mar 27 21:09:49 1992
")
>>> (getf "seed24")
(1 (4 5.4))
>>>
```

8 Conclusions

Present treatment of the migration of data between layers of memory under functional programming falls into two categories. Sometimes the problem is set aside, and the language enjoys an absence of restriction on transactions that are "outside" the program, as in ML, Lisp 1.5, or Scheme. Such languages do not extend very well to parallel processing.

Other languages attempt to isolate the problem: Backus's ASM in FP [1], William's and Wimmer's *histories* in FL [16], and Lucassen's and Gifford's *effect streams* in FX [10] are all serious efforts to encapsulate the "impure" file activity in order to isolate it from the "pure" functional portion of the language. Haskell [5] follows this tack. Alternatively, the time or scope for creation of persistent structures has been restricted [2].

In contrast, this research neither partitions nor encapsulates data. This treatment, in fact, would be transparent to the user if she were not required to participate, contributing some important declarations about her data. (ObjectStore [8], a general database system, similarly depends on only a few type assertions.) This is *experimental* work; one test of success is just to build a hierarchical memory in a purely functional environment without any "barriers." Another is to make it work well.

We have succeeded in the first test. The design and construction effort was not straightforward, but the difficulties we encountered all had an elegant solution, appropriately within the scope of the tools we had chosen.

Scheme is hardly an ideal language for this experiment; its lack of lazy evaluation corrupts the transparent implementation of UNIX pipes for `stdin` and `stdout`. However, a remarkably convincing test for such a generalized file system is to bring it to life under a general-purpose programming environment. We have built a production environment.

References

1. John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM,* **21**,8 (August 1978), 613–641.
2. David J. McNally and Antony J. T. Davie. Two models for integrating persistence and lazy functional languages. *SIGPLAN Notices,* **26**,5 (May 1991), 43–52.
3. Daniel P. Friedman and David S. Wise. Garbage collecting a heap which includes a scatter table. *Information Processing Letters* **5**,6 (Dec 1976), 161–164.
4. Daniel P. Friedman and David S. Wise. Aspects of applicative programming for file systems. In *Proc. of ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices* **12**,3 (Mar 1977), 41–55.
5. Paul Hudak, Simon Peyton Jones, and Philip Wadler (eds.). Report on the Programming Language Haskell. *SIGPLAN Notices* **27**,5 (May 1992), R1–R164.
6. Peter Henderson, Geraint A. Jones, and Simon B. Jones. *The LispKit Manual.* Tech. Monograph PRG-32 (2 vols.), Programming Research Grp., Oxford Univ. (1983).
7. Donald E. Knuth. *The Art of Computer Programming* **1** (2nd edition), Reading, MA, Addison Wesley (1973).
8. Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system, *Comm. ACM* **34**,10 (Oct 1991), 50–63.
9. P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation: Part I. *Comm. ACM* **8**,2 (Feb 1965), 89–101.
10. John M. Lucassen and David K. Gifford. Polymorphic effect systems. *Conf. Rec. 15th ACM Symp. on Principles of Programming Languages* (Jan 1988), 47–57.
11. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* Cambridge, MA, MIT Press (1990).
12. D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Bell System Tech. J.* **57**,6 (Jul–Aug 1978), 1905–1930.
13. Lightship Software. MacScheme © *Version 1.9 development.* Beaverton, OR (1989).
14. K. Thompson. UNIX Implementation. *Bell System Tech. J.* **57**,6 (Jul–Aug 1978), 1931–1946.
15. J. Weizenbaum. Symmetric list processor. *Comm. ACM* **6**,9 (Dec 1963), 524–544.
16. John H. Williams and Edward Wimmers. Sacrificing simplicity for convenience: where do you draw the line? *Conf. Rec. 15th ACM Symp. on Principles of Programming Languages* (Jan 1988), 169–179.

17. David S. Wise. Design for a multiprocessing heap with on-board reference counting. In P. Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (Sept 1985), 289–304.
18. David S. Wise, Caleb Hess, Willie Hunt, and Eric Ost. Uniprocessor performance of a reference-counting hardware heap. Tech. Rept., Computer Science Department, Indiana Univ. (in preparation).

This article was processed using the LaTeX macro package with LLNCS style