The Swift/RAID Distributed Transaction Driver

Bruce R. Montague

UCSC-CRL-93-03
1 January 1993

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

This document describes a distributed transaction driver developed to support the reimplementation of Swift with added RAID (Redundant Arrays of Inexpensive Disks) functionality. Both a high-level overview and a low-level program description are provided. The Swift system was developed to investigate the use of disk striping to achieve high I/O performance. The transaction driver described here has been used to implement RAID-0, RAID-4, and RAID-5 Swift systems. Swift uses a network of workstations in a manner similar to a redundant disk array, i.e., an application on a client node requests I/O via library routines which evenly distribute I/O across multiple server nodes. Data blocks in RAID files are distributed over the servers. RAID-0 contains no redundancy/parity information, RAID-4 uses a dedicated parity node, and RAID-5 uses distributed parity. The original Swift system used a straight–forward RAID-0 scheme that did not readily scale to RAID-4 and RAID-5 implementations. The transaction driver described here was developed to cope with the distributed concurrent programming problems posed by these implementations. In principle, this transaction driver can be used for a wide variety of distributed concurrent programming problems.

Keywords: Swift, RAID, concurrent programming, transactions, distributed systems

Contents

1. Conceptual Design of the Swift/RAID System

1.1 Introduction

Striping files across disks has been used for some time to increase disk throughput and balance disk load. In such systems a write request 'scatters' write data across a number of devices while a read 'gathers' data from these devices. The number of devices that can participate in such an operation defines the strip size. Recently, much work has been done using Redundant Arrays of Inexpensive Disks (RAID) to provide reliable high performance disk storage [Katz, et al., 89]. High performance is achieved by disk striping, while high availability is provided by the various RAID techniques, such as RAID-1 (simple disk mirroring), RAID-4, and RAID-5. These latter two techniques keep at least one parity block for every data strip. This parity data can be used to reconstruct an unavailable data block in the strip. They differ in that RAID-4 uses a dedicated parity device, which can become a bottleneck, while RAID-5 scatters parity data across the devices in the strip, thus achieving a more uniform load.

A system called Swift was implemented in the Concurrent Systems Laboratory at UCSC during 1990-1991 [Emigh 92]. Swift was designed to investigate the use of disk striping to achieve the high I/O rates required by multimedia applications. The performance of this system was investigated and reported in [Cabrera and Long 91]. The Swift system used striping only to enhance performance. No parity schemes were supported, i.e., Swift is a RAID-0 scheme.

The original Swift system was modified in July through August of 1992 to use a very different internal implementation. This new implementation was designed to support the addition of RAID-4 and RAID-5 functionality to Swift. This new implementation is based on a *transaction driver*. The transaction driver was used to implement RAID-0, then RAID-4, and finally RAID-5 versions of Swift. The RAID-0 and RAID-4 implementations were primarily used as scaffolding to develop the RAID-5 implementation.

1.2 The Problem

Obtaining the fault tolerant advantages of RAID-4 and RAID-5 in the Swift network environment is of obvious benefit. Indeed, workers in distributed systems have long been motivated by the effort to design systems which tolerate node failure. However, it is not easy to maintain acceptable performance when implementing RAID-4 and RAID-5 in the Swift environment. [Hartman and Ousterhout 92] correctly note that no single location exists to accumulate parity, that update operations, especially in the case of small random writes, require multiple network accesses to correctly perform parity update, and that ensuring atomic update of data and parity blocks is difficult.

Some of these drawbacks can be turned to advantage. For instance, in the case of small writes the old and new contents of the data block need to be accessed, XOR'ed, XORed with the old parity block, and the resulting new parity block written. In our current implementation, small write data XOR's are calculated on the nodes containing the data, thus taking advantage of parallelism in the distributed environment.

The original Swift prototype used a straight-forward send-receive protocol essentially implemented with matching in-line code. Transmission and timeout errors could be handled by a simple retransmit. A RAID implementation is more complex. For instance, if a node failure occurs in the midst of a write operation, the write operation must be altered to read all the nodes involved in all strips in the I/O (including the parity data), and then write the new parity data. Likewise, node failure during a read operation requires all the nodes in the strip, including the parity node, to be read. This change in the I/O sequence must take place dynamically when the error occurs.

1.3 Transaction Driver Models

There are two ways to look at the transaction driver – as a distributed virtual CPU or as a set of distributed objects. The distributed virtual CPU was the primary implementation model. In this approach distributed programming becomes an exercise in the design and implementation of a virtual instruction set. The distributed virtual CPU model is described here in some detail as it is of great help when reading the source code.

1.3.1 The Virtual CPU Model

Some early computers, such as the ENIAC, were fully asynchronous machines. Programming these machines proved so complex that the clocked synchronous model, commonly called the von Neumann model, was developed. The Swift programming problem resembles that found on early asynchronous machines. Indeed, computation and communication between the components of a modern network may outperform that of early asynchronous machines. Thus, an approach worthy of examination is to provide the programmer facilities similar to those developed by the hardware engineers who evolved the synchronous instruction model.

In the synchronous instruction model a program consists of a series of instructions. Each instruction executes atomically, i.e., from start to finish without possibility of external interruption. While the instruction executes, the instruction coordinates all asynchronous hardware activity internal to the CPU that is required to complete the instruction. During this period the instruction has total control of the machine. A small set of I/O instructions remain that are irreconcilable related to asynchronous real-world hardware. Interrupts were developed to assist these instructions. I/O instructions activate asynchronous activity and then the I/O instruction immediately completes, i.e., the instruction completes without waiting for the I/O to finish. Later, a hardware interrupt will activate the machine at a known instruction location. This activation takes place at a very specific time when both the hardware I/O has completed and an atomic instruction execution has completed.

This model has been quite successful, although performance considerations are currently pushing this model back towards including additional asynchronism, with the advent of multiple instruction per cycle designs, deep pipelines and delayed branches, etc.

The transaction driver is a simple distributed virtual machine that performs in the preceding manner. Each node executes a transaction driver. A transaction driver processes a sequence of data structures, each containing opcode and operand fields. Such a sequence defines a *transaction program*, with each structure specifying one logical instruction capable of executing atomically with respect to network I/O. Logical instructions never block internally on network I/O. If an instruction activates network I/O, it immediately completes. Each transaction driver may concurrently process an arbitrary number of transaction programs, i.e., the virtual machine defined by the transaction driver is multiprocessed. In Swift/RAID, the transaction driver for a given node has one transaction program active for every other node involved in the transaction. The transaction driver maintains the equivalent of CPU context for all its transaction programs within the data structure it uses to describe other nodes.

A *transaction plan* consists of a set of transaction programs cooperating to effect a distributed system service. A request for system service, for instance a Swift read or write request, is treated as a request to 'compile' a suitable transaction plan satisfying the service request. The transaction plan is compiled on the single node issuing the request. Compilation is achieved by calling simple routines containing assembler-like functionality. Two transaction programs are compiled for every node involved in the requested operation. One program will remain on the local node while the other will be transmitted to the remote node. This pair of transaction programs coordinate activity on the remote and local nodes in such a way as to complete that part of the transaction involving the remote node. Note that the message sent to the remote node is simply the program required to satisfy the request.

Most transaction instructions are part of a distributed pair of instructions, with one instruction on the remote node corresponding to one instruction on the local node. Such a pair defines a *distributed instruction*. This instruction can be thought of as a single instruction consisting of multiple control fields routed to different execution units or it can be considered a conventional instruction coordinating with a co-processor instruction. If an error occurs anywhere in the distributed instruction, the entire distributed instruction is restarted.

Each transaction driver interprets transaction programs by walking a virtual PC through the transaction program, executing code corresponding to the current opcode. A set of basic opcodes are predefined by the transaction driver. In the case of Swift these are essentially Read, Write, and Synchronize. The implementor designs a system by defining transaction programs using the basic opcodes and, when necessary or convenient, by defining additional custom opcodes. Note that when an instruction executes it has complete access to any system data without need of synchronization since each instruction constitutes a critical section.

All communication between nodes is supervised by the cooperating transaction drivers. The transaction driver, using an interface to the underlying communication system, is responsible for managing the transmission and reception of both transaction programs and the communication required to effect the transaction programs. The transaction drivers handle nacks, retransmits, and timeouts. Higher level code (i.e., transaction programs and both predefined instruction and custom instruction code) never has to deal with acks, nacks, etc., similar to the manner in which assembler programmers do not deal with internal machine timing and most hardware error recovery. All code requiring communications programming has been collected into the transaction driver.

The operation of advancing the virtual PC includes assuring that required communication has occurred. All I/O is activated asynchronously. The transaction driver never blocks awaiting I/O if it has any other virtual PC's that it can advance. As soon as any communications activity completes, the transaction driver attempts to advance the effected transaction program's PC.

Communication failures result in timeouts or garbaged reception. Both result in PC resets occurring within both the local and remote transaction programs. This is effectively a nack, resulting in a resync of the distributed program. Note that a transaction program is never discarded until the program has successfully completed execution. This implies that burst mode protocols where immediate acknowledgments are not required are possible.

The error protocol is now described in more detail. The following two situations exist:

- Timeout: Since instructions are executed in 'the pipe' till we block awaiting communication completion, all timeouts correspond to a given local instruction that is blocked (stalled). This instruction is part of a distributed instruction pair, and at transaction program compilation time the PC's of both the local and remote instructions that make up the pair are known. Each instruction is tagged with the PC of its distributed partner instruction. This format is somewhat remenisent of drum machines where each instruction contained the PC of the next instruction. Upon timeout, a RESTART request is sent to the other transaction driver containing the PC of the remote instruction at which the cooperating transaction program should resume execution. Effectively, a RESTART forces a jump to a predefined location in the remote transaction program, syncing both the local and remote halves of the distributed transaction program.
- Out of Order packet. The header of each received packet contains a copy of the remote instruction that resulted in the packet's transmission and a target transaction program ID. The remote instruction copy contains the PC of the local instruction corresponding to the remote instruction (the *match* PC). If the local node does not have the given transaction program, the packet is assumed to be really lost and is discarded. If the incoming match PC is less than the current PC, the packet is assumed to be a duplicate and is also discarded. If the match PC is greater than the current PC, some packets have been lost, perhaps due to overrun. The packet is discarded and a RESTART sent to the remote side containing the remote PC corresponding to the current local PC.

Upon reception of a RESTART the local PC of the corresponding transaction program is simply forced to the indicated PC. Note that nothing keeps track of whether a RESTART has been transmitted other than a timeout. If the restart is lost, a timeout to the same blocked instruction will reoccur and another RESTART will be generated. After transmitting a RESTART, either communication from the expected remote instruction is received, in which case we proceed, or another out of sync instruction is received, in which case a new RESTART is issued. If a RESTART is received while awaiting the response from a RESTART, the received RESTART is respected, and the local PC reset. Note that restarts can never 'unwind' activity back past the beginning of the current transaction program.

1.3.2 The Object Oriented Model

Object oriented systems are becoming widely used to implement direct manipulation windowing applications. In such an environment multiple 'real-time' events (such as mouse movements, button clicks, and key strokes) may impact a large collection of program objects (windows, icons, menus, etc.). The object oriented system sorts out the mapping of real-time events to program objects. In theory, the programmer just defines simple self-contained reactive objects and the object oriented system provides the real-time program skeleton that dispatches events to the appropriate object at the appropriate time. Many ugly real-time issues can thus be avoided by programming to the object oriented system. This aspect of object oriented programming harks back to the initial use of object orientation to support simulation environments.

An example of a current object oriented environment is X-windows. The layers of a typical X-windows protocol stack are:

- X lib – distributed I/O primitives.
- X Intrinsics – 'real-time' object oriented dispatcher, and basic object set (widgets).
- Motif – Full widget set defining a specific look and feel.

In theory, it is easy to implement alternative look and feel environments by simply writing alternate widget sets at the Motif level. The real-time aspects of the windowing system have been hidden in the X Intrinsics layer and its basic widget set.

Consider the following Swift/RAID functionality stack, analogous to the previous X-window stack. The indicated files contain the corresponding Swift/RAID-5 functionality.

- *dgram.c* – Network I/O primitives.
- *trans_driver.c* – The transaction driver, which implements the real-time event dispatcher, distributed objects, and a basic object set.
- *swift_r5.c* – Provides the object set and functionality specific to RAID-5 (as opposed to RAID-4, for instance).

The above 3 files are the core of the Swift/RAID-5 implementation. There is one additional support file, *compile.c*, which provides primitives to assemble transaction programs. Each server node in a Swift/RAID system runs a generic server, file *swift_server.c*. The server's primary function is to simply run the transaction driver. The above files are linked into a library, *trans.a*. Both the generic server and client application programs link to this library. Aside from *.h* and debug files, these are the only files in the Swift/RAID implementation.

A motivation for this layering is to support experimentation with different RAID schemes simply by replacing the *swift_r5.c* file (for instance, with *swift_r4.c*), in the same manner that X-windows supports multiple look and feel implementations.

Unlike an object oriented system intended for a user interface, which is usually functionally rich and often suffers from poor real-time performance, the object orientation provided by *trans_driver.c* is very lightweight and specific to the Swift/RAID requirements.

All Swift entry points, (*swift_open, swift_read, swift_write*, etc.), are located in *swift_r5.c*. When an application issues a request to one of these procedures, *swift_r5* generates both a set of objects and a set of logical events. When the events drive the objects to completion, the request is satisfied. The object definitions contain the object's methods. Objects are instantiated using the functions in *compile.c*. Events are controlled by the order in which objects are assembled into sequences as a result of *swift_r5* flow of control. A specific event will occur when the object to which it corresponds is the current object in the sequence. Note that at this level events are high level logical events, e.g., 'write a remote file block'.

The *trans_driver()* routine in *trans_driver.c* hides real-time object invocation, object sequencing, object distribution, and communication between distributed objects. Most Swift/RAID objects can be considered as one element in a pair of objects, one local and one remote. This distributed pair constitutes a distributed object. When a method of one of the objects is invoked, it results in activity at both objects. The *trans_driver()* handles all physical events that tie the pair together, such as message invocation, timeouts, and retransmit requests. These low level events are not seen by the programmer dealing in logical events at the *swift_r5* level. The *swift_r5* programmer does see node failure as an event since this requires restarting a Swift request using a different approach, i.e., performing parity calculations to restore data.

The *trans_driver.c* file also includes the basic primitive objects used to implement any Swift/RAID system. These are basically Read, Write, Await Sync, Send Sync, and Delta Wait. A method code in each object indicates what the object is and what code is required to execute the indicated function. In addition to the above exported functionality, internal methods are provided for restart, migrating objects, and termination. These objects provide minimal encapsulation. They all contain 3 data fields that are usually data values or pointers to external resources such as buffers.

As with Motif widgets inheriting functionality from an X Intrinsics class widget, it is often the case when experimenting with a RAID protocol at the *swift_r5* level that a slight modification to primitive object behavior is required. An example is the operation 'read the old block and xor with the new data before writing the new data'. To support this operation, consider the *trans_driver.c* functionality as a base class and the *swift_r5* level code as a derived class. At each level functionality exists and methods can be executed. Objects defined by *tran_driver.c* support a *swift_r5* level method code. As in any object oriented system, *trans_driver()* invokes the methods as required, effectively walking down the class stack and invoking all applicable methods.

The event dispatcher in a real-time object oriented system is often the system bottleneck. Given an event, it must identify which object is effected and what method of that object should be invoked. This problem is avoided in Swift/RAID by preordering all objects in sequences such that the objects are in exactly the order in which they will be required. This is possible because *swift_r5* totally prespecifies the ordering in which logical events are to occur and the *trans_driver()* always deals with low level events with respect to the current object.

2. Introduction to the Swift/RAID Implementation

2.1 A Sketch of How Things Work

Every server node supporting a Swift file must have a running *swift_server*. There is a single Swift server program, *swift_server.c*. It is linked with the correct Swift/RAID library to produce a different version of the *swift_server* for each RAID implementation. For Swift/RAID-5 the resulting server is *swift_r5_server*. The application program is also simply linked to the Swift library that contains the appropriate Swift/RAID code.

Every Swift file the application opens has a *core_dir_t* structure maintained by the library. The Swift/RAID-5 *core_dir_t* structure is shown in Figure 2.1.

There is a different *core_dir_t* structure for each RAID implementation. The *core_dir_t* structure roots all data structures involved in the file's operation. File location, file status, node failure information, and the node and filenames of the Unix files that make up the Swift file are all described by the *core_dir_t* structure. The *core_dir_t* structure also includes a *trans_t* structure that roots all transaction driver activity. The format of the *trans_t* structure is shown in Figure 2.2. The *trans_t* structure contains asynchronous I/O masks, timeout values, and pointers to all transaction programs and transaction program contexts. The transaction driver is not aware of the format of the *core_dir_t* structure.

The *core_dir_t* structure contains arrays of *instruct_t* structures. The format of an *instruct_t* structure is shown if Figure 2.3. Transaction programs are constructed using elements from these arrays. There are two transaction programs assembled for every node involved in an I/O. The two constitute the halves of a cooperating distributed program. One transaction program will run on the local node and one on the remote node. The *instruct_t* buffers contained in the *core_dir_t* structure are divided up so that instructions are assembled contiguously. Instructions are not on linked lists.

```
typedef  struct  core_dir_t {

   int           cd_status;
   char          cd_swift_name[30];
   long          cd_block_size;
   long          cd_cur_file_loc;    /* Location in bytes within file. */
   int           cd_send_ahead;
   int           cd_num_nodes;
   long          cd_file_len;        /* Existing len of dist file in bytes. */
   int           cd_failed_node;
   int           cd_rebuild_flag;    /* 1=writes have occurred w/node down */
   char         *cd_missing_buf;
   char         *cd_parity_bufs;
   char         *cd_pad_blk;
   char         *cd_wrk_blk;

   char          cd_node_names[  MAX_NODES ][18];
   char          cd_file_names[  MAX_NODES ][ FILE_NAME_LEN ];
   int           cd_node_status[ MAX_NODES ];

   trans_t       cd_trans;
   instruct_t    cd_pgm_buf[        NUM_INSTRUCTS ];
   instruct_t    cd_remote_pgm_buf[ NUM_INSTRUCTS ];
   server_info_t cd_server_buf[ MAX_NODES ];

   strip_t       cd_strip_info[ MAX_XFER_STRIPS ];

   short         cd_sanity;
} core_dir_t;
```

Figure 2.1: The Swift/RAID-5 *core_dir_t* structure.

```
typedef  struct     transaction_t {
    int             tr_status;
    int             tr_num_servers;  /* Total number of servers.   */
    int             tr_server_id;    /* Our server ID              */
    long            tr_trans_id;     /* Current trans ID.          */
    fd_set          tr_async_fds_in;
    fd_set          tr_async_fds;
    int             tr_max_fd;
    server_info_t  *tr_server_info;   /* Base of server array.      */
    instruct_t     *tr_local_pgm;     /* Base of 2d instruct array. */
    instruct_t     *tr_remote_pgm;    /* ditto for remote pgms.     */
    int             tr_max_pgm_len;   /* longest # of instructions  */
    int             tr_num_cur_pgms;  /* # of execing transact pgms.*/
    int             tr_file;          /* File handle of backing str.*/
    char           *tr_callback_data; /* (cdir, ... ) */
    struct timeval  tr_timeout;       /* sec/usec */

} trans_t;
```

Figure 2.2: The *trans_t* structure.

```
typedef  struct     instruct {
    int             ins_opcode;      /* CC_XMIT */
    int             ins_extern_op;   /* upper level protocol operation. */
    int             ins_pc;          /* # of this instruction */
    int             ins_remote_pc;   /* Instruction paired with. */

    char           *ins_buf;         /* operand 1 */
    long            ins_byte_loc;    /* operand 2 */
    long            ins_len;         /* operand 3 */

    short           ins_sanity;      /* ID for sanity check. */
} instruct_t;
```

Figure 2.3: The *instruct_t* structure.

The *core_dir_t* structure contains an array of *server_info_t* structures. The format of these structures is shown in Figure 2.4. One of these structures is used to describe every node involved in support of the Swift file. The *server_info_t* structure contains the socket used to communicate with the node and the PC and status codes that provide virtual machine context for executing the local transaction program corresponding to that node. Each of the *server_info_t* structures corresponds to an array of local *instruct_t* instructions and an array of remote instructions. The first instruction in the local transaction program usually causes the remote instructions to be transmitted to the node described by the *server_info_t* structure. Note that when a Swift I/O request has been converted into a transaction, the total number of transaction programs under execution will be two times the number of nodes supporting the file, i.e., a local and remote transaction program for every node. All these transaction programs will be executing concurrently, and the distributed transaction program for one node may be executing at a very different rate from the transaction programs supporting another node (perhaps due to performance differences).

The Swift/RAID-5 *core_dir_t* structure contains an array of type *strip_t* that provides status information and parity calculation support for every strip involved in a single I/O. Note that a single Swift I/O can span many Swift file strips.

When the application issues a Swift request, the Swift library 'assembles' the transaction programs required to complete the transaction. The transaction driver then executes these programs. As the transaction driver advances through the *instruct_t* structures, it dispatches code as indicated by the instruction opcode. If an instruction initiates I/O it stalls. In this case, the transaction driver begins execution of another transaction program. Because the complete 'program' of the transac-

```
typedef  struct     server_info_t {
    int             svi_status;
    int             svi_server_num; /* 0..n array index of self.              */
    long            svi_file_len;
    int             svi_dest_pgm;   /* # of corresponding (dest) program       */
    int             svi_num_local_inst;
    int             svi_num_rmt_inst;
    int             svi_socket_num; /* Socket to communicate with given server. */
    int             svi_server_id;  /* Server's ID. */
    address_t       svi_address;    /* dest Network port address               */

    int             svi_pc;         /* Current instruct in pgm.                */
    int             svi_remote_pc;  /* PC on other side (remote) cooping with.  */
    char            svi_cc_stall_inst;  /* Condition code - stall.             */
    char            svi_cc_halt;        /* Condition code = Halt.              */

    char            svi_sync_buf[4];/* Rcv RESETs into here...                 */
    short           svi_sanity;
} server_info_t;
```

Figure 2.4: The *server_info_t* structure.

tion has been computed in advance, unexpected data I/O never occurs, that is, when a message is received, the operands of the current (usually stalled) instruction in the transaction program on whose behalf the message is received have been assembled so that the data will be delivered directly to its final destination.

All Swift entry points thus consist of two phases: a compilation phase in which *compile.c* routines *loc_compile()* and *rmt_compile()* are used to assemble transaction programs, and an execution phase driven by a call to *trans_driver()*.

2.2 Using the Swift/RAID Library

A Swift/RAID application is shown in Figure 2.5. This application simply writes the first strip in the Swift file 1000 times. The *ca_test3* Swift data file uses 3 nodes and has an 8K byte blocksize. Each strip thus consists of two 8K data blocks and a parity block. This test program performs a 'full strip' write.

The syntax of the Swift functions is shown in Figure 2.6. With the exception of *swift_dir_init()* these calls are all analogous to the corresponding Unix file calls. The only difference between Swift and Unix file I/O of which the programmer must be aware is that the *swift_read()* and *swift_write()* calls must occur in multiples of the blocksize specified in the 'plan' (directory) file.

The bulk of the Swift/RAID functionality in all 3 implementations is located in the *swift_read()* and *swift_write()* routines.

A Swift 'plan' file is shown in Figure 2.7. In this plan file, *test_01.dat* is a Swift/RAID-0 file, *test_03.dat* is a Swift/RAID-4 file, and *ca_test4* is a four node Swift/RAID-5 file. Note that all nodes in the Swift/RAID-0 file have a DATA keyword at the end of the line. In the Swift/RAID-4 file, the user specifically indicates which node is to be the parity node via the PARITY keyword. For Swift/RAID-5 files, all nodes are denoted as R5 nodes. File definitions for Swift/RAID-0, Swift/RAID-4, and Swift/RAID-5 can all be included in the same plan file. This greatly facilitates testing and comparison.

```
/* ca_test_01.c */

#include <stdio.h>
#include <fcntl.h>

#include  "swift.h"
#include  "swift_test.h"


char  *pgm_name = "ca_test_01";

char   buffer[ 16*8192 ];

long     start_mikes, end_mikes, delta_mikes, mikes_per_cycle;
int      num_iterations;
double   tot_bytes;

/*-------------------------------------------------------*/
main( int argc, char  **argv ) {
int      fin;
int      stat;
int      i;

  PRINTF "\n ca_test_01. 3 node write timing loop. \n" );

  stat = swift_dir_init( "plans" );
  if( stat < 0 ) crash( "Can't access directory!" );

  fin = swift_open( "ca_test3", O_RDWR, 0 );
  if(!fin ) crash( "swift_open" );

  start_mikes = get_mikes();
  tot_bytes   = 0.0;
  /*----------------------------------------*/
  for(i=0;i<1000;i++) {

      stat = swift_seek( fin, 0L );
      if( stat < 0) crash( "swift_seek" );

      stat = swift_write( fin, buffer, 2*8192 );
      if( stat < 0) crash( "swift_read" );

      tot_bytes += 2 * 8192;
   } /* end for */

  end_mikes = get_mikes();

  swift_close( fin );

  delta_mikes = end_mikes - start_mikes;
  PRINTF "\n Microseconds:       %ld. ", delta_mikes );
  PRINTF "\n Seconds:            %f ",
       ((float)delta_mikes) / 1000000.0 );
}
```

Figure 2.5: An Example Swift/RAID Application.

```
  swift_dir_init( swift_directory_file_name );

  swift_open( swift_file_name, file_flags, file_mode );

  swift_seek(  swift_handle, swift_file_location );

  swift_write( swift_handle, buffer, bytes );

  swift_read(  swift_handle, buffer, bytes );

  swift_close( swift_handle );
```

Figure 2.6: Swift/RAID Function Syntax.

```
// Swift plan file
//
// This described a set of distributed Swift files
//
// Format:
// plan-name  block-size  send-ahead  server1 file1 :  ... : servern filen;


test_01.dat     8192    1       maple   /wrk/brucem/swift_test.dat  DATA
                                oak     /wrk/brucem/swift_test.dat  DATA

test_03.dat     8192    1       maple   /wrk/brucem/swift_test.dat  DATA
                                oak     /wrk/brucem/swift_parity    PARITY
                                fern    /wrk/brucem/swift_test.dat  DATA

ca_test4        8192    1       maple   /wrk/swift_test/ca_test_r5_04.dat  R5
                                cedar   /wrk/swift_test/ca_test_r5_04.dat  R5
                                fern    /wrk/swift_test/ca_test_r5_04.dat  R5
                                dogwood /wrk/swift_test/ca_test_r5_04.dat  R5
```

Figure 2.7: The Swift/RAID Plan file.

3. Swift/RAID Implementation Internals

3.1 Introduction

This section is intended to be read in conjunction with a study of the Swift/RAID code. The code narratives in this section are intended to convey the 'middle-level' detail that is all to often lost between high-level conceptual exposition and detailed source code commentary.

3.2 swift_dir_init

The *swift_dir_init()* routine parses an ASCII file and builds a RAM resident description of known Swift files. The file, called a 'plan' file, maps Swift file names to node names/Unix pathnames.

The format of *swift_dir_init()* is:

```
swift_dir_init( swift_plan_file_name );
```

When called, *swift_dir_init()* works as follows:

- The specified plan/directory file is opened. Each Swift file will be described in RAM by a *core_dir_t* structure. These structures are organized in an array, the *cdirs* array. The *cdirs* array is cleared. Note that contiguous lines in the directory file are used to specify the components (nodes and files) that make up the Swift file. Each component is described on one file line.
- The open file is read via a loop that uses *fgets()* to read lines of the file. Maximum record length is 120 bytes. Empty lines and lines that begin with a '/' are skipped.
- The routines *is_empty(), is_white_space(), get_text(),* and *skip_white()* are used to parse the record. Routines *get_text()* and *skip_white()* scan a pointer through the record. Routine *get_text()* places contiguous text at which the pointer is currently pointing into an argument buffer, advancing the pointer over the text.
- The buffers filled by *get_text* are either copied into fields of the current *core_dir_t* structure, or converted to numeric fields. The *core_dir_t* structure contains two arrays that specify the nodes and the file pathnames of the components making up the Swift file. These are the *ca_node_names* and *ca_file_names* arrays.
- Status fields are initialized, the count of directory entries is updated, and the next record parsed. Upon completion of file processing, the plan file is closed.

3.3 swift_open

The syntax of *swift_open()* is:

```
swift_open( swift_file_name, file_flags, file_mode );
```

When called, *swift_open()* works as follows:

- The *cdirs* array is scanned and file names compared. If a *core_dir_t* structure is not found with the identical name, an error (-1) is returned.
- The *core_dir_t* structure is initialized for file operations (the current location set to 0 and failed node count set to -1).
- The transaction driver routine *trans_init()* is called. This routine takes elements of *core_dir_t* as arguments and does the following:
  - Computes the maximum buffer size available for 'compiled' transactions. There are two buffers within a *core_dir_t* in which transactions are compiled, one for local transaction programs and one for remote transaction programs. Each of these buffers is divided by the number of servers participating in the Swift file to form sub-buffers into which each server's specific transaction programs will be assembled.

  - Every open file has a *trans_t* structure that roots all transaction driver activity. This structure is embedded within the *core_dir_t* structure. It is now zeroed as are the program buffers.
  - An array of type *server_info_t* is also embedded in structure *core_dir_t*. These elements are used to describe that status of every other node involved in the Swift file. Included in this status is the status and context of the local transaction program communicating with that node. Note that when a transaction plan is generated, there will be a local transaction program for every node involved in supporting the Swift file. The *server_info_t* array is zeroed.
  - The *ins_sanity* field in all the instruction structures (*instruct_t*) in the instruction buffers is set to INS_TAG. This value is used as a sanity check whenever an instruction is passed as an argument or transmitted. Note that transaction programs are assembled into these *instruct_t* arrays.
  - The fields in the *trans_t* structure are now all initialized to reflect its 'open stream' status. Since the transaction driver operates without knowledge of the *core_dir_t* structure, the *tr_callback_data* field in the *trans_t* structure is set to point to the *core_dir_t* structure in which the *trans_t* structure is contained. This field is used by high-level (Swift-level) opcodes (as opposed to low-level transaction driver opcodes).

- The transaction driver routine *connect_to_node()* is now called. This is a very simple routine. It loops over the node and pathname arrays in the *core_dir_t* structure. For every node participating in the Swift file, routine *connect_to_node()* is called. This routine contains low-level *dgram.c* code, which performs straight-forward UDP socket connection. Connection is established to every server node in the following manner:
  - A timeout handler, *con_timeout_handler()*, is established that will handle SIGALRM.
  - Routine *get_node_address()* in *dgram.c* is called. It forms a network port address in the *address_t* field of the *server_info_t* structure. The global define PUBLIC_PORT specifies the port on the server node and the *gethostbyname()* system routine obtains the server address.
  - A local private datagram socket is allocated using system routine *bind_socket()*.
  - All Swift/Raid messages use only two communication routines - *send_message()* and *get_message()*. These routines are located in *dgram.c*. Routine *send_message()* takes as arguments the local private socket address, the remote socket address, a standard transaction header, and a variable sized transaction body. Routine *get_message()* is the inverse, with the address of the remote socket from which a message was received being returned. Both these routines communicate using two buffers - the first contains or receives a fixed sized header, and the second contains or receives a variable length message.
  - A 'forever' loop is entered that will only loop if a timeout occurs. This loop sends a connect message to the remote public socket. The body of the message is of type *connect_t*, which specifies the remote file name and open flags. A longjump buffer is then established to handle timeouts, an interval timer activated, and a *get_message()* call issued. The *get_message()* will block until a message is received. If the timeout occurs, SIGALRM will cause *con_timeout_handler()* to run which simply notes the timeout failure and, if no more than 4 timeouts have occurred, longjumps back to resend the connect message. When a response message from the remote server is received, the system *connect()* call is used to connect the local private socket to the remote local socket address returned by *get_message()*. The timer countdown is aborted and the SIGALRM handler revoked. Note that the remote server will have forked a copy of itself that communicates via a remote private socket rather than the PUBLIC_PORT socket.
  - If the received connect message status is successful, the remote Unix file has been opened by the server. If not, Swift error exits. The body of the message received by *get_message()* is of type *file_info_t*. This structure contains the length of the subcomponent file located

on the server. The sum of all these file lengths constitutes the total Swift file length. This information is needed for file operations such as *swift_seek()*.

– The local socket number is stored in the *server_into_t* structure and the status of the structure set to 'connected'.

- Upon completion of *connect_nodes*, the total Swift file size is accumulated by adding up the local file sizes stored in the *server_info_t* structures.
- Swift RAID-5 requires two work buffers for each open file. These buffers, *cd_pad_blk* and *cd_wrk_blk* are malloced and attached to the *core_dir_t* structure.
- The Swift file handle returned from the *swift_open()* call is simply the index in the *cdir* array of the file's *core_dir_t* structure.

3.4 swift_seek

The Swift seek call has the following form:

```
swift_seek(  swift_handle, swift_file_location );
```

The implementation of this call is trivial. The handle is used to locate the corresponding *core_dir_t* and the *cd_cur_file_loc* field is simply set to the specified file location. No network or transaction activity occurs. Note this implies that no Unix 'common file pointer' semantics are associated with the Swift file.

3.5 swift_write

The *swift_write()* routine writes blocks that are multiples of the Swift file blocksize. The definition of a Swift file specifies this size in bytes. A read or write can occur in any multiple of this size. Attempting to read or write a blocksize that is not a multiple of the file blocksize is an error. The syntax of *swift_write()* is:

```
swift_write( swift_handle, buffer, bytes );
```

When called, this routine works as follows:

- The file handle is used to locate the corresponding *core_dir_t* structure, and the structure is sanity checked. If the file is in CD_FAILED mode, more then one node has failed and an error is returned.
- The *trans_t* structure contained within the *core_dir_t* structure manages all transaction driver activity with respect to the Swift file. If the transaction status is TR_DEGRADED, one node has been lost and the *cd_rebuild_flag* is set. Currently nothing is done with this flag. In the future, every so many seconds or perhaps every so many Swift I/O calls, reconnection to the failed node should be attempted. Upon successful reconnection, a RAID rebuild phase should be activated.
- The *dist_program_init()* routine loops over every of the *core_dir_t*'s active *server_info_t* structures. Routine *init_svi_pgm()* in *compile.c* is called for each *server_info_t* structure. This routine simply clears the 'code exists' flag for the node, and sets the transaction program PC and the local and remote instruction counts to 0. Code can now be 'assembled' into the server buffers using *compile.c* routines.
- The particulars of the requested I/O are calculated, i.e., items such as *start_strip*, *end_strip*, *start_block*, and *end_block* are calculated. The request has now been converted into a request in physical RAID parameters. Note that large I/O requests often will span multiple contiguous strips.

- Depending upon the implementation, the base of the parity buffer array is either set to a work buffer or simply malloced. There needs to be one parity buffer for every strip involved in the I/O. The parity buffer is the same size as the Swift file granularity. The performance of malloc does not seem to be a big hit here, but the code is set up to support either approach. Field *cd_parity_bufs* roots the parity buffer array.
- If the current I/O starts beyond the logical end-of-file byte for the Swift file, a flag is set to indicate the Swift file needs to be extended to the appropriate size.
- An array of type *strip_t* located in the *core_dir_t* structure is initialized. This structure reflects the status of the I/O operation with respect to each strip.
- A loop that scans all strips involved in the I/O is the heart of the *swift_write()* logic which assembles the transaction programs to effect the write. This loop is now executed. For all strips involved in the I/O, the start block within the strip and the end block within the strip are identified. This results in 3 different situations which are handled as follows:
  - If the entire strip in involved in the I/O (all the blocks in the strip need to be written), *span_write()* is called. Note that in this case we will not need to read any parity information since we can calculate parity for the entire strip from data in the user's buffer. Note that the written strip may become the new end-of-file strip.
  - If I/O into the strip will extend the file (the strip will become the new end-of-file strip), *pad_write()* is called. This will occur whenever the end of the I/O transfer occurs within the strip one past the current last strip in the file. Note that *pad_write()* will write all blocks in the strip, even if the I/O does not specify that the I/O fill the strip. This strip will become the new end-of-file strip. Note that in this case we do not need to read any parity information since we can calculate parity for the entire strip.
  - If both the start and end of the I/O occur within the strip, *update_write()* is called. Note that in this case we may need to read the parity node. In this case flag *parity_update_flag* is set.
- All 3 of the previous routines compile transaction programs appropriate to their circumstances. They will be described later.
- Upon completion of the loop assembling instructions for all strips, a loop is executed that calls *t_init_pgm* for all servers involved in the I/O. This routine simply performs program fixup, which consists of calculating the size of both the local and remote programs, ORing the CC_END_OF_PROGRAM bit into the last opcode of both programs, and setting both the local and remote PCs to 0.
- Routine *trans_driver()* is then called to process the *trans_t* structure and its associated transaction programs. This routine will have the effect of concurrently executing all the transaction programs that have been assembled. Discussion of this routine is presented in the section on *trans_driver()*.
- Upon completion of the *trans_driver()* call, all blocks in all strips have been written. Note however, that the parity information may not yet have been written (i.e., in the case of *update_write()*). There is an opportunity here for additional concurrency in the implementation.
- If the *parity_update_flag* is set, only part of the strip was written and a parity update cycle is therefore performed. This is done by calling *dist_program_init()* to reinitialize transaction programs, *bld_parity_read_program()* to assemble the required parity update programs, and then *trans_driver()* to execute the parity update. Routine *bld_parity_read_program()* will be described later.
- Upon completion of the preceding, the parity buffer array is freed if one was allocated. If the I/O completed successfully, the current file location is incremented by the size of the I/O. The file length is updated if the file has been extended. A successful completion returns from *swift_write()* at this point, returning the length of the I/O to the caller.
- If the *trans_t* structure is marked as failed, we mark the *core_dir_t* structure as failed, and *swift_write()* returns with a failure. This usually indicates more than one node has failed.

- The only other alternative to the preceding two steps is that a single node has failed. The system then puts the Swift file in DEGRADED mode. It does this by scanning all the *server_info_t* structures belonging to the *core_dir_t* and locating the one with status SVI_NODE_DEAD. If a call to *trans_driver()* returned with a DEGRADED status, we are guaranteed to find one such node. The *cd_failed_node* field of the *core_dir_t* structure is set to indicate this node. Control then transfers back to the top of *swift_write()* and the entire I/O request is reprocessed. Since the file is now in DEGRADED status, the transaction programs assembled will be different and will reflect the required use of the parity node. In this manner the *swift_write()* will cope with a node loss occurring while the write is in progress.

Support routines for *swift_write()* are now described.

3.5.1 span_write

This routine is called when the I/O spans the entire strip. In this case parity can be calculated directly from the user's buffer. This routine works as follows:

- The parity buffer corresponding to the current strip is located in the parity array.
- For all the data nodes in the strip, the block within the users buffer that is to be written to that node is located, and *do_parity_calc()* called to accumulate parity into the strip's parity buffer. The first time *do_parity_calc()* is called it simply copies the input data block to the output parity block. Thereafter it performs an exclusive OR of the input data block and the parity block.
- The location of the parity node within the strip is now calculated. This is done via a simple modulo calculation. Given the absolute strip number and the number of servers, the parity node is located at 'strip % num_servers'.
- If the parity node has failed, the code to write the parity node is skipped. Otherwise, the following code is assembled into the transaction program of the node that is handling this strip's parity. This code will write the parity buffer to the parity block location in the Unix file on the parity node. The opcodes of the assembled parity code are shown in the following:

```
local                      remote

WRITE_CMD|CC_PARITY_FILE   WRITE_DISK|CC_PARITY_FILE
AWAIT_SYNC                 SEND_SYNC
```

- The CC_PARITY_FILE opcode bit is simply a debugging aid. The local WRITE_CMD instruction takes the parity buffer address as an operand, and the remote WRITE_DISK instruction takes the location of the parity block within the remote file as an operand. Both take the blocksize as an operand.
- A loop now generates code to write every node in the strip. A pointer is set to the start of the user's buffer (the first block). The target node for the block is computed. If the data block number is less than the number of the parity block (the same as the parity node calculated by the preceding modulo calculation), the data block number directly specifies the node number (i.e., is an index to the appropriate *server_info_t* structure). If the data block is greater or equal to the parity node, it is incremented by one to skip over the parity node. The parity node is thus 'invisible'. The following code is generated for each data block:

```
local                      remote

WRITE_CMD                  WRITE_DISK
AWAIT_SYNC                 SEND_SYNC
```

- The local WRITE_CMD instruction takes the address of the block within the user's buffer as an operand, and the remote WRITE_DISK instruction takes the location within the remote file to write the block as an operand. Both take the blocksize as an operand.

3.5.2 pad_write

This routine works in the manner of the previous *span_write()* except that it must pad the I/O to the length of the strip (and must write empty blocks into that section of the strip). Its overall operation is very similar to *span_write()*:

- The *core_dir_t's* pad block is zeroed (there is one allocated to every *core_dir_t*).
- The parity buffer for the strip is located and parity on all the user data blocks involved in I/O into the strip is calculated using *do_parity_calc()*.
- The parity node is calculated using a simple modulo calculation as described previously. Transaction code is generated to write the parity buffer to the parity node (in the proper file location).
- Code is assembled to write the user's data buffers to the appropriate location within the Unix files located at each corresponding node.
- A loop generates code to write as many null blocks as are needed to the remainder of the strip. The form of the assembled code is similar to that of the data blocks, with the exception that the pad buffer is used as input.

3.5.3 update_write

The *update_write()* routine handles the assembly of code to perform I/O which writes a subsection of a strip. This is the most difficult update case. It works as follows:

- The current location of the parity buffer for the strip is located in the parity buffer array. The parity node is located by a simple modulo calculation (absolute strip number % the number of servers in the strip). Note that the parity buffer array will contain as many parity buffers as there are strips involved in this I/O transfer.
- First, which of two strategies to use for the update is determined. The idea here is that if most of the strip is being written, it is sensible to read the blocks in the strip that aren't being written, and recalculate a new parity block for the strip. The alternative is to read the old parity block, read the old data blocks, XOR the old data with the new data, XOR this result with the old parity, and then rewrite the parity block.
- To do the strategy calculation, a loop iterates over all block numbers involved in the strip, generating the node numbers at which the corresponding blocks are located. If the block number is greater than or equal to the parity node, the node number is incremented by one to skip the parity node. We count the number of blocks that we are going to write. If we note that one of the blocks that we are going to write is the current *failed_node*, we jump to the 'read rest of strip strategy' as we will be unable to read the failed node to XOR the old data. If we note that one of the blocks that we are NOT going to write has failed, we jump to the 'XOR strategy' as we would be unable to read a block in the 'read rest of strip strategy'.
- Upon completion of the previous loop we have either forced one of the two strategies due to node failure location or we have counted the nodes we will read and those we will write. The strategy is now selected. Currently this is done by using the 'XOR strategy' if the number of *write_blks* is less than or equal to the number of *non_write_blks*.
- The appropriate strategy is now executed. The 'XOR strategy' is executed by calling *xor_update_write()* and then returning from *update_write()*. The 'read rest of strip' strategy is executed by falling into the remainder of *update_write()*.
- The code for 'read rest of strip' first loops over all the block numbers in the strip. The strip's parity buffer is filled by calculating parity on all the blocks to be written using the user's data buffers and *do_parity_calc()*.
- If the node in the strip containing the parity block has not failed, the strip's parity needs to be calculated. A loop over all the block numbers in the strip assembles the following code for blocks that we are NOT writting:

```
local                          remote

READ_RESULT;SMALL_WRITE_RD     READ_DISK
```

- Note that the SMALL_WRITE_RD is a high-level opcode that is processed by *swift_r5.c*, not by the transaction driver. The remote READ_DISK operands specify the data block location and the block size to be read. The local READ_RESULT instruction specifies the *cd_wrk_blk* buffer associated with the Swift file's *core_dir_t* structure as the destination buffer. The high-level opcode SMALL_WRITE_RD and the strip number are operands that are 'piggy-backed' on the instruction. After the data block from the remote READ_DISK instruction has been received, the local transaction driver will call *post_hi_handler()* in *swift_r5.c* before executing the low-level READ_RESULT code. The *post_hi_handler()* routine dispatches on the high-level opcode. In the case of SMALL_WRITE_RD this routine will execute *do_parity_calc()*, XOring the data that has been received in *cd_wrk_blk* into the appropriate strip's parity buffer in the parity array. Note that potentially both data and parity reads for strips involved in the I/O could be received out of order, that is, potentially a parity read for strip 1 could complete before a parity read for strip 0.
- A loop over all the block numbers in the strip now assembles code to write the user's data. Node numbers correspond 1:1 with block numbers up to the parity node, after which the node must be incremented by one so as to skip the parity node. For all data nodes to be written that have not failed, the following code is assembled:

```
local                          remote

WRITE_CMD                      WRITE_DISK
AWAIT_SYNC                     SEND_SYNC
```

- If the parity node within the strip has not failed, the *strip_t* structure describing the status of this strip for the duration of this I/O is initialized to indicate the number of reads the write operation has outstanding, the current number received, and a file location within the strip's parity node. This last location is where the new parity block is written when all the parity data from the strip has finally been computed.

3.5.4 xor_update_write

This routine is called within *update_write()* and handles the 'small write' case, that is, the case where a small subsection of the strip is written. In this case we want to XOR the new and old data blocks, then XOR the result with the old parity block, and finally rewrite the result as the new parity block. This routine works as follows:

- A loop over all the block numbers in the strip assembles code to write the user's data after XORing it with the old data. Node numbers correspond 1:1 with block numbers up to the parity node, after which the node must be incremented by one so as to skip the parity node. For all data nodes to be written the following code is assembled:

```
local                          remote

WRITE_CMD                      WRITE_DISK;SRV_READ_PARITY
WRITE_DISK;XOR_PARITY          WRITE_CMD;SRV_SEND_PARITY
```

- The local WRITE_CMD takes as operands the proper buffer pointer and buffer size. The remote WRITE_DISK specifies the location in the Unix file at which the block is to be written.
- The remote WRITE_DISK instruction blocks until a message is received containing the data to write (from the local WRITE_CMD). Upon receiving data, *post_dispatch()* in *trans_driver.c* will execute. This is the routine that will actually write the data. Before dispatching on any opcodes, however, this routine calls *post_hi_handler()*. This routine is in *swift_r5.c*. It provides

high-level opcode dispatching. Note that the server, not the client, is issuing this call. The server does not have a *core_dir_t* structure containing the relevant *trans_t* structure, rather it has a 'naked' *trans_t*. If a buffer is not currently allocated, a buffer of the file's granularity is malloced and attached to the *tr_callback_data* field of the *trans_t*. The old data block is then read into this buffer. The received data block (i.e., the block to be written into the file) is then XORed into the old data block.

- The remote transaction driver then executes the WRITE_DISK instruction and writes the new data block to the file.
- The local WRITE_DISK instruction takes the *core_dir_t's* pad block and the strip number as operands. The assembly of the strip number into this instruction is a subtle trick - it is used later to reestablish context. Other than the standard blocksize, the remote WRITE_CMD takes no operands.
- The transaction driver (*trans_driver()*) on the remote (server) side now issues the WRITE_CMD. The *pre_dispatch()* routine in *trans_driver.c* will execute this command (which transmits a data block). Since a high-level opcode exists, before executing the transmit code the *pre_hi_handler()* routine in *swift_r5.c* is called. This routine is a one line routine that simply assigns the buffer hanging off the *tr_callback_data* field to the *ins_buf* field of the instruction (i.e., it 'self modifies' the instruction). Thus the buffer that will be transmitted is the XOR of the old and new data. Note that this XOR has been performed on the server, not the client.
- On the local side, the WRITE_DISK instruction does not execute until the XORed data block sent by the WRITE_CMD is available. As with the server side, before the data is actually written by the low-level transaction dispatcher in *post_dispatch()*, the high-level *post_high_handler()* routine in *swift_r5.c* is called. This routine dispatches on the XOR_PARITY opcode.
- The XOR_PARITY code retrieves the strip number from the instruction that is being executed (recall the strip number was specified as one of the operands). The strip number is used to establish which parity buffer in the *core_dir_t's* parity array and which *strip_t* structure are effected by the XORed data that has arrived. The *do_parity_calc()* routine is then executed to XOR the previously XORed data block into the strip's parity buffer. The last thing this code does is 'zap' the opcode of the instruction being executed to NULL. This will cause the low-level transaction driver code to consider the instruction a no op. Thus the WRITE_DISK will not actually write anything to disk (in effect it wrote into the parity buffer).
- Note that high-level opcodes are always executed before low-level opcodes. This gives the high level opcode a chance to alter the low-level opcode. If the low-level opcode is set to NULL, no low-level processing of the instruction is performed.
- After the previous code has been assembled, the code to read the old parity block into the strip's parity buffer in the *core_dir_t*'s parity array is assembled. Note that all code for all nodes is executed concurrently. The actual execution of the parity read instructions may take place concurrently with the execution of the previously described XOR code. It does not matter when in the course of accumulating the parity for the strip the old parity block is received. The assembled code to read in the old parity and XOR it with the accumulated XOR in the parity buffer is:

```
local                              remote

READ_RESULT|CC_PARITY_FILE;XOR_PARITY  READ_DISK|CC_PARITY_FILE
```

- The local READ_RESULT takes the pad block buffer and the strip number as operands. The strip number will be used to reestablish strip context (recall that a number of strips could be concurrently processed by the sequence described here). The remote READ_DISK takes the location of the parity block within the server's Unix file as an operand.
- The CC_PARITY_FILE bits in the opcodes are only used for debugging.

- The remote READ_DISK causes the parity block to be read and transmitted. The READ_RESULT receives this block. Since this instruction has a high-level opcode, *post_hi_handler()* in *swift_r5.c* is called. The XOR_PARITY code operates exactly as if this parity block were an XORed data block, that is, it simply XOR's the old parity block into the strip's parity buffer as previously described.
- Upon completion of the assembly of this code, *xor_update_write()* initializes the *strip_t* structure describing the strip with the number of blocks that need to be XORed into the parity buffer, the number of blocks that have been received (0), and status.
- *xor_update_write()* has now assembled all code and built all supporting data structures required of the transaction programs. It now returns to *update_write()*, which will return to *swift_write()*, which will invoke *trans_driver()* to execute the assembled transaction programs.

3.5.5 bld_parity_read_program

Recall that a single Swift write I/O can write a number of strips. Each strip must have parity correctly calculated and updated. After the previously assembled transaction plan has been executed, any strip for which 'within strip' I/O was performed will have valid parity data in its parity buffer, i.e., the parity data in the *core_dir_t*'s parity array will be valid. Code is now assembled to write these parity blocks to their correct location on the respective parity nodes for the strips involved. Note that not all the strips in an I/O may require this parity update from the parity array - only the first and last strips involved in the I/O can be so effected. Routine *bld_parity_read_program()* is called at the end of *swift_write()* if such update is required. This routine works as follows:

- All the *strip_t* structures in the *core_dir_t*'s strip description array are scanned. If the strip has accumulated parity (*str_parity* == 1) then the parity node for that strip is calculated, and, if it is not the failed node, the following code is assembled:

```
local                         remote

WRITE_CMD|CC_PARITY_FILE      WRITE_DISK|CC_PARITY_FILE
AWAIT_SYNC                    SEND_SYNC
```

- The CC_PARITY_FILE bits in the opcodes are for debugging purposes only.
- The WRITE_CMD instruction takes the address of the parity buffer as operand, while the WRITE_DISK takes the location in the Unix file on the server node that is to be written.
- After the above code has been assembled for any strip with parity data, *init_all_programs()* is called. This is a general utility routine that simply scans all the server node's *server_info_t* structures, and calls *t_init_pgm()* for any servers that have assembled code. The *t_init_pgm()* routine performs transaction program fixup.
- Control returns to *swift_write()* which can now immediately execute *trans_driver()* to write the parity blocks.
- There is an opportunity here for additional parallelism. A given strip's parity block could be written as soon as the parity block is valid.

3.6 swift_read

The *swift_read()* routine is simpler than *swift_write()*. It also, however, must handle node failure in the middle of a transaction and must use parity to recreate data from a missing node. It works as follows:

- The Swift file handle is used to establish the *core_dir_t* and the *trans_t* which correspond to the Swift file.
- Routine *dist_program_init()* is called to initialize assembly for all servers supporting the file.

- The particulars of the requested I/O are calculated, i.e., items such as *start_strip*, *end_strip*, *start_blk*, and *end_blk*. The request has now been converted to physical RAID parameters. Note that large I/O requests will often span multiple contiguous strips.
- Depending upon the implementation, the base of the parity buffer array is either set to a work buffer or simply malloced. There needs to be one parity buffer for every strip involved in the I/O. The parity buffer is the same size as the Swift file granularity. The performance of malloc does not seem to be a big hit here, but the code is set up to support either approach.
- A loop now loops over the strip numbers of every strip involved in the I/O. This loop over all strips drives the assembly of code to implement the read transaction.
- For each strip, *chk_parity_fixup()* is called to determine if there is a failed node that contains a data block required for this strip's contribution to the I/O.
- Routine *chk_parity_fixup()* works as follows: if the file is not in DEGRADED status, all is well and no parity fixup needs to be performed. Otherwise, the nodes corresponding to each block in the strip are calculated. If one of these nodes is the failed node, we will have to do parity fixup. The *strip_t* structure describing this strip is located (there is one such structure inside the *core_dir_t*'s *cd_strip_info* array for every strip involved in the I/O). The location within the user's read buffer that corresponds to the data on the failed node is stored in the *strip_t* structure. This is the data block that will have to be reconstructed using parity information.
- A loop over all blocks within the current strip now assembles the required code. For all nodes directly involved in the read transfer the following code is assembled to read data directly into the user's buffer:

```
local                       remote

READ_RESULT;high_code       READ_DISK
```

- In the preceding code, *high_code* is the high-level opcode within the local instruction. This is set to HI_XOR_BUF if parity fixup is required and is null otherwise.
- The local READ_RESULT instruction takes as operands the buffer address at which to place the data, the block size, and the strip number. The remote READ_DISK instruction takes as operands the file location within the server's Unix file and the blocksize. Note that to fill the user's buffer numerous READ_RESULT instructions may be executed, one for every block within every strip that contains data that is deposited into the user's read buffer.
- For all data nodes indirectly involved in the transfer due to a parity fixup, i.e., those data nodes not read by the user which still need to be read to calculate parity (recall all blocks in a strip will need to be read to calculate parity), the following code is generated:

```
local                       remote

READ_RESULT;XOR_BUFFER     READ_DISK
```

- The local READ_RESULT instruction takes as operands the first address in the parity buffer array, the strip number, and the block size. The parity buffer address is used as a temporary work buffer for parity calculations. The remote READ_DISK takes as operands the file location within the Unix file on the server, and the blocksize. The XOR_BUFFER high-level opcode will cause code in *post_high_handler()* to be executed before the low-level READ_RESULT code is executed. Routine *post_high_handler()* in *swift_r5.c* locates the appropriate *strip_t* structure using the strip number operand embedded in the instruction. Recall that the *strip_t* structure contains a pointer to the space in the user's read buffer where data has to be reconstructed via parity. This pointer is used as the destination block in a call to *do_parity_calc()*. The source block for the *do_parity_calc()* call is the received data in the parity block. Thus the parity information in the 'missing' section of the users buffer will automatically build up until all blocks in the strip have been read. At this point the data in the user's buffer will be correct. Note that all data blocks will also participate in this process if needed because *high_code* will also be set to HI_XOR_BUF.

- In addition to the above reads of all the data blocks, the parity block within the strip must also participate in the previous process. The following code is assembled to accomplish this:

```
local                               remote

READ_RESULT|CC_PARITY_FILE;XOR_BUFFER   READ_DISK|CC_PARITY_FILE
```

- The above code is identical to the previous with the exception that it is assembled into the transaction program of the previously calculated parity node and has the CC_PARITY_FILE bit set in the opcode. This bit is only used for debugging purposes.
- At this point all the code to execute the transaction has been assembled. The loop over all strips is thus complete.
- A loop over all *server_info_t* structures associated with the current file's *core_dir_t* structure is performed. For any that have assembled code, *t_init_pgm()* is called to perform program fixup. The transaction programs are now ready to execute. The transaction driver, *trans_driver()*, is called to execute the entire transaction.
- Upon completion of *trans_driver()*, the transaction has either completed successfully or a node failure has occurred. The parity buffer array is freed regardless. Upon successful completion the current location within the file is advanced by the size of the read transfer, and control returns to the user.
- If more than one node contributing to the current file has failed, the *core_dir_t* structure is marked as FAILED and the call returns to the user with a failure status.
- If only one node has failed, the file must enter DEGRADED mode. This is accomplished by scanning the *server_info_t* structures (which are used as program context by the transaction driver) and identifying the node that has failed. The *core_dir_t*'s *cd_failed_node* field is set to the number of the node that has failed. In this case, the entire *swift_read()* request is now restarted. Since the file now has a failed node, the code that will be assembled to execute the transaction will be different then the code assembled on the first attempt.

3.7 swift_close

The *swift_close()* routine has the following syntax:

```
swift_close( swift_handle );
```

When called, this routine does the following:

- The handle is used to locate the corresponding *core_dir_t*, *server_info_t*, and *trans_t* structures. The *dist_program_init()* routine in *swift_r5.c* is called to initiate transaction program assembly.
- For each server node indicated by the *core_dir_t* structure, the following program is assembled:

```
Local                                   Remote

SEND_SYNC|CC_CLOSE                      AWAIT_SYNC
AWAIT_SYNC                              SEND_SYNC
```

- Routines *loc_compile()* and *rmt_compile()* perform the assembly, and routine *t_init_pgm()* performs program fixup.
- The transaction driver is invoked to execute the assembled transaction programs via a call to *trans_driver()*. The argument to this routine is the *trans_t* data structure.
- Upon completion of the *trans_driver()* call, all Unix files have been closed. Each server node is disconnected from the transaction by calling the transaction driver routine *disconnect_from_node()* with the corresponding *server_info_t* structure. Routine *disconnect_from_node()* simply closes the open socket to the server node, updates the *server_info_t* structure, and maintains the disconnect count.

- The two support blocks that were malloced for open file operations, the pad block and the work block, are freed.

3.8 trans_driver

The *trans_driver()* routine is the heart of the Swift/RAID system. Routine *trans_driver()* is driven by a *trans_t* data structure. Structure *trans_t* contains a pointer, *tr_server_info*, which points to the base of an array of type *server_info_t*. Each of the *server_info_t* structures provides the context of a transaction program. Transactions are created by assembling transaction programs, one per node, for all the nodes involved in the transaction. The *trans_driver()* routine then concurrently executes these programs as driven by events. Before transaction programs can be executed on behalf of a *trans_t* structure, the structure must be initialized by *trans_init()*. The *trans_inist()* routine is discussed in the section on *swift_open()*. Routine *trans_driver()* can be considered a distributed interpreter. This is how it works:

- Support routine *setup_async_io()* is called to setup an asynchronous receive mask to 'listen' to all the nodes involved in supporting the Swift file. Each node to which the current *trans_t* is connected (established via *swift_open()*) has a socket number stored in its *server_info_t* structure. The system macro FD_SET is used to set bits in the asynchronous I/O mask such that a set bit corresponds to every socket on which we wish to 'listen' for I/O. This mask is stored in the *tr_async_fds_in* field. Timeout values are also initialized.
- A loop scans the *server_info_t* array, and for every node for which code has been assembled, *activate_insruction_stream()* is called. This routine clears the node's condition codes in the *server_info_t* structure and sets up the initial local PC. It then attempts to execute as many instructions as possible. All opcodes have a PRE_EXEC bit set if they contain code that either does no I/O or transmits an I/O. Thus these instructions activate the I/O events which drive the system. These instruction are executed by calling instruction interpretation routine *pre_dispatch()*. After calling this routine, the condition codes are checked. If the instruction is stalled awaiting I/O completion, *activate_instruction_stream()* returns to the *trans_driver()* loop to locate and start execution of the next transaction program.
- After all instruction streams have been activated, an event driven infinite loop begins. This loop exits when there are no more executing transaction programs. Otherwise it waits for any I/O to occur via a *select()* system call using a clean copy of the *tr_async_fds_in* bitmap. This call also takes as an argument the timeout value previously inserted into the *trans_t* structure. Note that no special timeout code has to be established as *select()* will complete with a return value of 0 if the timeout period passes without any message reception. In this case, *timeout_handler()* is called to send out restart messages.
- If the return value from *select()* is non-zero, one or more bits will be set in the *tr_async_fds* mask corresponding to the sockets on which activity has occurred. A loop over the *server_info_t* array uses system macro FD_ISSET to compare the socket numbers used by each server with the set bits in the mask. If the corresponding bit is set, instruction interpretation within that server's transaction program is restarted at its current instruction via *exec_pgm_instructs()*. This routine drives all instruction interpretation activity subsequent to the initial *activate_instruction_stream()* code.
- Upon return from *exec_pgm_instructs()*, code that handled the results of an I/O and any code that initiated a new I/O will have been executed. The instruction stream for the node will be in a stalled state. If the return status from instruction interpretation is less than 0, a node has died. If the *trans_t* structure was already in degraded mode it enters failure mode, otherwise it enters degraded mode. Whenever a node failure occurs, *setup_async_io()* is recalled to recompute the bitmap of sockets on which we wish to 'listen'. This eliminates the nodes that have died from participating in transaction execution.
- This concludes the *trans_driver()* routine. Clearly, most of the low-level operation occurs within *exec_pgm_instructs()*.

3.8.1 exec_pgm_instructs

This is the routine that drives instruction interpretation. It works as follows:

- Context is established, including a global pointer to the current instruction, *cur_ins*. This is an event driven interpreter, so there is data to be retrieved from a socket. This data is obtained by calling *get_message()* in *dgram.c*. All transaction instructions (structures of type *instruct_t*) have a valid buffer address and length field (fields *ins_buf* and *ins_len*). If *get_message()* cannot be completed, the corresponding node is assumed to have died. Routine *get_message()* always returns the message 'header' in addition to the variable length data buffer. The message header contains a copy of the remote instruction which resulted in the message.
- The error protocol is now executed. If a message has been received that does not belong to the current *trans_t* structure, it is discarded. If the remote instruction is not part of a distributed instruction pair that matches the current program, it is discarded. If the remote instruction is a RESTART, we call routine *restart()* to reset our execution location in the local program. If the remote instruction 'matches' a PC less than the current local PC, we discard it as a duplicate. Recall that all instructions are tagged with a 'match' PC. This is the PC within the partner program with which they expect to execute in lockstep. If the remote instruction's 'match' PC is greater than the current local PC an overrun has occurred and messages have been lost. In this case *send_restart()* is called and the currently received message is discarded. Note that the local PC in this discussion is the PC located in the *server_info_t* structure which corresponds to the transaction program on which activity is occurring. The remote instruction's 'match' PC is determined by examining the copy of the remote instruction located in the message header.
- If we have discarded the received message, we do not proceed. Otherwise, a valid message has been received and *post_dispatch()* is called to process that part of a stalled instruction which occurs after an I/O is received. After calling *post_dispatch()* to complete interpretation of the stalled instruction, condition codes in the appropriate *server_info_t* structure are checked. If the HALT condition code is set, the transaction program has completed execution. In this case the count of executing transaction programs is decremented and control returns to *trans_driver()*. Otherwise, all possible instruction code is executed until an instruction stall condition occurs. Instruction interpretation is driven by a loop similar to that found in *activate_instruction_stream()*. This loop calls *pre_dispatch()* to execute code on the instruction's behalf that either does no I/O or activates an I/O. As before, if the transaction program completes, the count of active programs is decremented and control returns to the infinite *trans_driver()* loop. When an instruction stalls, control exits from the *exec_pgm_instructs()* execution loop.
- Besides waiting on I/O, an instruction stream can be stalled because it previously executed a DELTA_WAIT instruction. One of the uses of DELTA_WAIT is to introduce explicit timeouts to debug timeout handling code. DELTA_WAIT puts a transaction program to 'sleep'. To awaken the instruction stream, the appropriate *server_info_t* status is set to SVI_RESTART and the global *interrupt_restart_flag* set. If this flag is set *exec_pgm_instructs()* calls *restart_instruction_streams()*. This routine simply scans the *server_info_t* array. Any servers that have SVI_RESTART status set have their condition codes cleared and their PC advanced (thus skipping over the DELTA_WAIT instruction). The *activate_instruction_stream()* routine is then called to execute the instruction stream until an instruction stalls.

3.8.2 pre_dispatch

This routine contains instruction functionality, i.e., this routine dispatches on the current instruction opcode to execute code. The opcodes implemented here either do no I/O or initiate an I/O transmission. The *pre_dispatch()* routine contains low-level opcode 'primitives' that are specific to the Swift/RAID implementation. It works as follows:

- If there is a high-level opcode in the current instruction, *pre_hi_handler()* is called. This is not a transaction driver routine. It must be supplied in the Swift high-level code. The *pre_hi_handler()* code executes prior to the low-level *pre_dispatch()* code, so it can do whatever it wants with the current instruction.
- If the opcode is READ_DISK, the file handle in the current *trans_t* structure is used for an lseek to the location specified by the *ins_byte_loc* opcode field of the instruction. The *ins_buf* opcode of the instruction is set to a global *work_buf* and a read of *ins_len* bytes performed. Control then transfers to the common completion code which transmits the instruction and its data buffer to the other side of the connection. This opcode is usually executed on the server side of a transaction.
- If the opcode is a WRITE_CMD, control transfers directly to the common completion code which transmits the instruction and its data buffer. The instruction has been assembled and the data buffer established by code at the Swift level. This instruction is usually encountered on the client side.
- If the opcode is a SEND_SYNC, the instruction buffer is set to point to a special sync buffer that is inside the *server_info_t* structure. This buffer has a 'magic value' (17) as its first byte. Control is transferred to the common completion routine which transmits the sync instruction and the sync buffer.
- If the opcode is MIGRATE_PARTNER, control transfers directly to the common completion code which transmits the instruction and its associated buffer. The assembly routine *loc_compile()* always assures that MIGRATE_PARTNER is the first instruction in a client's local transaction programs. The buffer and length operands of this instruction are set by *t_init_pgm()* to contain the remote program that has been assembled (the remote half of the local program). Thus executing this instruction sends the remote half of a server's transaction program to that server.
- If the opcode is AWAIT_SYNC, the buffer and length opcodes of the instruction are set to point into the sync buffer within the *server_info_t* structure and the stall condition code is set. Control then returns to the caller of *pre_dispatch()*. This server's transaction program will not execute until the stall condition code is cleared. Note that the common completion code is not executed because no transmission is associated with this instruction.
- If the opcode is DELTA_WAIT, the instruction stream is simply marked stalled and control returned to the *pre_dispatch()* caller.
- The common completion code handles instructions which result in a transmission. The instruction's operands, that is, the buffer address and length fields, are valid at this point. Routine *xmit_instruction()* is called to build a message containing the instruction and its data buffer and transmit the message to the destination node.
- Upon completion of *xmit_instruct()*, the instruction opcode is checked to see if CC_END_OF_PGM is set. If it is this instruction is the last instruction in this server's transaction program, and the halt status code inside the appropriate *server_info_t* structure is set. This will stop instruction execution of this server's transaction program and give another transaction program a chance to execute.
- The last thing done by the *pre_dispatch()* common completion code is to advance the PC. This is required since the dispatcher has completed execution of the instruction.

3.8.3 xmit_instruct

Given an instruction with valid *ins_buf* and *ins_len* operands, this instruction bundles the instruction and the buffer into a message and transmits the message over the connection in the *server_info_t* structure on whose behalf the instruction is executing. The message header has a standard format which contains a transaction ID and a destination program ID. These are used by the receiving side's error protocol. The entire instruction (an *instruct_t* structure) is copied into the header. An instruction is currently 30 bytes long. The routine *send_message()* in *dgram.c* is then called to transmit the header followed by the buffer as a single message. The *send_message()* routine uses the socket number in the *server_info_t* structure for transmission.

3.8.4 post_dispatch

This routine contains instruction functionality, i.e., it dispatches on the current instruction opcode to execute code. The *post_dispatch()* routine executes instructions that execute after an I/O has been received. When a message is retreived in *exec_pgm_instructs()* by *get_message()*, the stalled instruction execution is resumed and *post_dispatch()* dispatches on the opcode to finish instruction execution. It is important to recall that all data messages are expected messages, i.e., a message is always 'sent' to an instruction that has been assembled to expect that message. This is true even when the data message is transmitted before the instruction is executed. Recall that the *select()* system call simply indicates which socket contains data, and subsequent *get_message()* calls access this data when the instruction is executed by *exec_pgm_instructs()*. Routine *post_dispatch()* works as follows:

- The stall condition code is cleared. The global *cur_ins* points to the stalled instruction. The received instruction in the message header is located (pointer *rcv_ins*). This is a copy of the remote instruction that resulted in the received message.
- Routine *post_hi_handler()* is called if the current instruction has a high-level opcode. The *post_hi_handler()* is defined in *swift_r5.c*. The high-level code can do whatever it desires to the instruction. If it sets the low-level opcode to NULL, low-level opcode dispatching will be skipped.
- If the remote instruction opcode is a RESTART, the *restart()* routine is called and control returns to *exec_pgm_instructs()*. The *restart()* routine resets the PC within the current transaction program. The current instruction (PC) will thus probably no longer be valid.
- If the remote instruction opcode has the opcode bit CC_PATCH_PARTNER_BYTE_LOC set the operand field *ins_byte_loc* is copied from the received instruction to the current instruction. This is tricky - one of the members of the distributed instruction pair is 'self-modifying' the other. The support routine *hot_patch()* in *compile.c* provides a clean way for Swift level routines to assemble code that will do this. NOTE THAT THIS IS NOT USED IN Swift/RAID-5. This technique was used for small write parity updates in Swift/RAID-4.
- If the current instruction opcode is WRITE_DISK, the *instruct_t* fields *ins_byte_loc, ins_buf,* and *ins_len* are used to perform the write I/O. A seek on the file handle within the *trans_t* structure (*tr_file*) is performed to the specified *ins_byte_loc* location, followed by a write of the buffer.
- If the current instruction opcode is READ_RESULT, there is no additional work to do on behalf of the instruction. The buffer was assembled so that received data was deposited directly into the proper location in the user's buffer.
- If the current instruction opcode is AWAIT_SYNC, a SYNC has hopefully been received. The sync buffer within the *server_info_t* structure is examined and the first byte checked to assure that it is the 'magic' sync value 17. If the CC_CLOSE bit of the RECEIVED instruction opcode is set, the local *trans_t* status is set to DONE, i.e., the connection is closed to further Swift file functions.
- With the exception of RESTART, all of the preceding routines terminate in a common handler that simply checks for the end of the instruction stream and advances the transaction program's PC. The PC is located within the *server_info_t* structure corresponding to the program. The CC_END_OF_PGM bit in the opcode indicates the last instruction in the program. The halt condition code is set if this opcode bit is 1.

3.8.5 send_restart

When a timeout occurs, or a receiver detects an overrun (lost packets), routine *send_restart()* is called to perform a 'shoulder tap' on the partner transaction program. This routine simply handcrafts an instruction (a structure of type *instruct_t*). The opcode (*ins_opcode*) is set to RESTART and the buffer set to the sync buffer contained in the current program's *server_info_t* structure. The current local transaction program instruction has a 'match' PC that is the instruction on the remote side that should execute in 'lockstep' with the current instruction. This value, *ins_remote_pc*, is placed in the *ins_pc* operand of the handcrafted instruction. The handcrafted instruction is then transmitted to the cooperating transaction program via the standard *xmit_instruct()* routine used for *trans_driver()* message transmissions.

3.8.6 restart

When a RESTART instruction transmitted by *send_restart()* is received, the *restart()* routine is called by *exec_pgm_instructs*. This routine simply sets the PC of the specified transaction program to the value contained in the *ins_pc* field of the received instruction (the 'match' PC). Note that PC's index into the transaction program, i.e., they are not RAM addresses. The *server_info_t* structure corresponding to the transaction program contains the PC and status codes. The status codes are initialized to 0. This will cause instruction execution to resume on the specified transaction program at the new location.

3.8.7 interrupt_restart

The *interrupt_restart()* routine is not used by *swift_r5.c*. This routine was used in conjunction with the DELTA_WAIT instruction to simulate timeouts and debug timeout handling code. This routine, when called by high-level *swift_r4.c* code, effectively restarts a sleeping transaction program. The transaction program was put in the 'sleep' state by the execution of the DELTA_WAIT instruction.

3.8.8 timeout_handler

The *timeout_handler()* routine is called when the asynchronous 'listen' performed by the system *select()* call in the *trans_driver()* workloop receives no messages within the timeout period. The timeout period is specified by the *tr_timeout* field in the Swift file's *trans_t* structure. Routine *timeout_handler()* loops over all the *server_info_t* structures belonging to the *trans_t* structure on which the timeout has occurred. The system macro FD_ISSET is used to determine if we were listening to the socket embedded in each *server_info_t*. The *send_restart()* routine is called given the following conditions: 1) we were listening; 2) the corresponding transaction program is not halted; and 3) the program's current instruction is not DELTA_WAIT. Routine *send_restart()* sends a RESTART instruction to the remote side to attempt to reactive the corresponding partner transaction program. Thus, all transaction programs in the transaction plan will be restarted.

3.9 compile.c

This file contains three routines of primary interest: *loc_compile()*, *rmt_compile()*, and *t_init_pgm()*.

Routine *loc_compile()* simply initializes the *instruct_t* structure at the current local PC location and advances the PC. All fields initialized are supplied as arguments to *loc_compile()*. If the current PC is zero, a MIGRATE_PARTNER instruction is generated before the instruction is initialized. The MIGRATE_PARTNER instruction will cause the 'partner' transaction program to be transmitted to the remote node.

Routine *rmt_compile()* simply initializes the *instruct_t* structure at the current remote PC location with values supplied as arguments, and advances the PC.

The *loc_compile()* and *rmt_compile()* routines should always be called as a pair. Each assembles into its instruction the PC that is current with respect to the other, that is, the two called as a pair will assemble instructions considered a 'match' pair.

Routine *t_init_pgm* performs final fixup on a transaction program. It calculates the size of both the local and remote programs, ORes the CC_END_OF_PROGRAM flag into the last opcode of both programs, and sets both the local and remote PCs to 0.

3.10 swift_server.c

The program *swift_server.c* must run on all the server nodes. The function of this program is essentially to receive transaction programs and fork off a child that will use *trans_driver()* to execute the program. The *main()* routine works as follows:

- A socket is allocated using the global define PUBLIC_PORT. This is the socket that is used for connect messages.
- The body of *main()* is an infinite loop. It first calls *build_connection_dbs()*. This routine builds a *trans_t* structure. The *trans_t* structure is allocated from heap. Since no *core_dir_t* structure exists, the instruction buffers and *server_info_t* buffers must be explicitly allocated as well. The *trans_init()* routine is used to initialize the structure for transaction driver operations.
- The server workloop waits on call *get_message()* until a message is received. A received message should be a connection request that specifies the name of a Unix file that will be used to support Swift I/O. This information is described in a *connect_t* structure that constitutes the body of the received message.
- When a message is received, the server immediately forks a child. When the child completes, routine *free_connection()* is called to free up the *trans_t* structure. The workloop is then repeated. Note that the child also frees the *trans_t* structure when it exits, but that this does not effect the parent's heap (the child has a copy of the parents heap so its operations do not effect the parent). The *trans_t* structure is allocated and initialized in the parent so as to provide a minor performance enhancement, i.e., so allocation does not occur after a connect request has been received. The connect process is already slow, in that it involves a *fork()* and an *open()* call.
- The child allocates a private local socket and connects this socket to the address of the client's socket using the system call *connect()*.
- The Unix file is now opened, and the file size determined. Routine *send_message()* in *dgram.c* is used to reply to the client and return the local socket and file information.
- Once connected, any number of transactions may occur over the connection. Transaction processing will consist of a transaction program being received by the server, followed by the execution of the transaction program.
- The first message of every transaction will not be processed by the *trans_driver()* routine. The first message of a transaction is the transaction program itself. This message is processed by *bootstrap_load()*. Thus the heart of the child server is a loop that executes *bootstrap_load()* followed by *trans_driver()* until the *trans_t* structure is marked DONE. Before this loop can be started, an asynchronous 'listen' mask needs to be established for *bootstrap_load()*. This is done via *setup_async_io()*, which is discussed at the start of the section documenting the *trans_driver()*.
- Routine *bootstrap_load()* performs a *select()* system call and uses the asynchronous mask and FD_ISSET to determine which node has sent the message. See the *trans_driver()* documentation for additional detail. The *get_message()* routine in *dgram.c* is then used to retrieve the message into the appropriate *instruct_t* buffer (i.e., the program buffer for the 'remote' side transaction program). The *server_info_t* structure matching the node from which the program was received is initialized to start executing the program at PC 0. The program is sanity checked for consistency.

- The one operand field of the received transaction program that *compile.c* could not assemble is the receive buffer address for data blocks that are sent to a remote instruction from its matching local instruction. These operands are all set to the address of a receive work buffer, *buf_1*.
- The transaction driver has been previously described. See the *tran_driver()* documentation. After this routine has been completed, the transaction is complete. Additional transactions, each corresponding to a Swift function request, are repeated using the *bootstrap_load()* – *trans_driver()* loop until the connection is closed.

4. Conclusions

The Swift/RAID system is an implementation of the core functions required for any RAID system. The present system can be used as a basis for additional enhancements, research, and perhaps for applications. Things that remain to be done include:

- After a node has failed, it needs to be periodically rechecked to see if it has returned to operation. If so, a rebuild operation needs to be started and integrated with current operations.
- The rebuild program can be written either as a separate utility or run as a concurrent transaction program. Rebuild can be implemented by reading all the nodes in each strip other than the rebuild node and using parity to calculate the value to write to the rebuild node.
- It would be fairly straight-forward to provide a more realistic directory service than the use of the 'plan' file.
- It would be straight-forward to add a client library cache which would permit byte-level user requests. Such a cache should be designed so that transfers that are Swift file block aligned will bypassed the cache, i.e., such an I/O will performed in the same manner as a current request.
- Most timing conditions and timeouts can be forced using the DELTA_WAIT instruction and *interrupt_restart()*. To bring the current Swift/RAID code up to 'beta release' quality, this should be done and all such error handling code validated. The current Swift/RAID-5 system has had almost no such testing due to scheduling constraints. Such a cycle was performed with the Swift/RAID-0 implementation and proved extremely effective.
- Additional asynchronism and burst mode protocols can easily be investigated by eliminating the assembly of AWAIT_SYNC and SEND_SYNC instructions on a per read/write basis.

The general conclusions that can be drawn from this implementation effort are:

- RAID systems can be built in the distributed server environment typified by Swift;
- The distributed virtual machine approach described here solves the distributed concurrent programming problem to the degree necessary to program the RAID implementation;
- The distributed virtual machine approach presents a non-trivial programming task that is reminiscent of low-level assembler programming or microcode programming. While this may prove intimidating to some, it provides a solution guaranteed to work in the same sense that microcode can be guaranteed to cope with hardware asynchronism.

The last two items are not particularly new or surprising. Variants of the virtual machine technique have long been used to implement multi-threaded servers in uniprocessor environments. The preceding conclusions confirm observations that have been made numerous times in the uniprocessor case, e.g., in [Allworth 81] and [Beizer 83].

References

[Allworth 81] S.Allworth, *Introduction to Real-Time Software Design*, Springer-Verlag, 1981.

[Beizer 83] B.Beizer, *Software Testing Techniques*, Von Nostrand Reinhold, 1983.

[Cabrera and Long 91] L.Cabrera, D.Long, *Swift: Using Distributed Disk Striping to Provide High I/O Data Rates*, Computing Systems, Vol. 4, No. 4, Fall 1991, pp. 405-436.

[Emigh 92] Aaron T. Emigh, *The Swift Architecture: Anatomy of a Prototype*, UCSC, 1992.

[Hartman and Ousterhout 92] J.Hartman and J.Ousterhout, *Zebra: A Striped Network File System*, in *Proceedings of the USENIX Workshop on File Systems*, May 1992.

[Katz, et al., 89] R.Katz, G.Gibson, D.Patterson, *Disk System Architectures for High Performance Computing*, Proceedings of the IEEE, Vol. 77, No. 12, Dec 1989, pp. 1842-1858.

Index