Swift/RAID: A Distributed RAID System

Darrell D. E. Long, Bruce R. Montague†
Computer and Information Sciences
University of California, Santa Cruz

Luis-Felipe Cabrera
Computer Science Department
IBM Almaden Research Center

Abstract

The Swift I/O architecture is designed to provide high data rates in support of multimedia type applications in general purpose distributed environments through the use of distributed striping. Striping techniques place sections of a single logical data space onto multiple physical devices. The original Swift prototype was designed to validate the architecture, but did not provide fault tolerance. We have implemented a new prototype of the Swift architecture that provides fault tolerance in the distributed environment in the same manner as RAID levels 4 and 5. RAID (Redundant Arrays of Inexpensive Disks) techniques have recently been widely used to increase both performance and fault tolerance of disk storage systems.

The new Swift/RAID implementation manages all communication using a distributed *transfer plan executor* which isolates all communication code from the rest of Swift. The transfer plan executor is implemented as a distributed finite state machine which decodes and executes a set of reliable data transfer operations. This approach enabled us to easily investigate alternative architectures and communications protocols.

Providing fault tolerance comes at a cost, since computing and administering parity data impacts Swift/RAID data rates. For a five node system, in one typical performance benchmark, Swift/RAID level 5 obtained 87% of the original Swift read throughput and 53% of the write throughput. Swift/RAID level 4 obtained 92% of the original Swift read throughput and 34% of the write throughput.

Keywords: Swift architecture, RAID, data striping, client-server data transmission, network data service, distributed atomic operations, concurrent programming, distributed state machines, real-time distributed programming.

1 Introduction

The Swift system was designed to investigate the use of network disk striping to achieve the data rates required by multimedia in a general purpose distributed system. The original Swift prototype was implemented during 1991, and its design and performance was described, investigated, and reported [Cabrera and Long, 1991, Emigh, 1992]. A high-level view of the Swift architecture is shown in Figure 1. Swift uses a high speed interconnection medium to aggregate arbitrarily many (slow) storage devices into a faster logical storage service, making all applications unaware of this aggregation. Swift uses a modular client-server architecture made up of independently replaceable components.

Disk striping is a technique analogous to main memory interleaving that has been used for some time to enhance throughput and balance disk load in disk arrays [Kim, 1986, Salem and Garcia-Molina, 1986]. In such systems writes scatter data across devices (the members of the stripe) while reads 'gather' data from

†Supported in part by the National Science Foundation under Grant NSF CCR-9111220 and by the Office of Naval Research under Grant N00014–92–J–1807

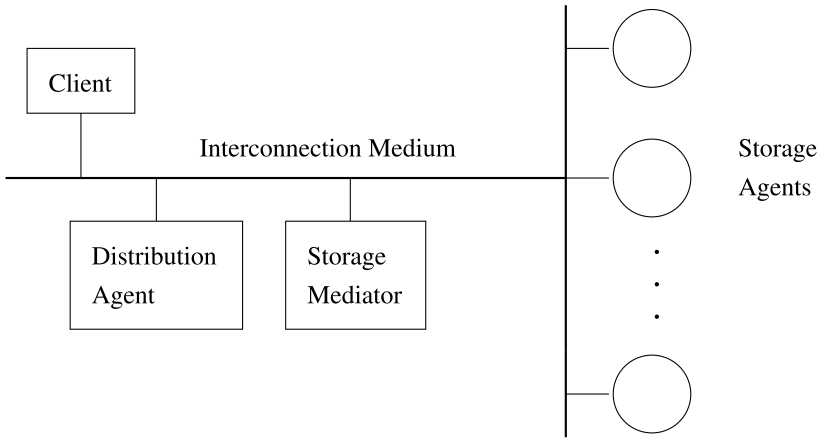


Figure 1: Base components of Swift architecture.

the devices. The number of devices that participate in such an operation defines the degree of striping. Data within each data stripe is logically contiguous. Swift uses a network of workstations in a manner similar to a disk array. A Swift application on a client node issues I/O requests via library routines which evenly distribute the I/O across multiple server nodes. The original Swift prototype used striping solely to enhance performance.

Currently, RAID (Redundant Arrays of Inexpensive Disks) systems are being used to provide reliable high-performance disk arrays [Katz et al., 1989]. RAID systems achieve high performance by disk striping while providing high availability by techniques such as RAID levels 1, 4, and 5. RAID level 0 refers to simple disk striping as in the original Swift prototype. RAID level 1 implies traditional disk mirroring. RAID levels 4 and 5 keep one parity block for every data stripe. This parity block can then be used to reconstruct any unavailable data block in the stripe. RAID levels 4 and 5 differ in that level 4 uses a dedicated parity device while level 5 scatters parity data across all devices, thus achieving a more uniform load. The fault tolerance of levels 4 and 5 can be applied to any block structured storage device, indeed, the level 4 technique appears to have been developed originally to increase the fault tolerance of magnetic bubble memories [Patel, 1982].

When a device failure occurs during a RAID level 4 or 5 read, all other devices, including the parity device, must subsequently be read and the stripe parity calculated to reconstruct the data from the missing device. When a device failure occurs during a write, all other devices must be read, stripe parity computed, the remaining modified data blocks written, and the new parity written. A RAID level 4 scheme is illustrated in Figure 2.

For Swift to tolerate server failure and to investigate the performance of RAID levels 4 and 5, the Swift prototype was reimplemented to support RAID levels 0, 4, and 5. The new implementation is based on a distributed *transfer plan executor*. The transfer plan executor decodes and executes *instructions*. The set of available instructions constitute a language of atomic I/O operations designed specifically for the Swift/RAID architecture and implemented atomically with respect to the network. The executor is a distributed virtual state machine which decodes and executes this language, thus driving all data-transfer activities in the system and managing all network communication.

Transfer plans are sequences of atomic instructions. Every Swift/RAID user request is implemented by a unique set of transfer plans, called a *plan set*. The exact plan set implementing a particular Swift/RAID user request is generated at run-time by the Swift/RAID library when it is called by the user. Transfer plans are then decoded and executed atomically with respect to other transfer plans acting on the same Swift file. The executor guarantees Swift/RAID clients that request completion implies coherent and consistent

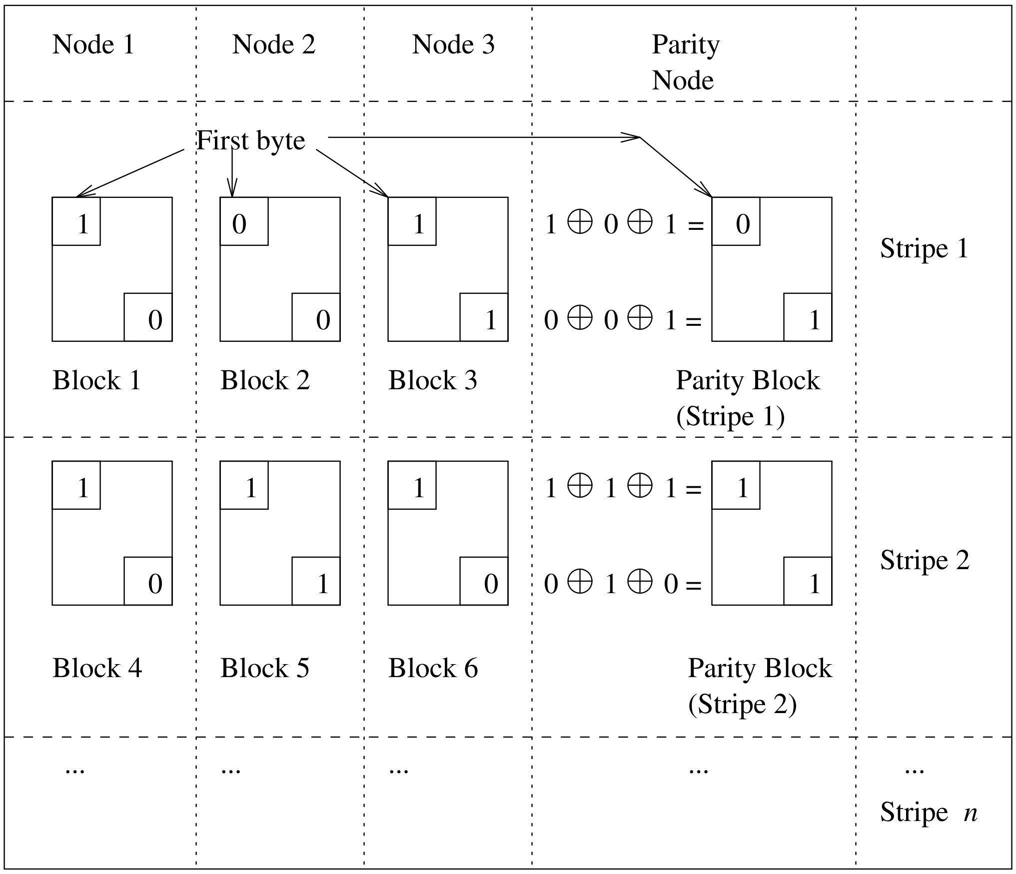


Figure 2: Operation of a four node RAID level 4 System. For every stripe, the corresponding parity block contains parity bytes for every byte in the stripe. If node 3 is lost, the bytes of Block 3 are reconstructed using stripe parity. The first byte is $1 \oplus 0 \oplus 1 = 0$ and the last byte is $0 \oplus 0 \oplus 1 = 1$. Data reconstruction must be done for every byte of the lost block.

completion of the corresponding Swift/RAID server I/O.

The *plan set* specifies a complete specification (including order and location) of the atomic primitives (*instructions*) that are to be executed to satisfy a particular distributed Swift/RAID operation. The plan set consists of *transfer plans* each describing a serially executed list of instructions. Transfer plans are generated in pairs, with each pair specifying the interaction between a single client and one of the Swift/RAID servers. There is a Swift/RAID server for each storage device making up a file. Each transfer plan pair consists of a *local* transfer plan, which specifies the instructions to be executed on the client, and a *remote* transfer plan, which specifies the instructions to be executed on the server. The local and remote plans in a transfer plan pair contain cooperating instructions. Each pair of cooperating local and remote instructions effectively defines a single distributed instruction. Each distributed instruction is generated such that a message from one side (local or remote) is required before the cooperating instruction can complete. Local and remote components of the plan pair execute in lock-step. The plan pair servicing each server executes concurrently with the plans servicing other servers; each plan pair can execute at an arbitrary rate relative to other plan pairs.

Transfer plan sets provide a mechanism to coordinate concurrent programming activity without requiring multiple threads or a single physical address space. Error recovery is simplified because each instruction in the plan is a simple atomic primitive. Additionally, this technique readily extends to single-copy communication protocols.

The clean error recovery model was particularly important as error recovery is fundamental to RAID operations. A single Swift/RAID user request involves the simultaneous coordination of a number of nodes. Both complete failure of a node and transitory communication failures must be handled transparently while in the midst of a user request.

The performance of the original Swift was compared with the new RAID level 0, 4, and 5 implementations. The performance of the Swift/RAID level 0 implementation was similar to the original Swift prototype, but Swift/RAID levels 4 and 5 write data rates were initially only 25% of the original Swift data rates. This was determined to result primarily from the computational cost of the exclusive-or parity computations required by RAID. After examining the generated code, a simple change from **byte** to **long** operations and from unoptimized use of the vendor supplied compiler to optimized use of the Gnu **C** compiler resulted in *peak* Swift/RAID write data rates approaching a more satisfactory 50-60% of the original Swift write rates. This peak result was not obtained on the average as performance over time exhibited a saw-tooth pattern varying by some 300 kilobytes/second. This was first thought to be an artifact of RAID striping, but was found to result from two subtle implementation errors which resulted in erroneous time-outs. Removing these errors resulted in the performance reported in §6.

The remainder of this paper is organized as follows: the atomic instruction approach is discussed in §2 and a functional description of the new implementation is provided in §3. The transfer plan executor is examined in §4 and software engineering aspects of the reimplementation are briefly considered in §5. Performance measurements are presented in §6. Related work is discussed in §7, followed by our conclusions and directions for future work in §8.

2 The Distributed Atomic Instruction Approach

A methodology based on table-driven distributed real-time state machines was used to implement the Swift/RAID prototype. This is similar to an approach that has long been used in programming real-time systems, and has been termed *slice-based* by Shaw, who characterizes it as one of the two basic approaches to real-time programming [Shaw, 1986]. Descriptions of systems using this approach are provided by MacLaren [MacLaren, 1980] and by Baker and Scallon [Baker and Scallon, 1986].

All Swift/RAID operations are defined as cooperating sequences of serially executed atomic primitives. The atomic primitives are called *instructions*. Each instruction corresponds to a C subroutine that implements the instruction. Each instruction's implementation is atomic with respect to Swift operations. This facilitates concurrent programming since *all* instruction implementation code occurs within a critical section. Swift/RAID *transfer plans* are formed by simply concatenating instruction data structures. The instruction data structure contains fields specifying *op-code* and *operand* elements. These are similar to the instruction fields of traditional machine language instructions, with op-codes indicating which implementation routine is to execute and operands supplying arguments to that routine.

Figure 3 shows the steps involved in servicing a Swift/RAID application request. The request results in the run-time generation of a unique *transfer plan set* consisting of a collection of transfer plans, each of which consists of an instruction sequence. The Swift/RAID library analyzes the request and generates a corresponding transfer plan set by issuing calls to two routines: **loc_compile** and **rmt_compile**. Each of these routines assembles a single instruction into the current location in a plan. For every remote node involved in the Swift/Raid operation, two cooperating plans are generated, one for the remote node and one for the local client node. The **loc_compile** routine assembles an instruction into the local plan and the **rmt_compile** routine assembles into the remote plan. These routines each take as arguments a node number, two op-codes, and three operands. One op-code drives low-level network activity and the other high-level RAID activity. Both local and remote instructions contain two distinct op-codes. The three operands often specify buffer address, transfer length, and file location, but may be used in any manner an instruction requires. The **loc_compile** and **rmt_compile** routines are almost always called together, thus generating a pair of cooperating local and remote instructions which in effect define a distributed instruction. The low-level op-codes specify operations such as reading a disk block into a buffer location, sending a buffer to another node, and writing a block to a file location. The high-level op-codes specify operations such as computing the parity of a buffer or another such RAID operation.

The application programmer is unaware of transfer plans. All plan assembly is performed automatically by the Swift/RAID library as it determines how to execute the request. The entire plan set is generated at the client node and contains a specific unique plan for each Swift/RAID server participating in the execution of the requested function. Each of the server plans cooperates with the corresponding plan executing on the client, that is, plans are generated in client-server pairs.

Because the implementation is factored into instructions, explicit synchronization programming is not required. The cost, however, is that all code must be written so that it never blocks on network communication, that is, network I/O cannot be requested via a synchronous system call that would block the transfer plan executor. An instruction implementation issues an asynchronous network request and then returns without awaiting completion, thus providing another instruction the opportunity to execute under control of the transfer plan executor. In practice, one function is provided which is used by the instruction implementor to activate all asynchronous I/O. The transfer plan executor itself handles all asynchronous I/O completion. Thus, once the executor is implemented, the programmer need not worry about explicitly handling asynchronous I/O.

The atomic primitives of Swift/RAID are somewhat similar to the *large atomic blocks* described by Park and Shaw, although the blocks of code defining a Swift/RAID instruction are determined based solely on I/O [Park and Shaw, 1991]. Loops and other operations of variable duration are permissible within Swift/RAID atomic primitives, although it is assumed that the duration of such operations is reasonably bounded.

The distributed atomic instruction approach described here is an implementation technique. Shaw has described the use of distributed real-time state machines for specification purposes [Shaw, 1992]. Although the implementation technique is very general, the implementation provided by the Swift/RAID system is very specific and does not constitute a general purpose concurrent programming environment.

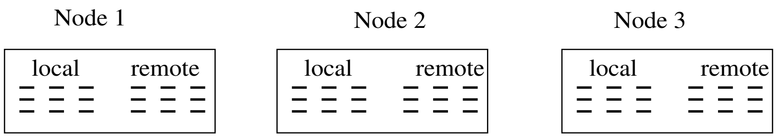
Individual Swift/RAID instructions have the atomicity properties required of distributed transactions

The operation of the Transfer Plan Executor proceeds in five steps:

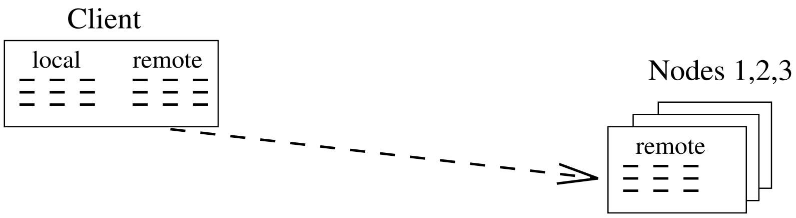
1. The application (the client) calls a Swift/RAID library function:

   swift_write(file, buffer, length);

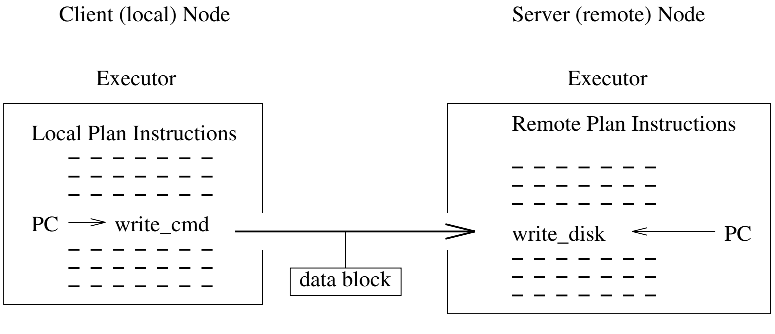
2. The library assembles (at the client) local and remote plans for every node.



3. Local and remote instructions will cooperate in lock-step. All remote plans are transmitted to the remote nodes.



4. Cooperating plans are executed by the executors on the client and each of the servers.



5. All operations and data transmissions are accomplished by cooperating local and remote instructions, as driven by the executors.

Figure 3: The five operation steps of the Transfer Plan Executor.

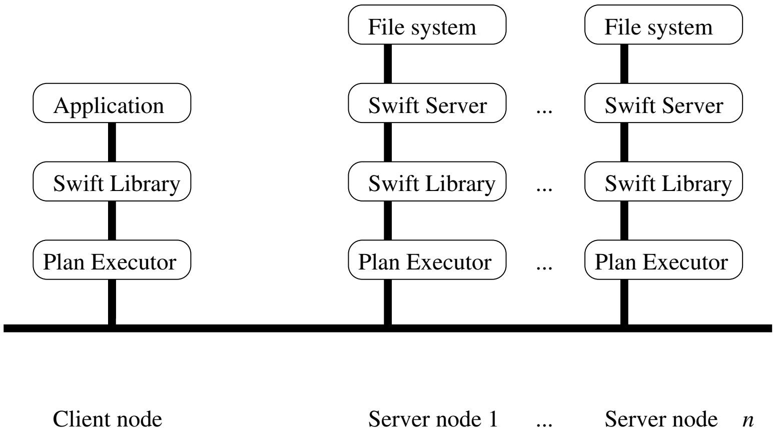


Figure 4: Library implementation Swift/RAID.

[Spector et al., 1985, Cabrera et al., 1993]. *Synchronization* is inherent in the dispatch mechanism of the executor. Since each instruction corresponds to a single I/O, *failure atomicity* is provided in that the operation either fails or succeeds. All instructions can be restarted on failure any number of times. Instructions have *permanence* in that the completion of an instruction which accesses the disk can only occur if the I/O is complete. Plans and plan sets do not have these transaction properties.

3 Swift/RAID Overview

A summary of the Swift/RAID library implementation is shown in Figure 4.

The Swift/RAID library contains an application interface which provides conventional Unix file operations, with the additional requirement that transfers always be aligned on the file block size and transfer lengths are always multiples of this block size. Different Swift/RAID files can have different block sizes. A Swift/Raid application always runs on a client node.

Every Swift/RAID server node is required to run a *swift_server*. A *swift_server* source is linked with the appropriate Swift/RAID library to produce the server supporting a particular RAID variant. Libraries exist for Swift/RAID levels 0, 4, and 5. Application programs also link to the corresponding Swift/RAID library.

When an application calls a Swift/RAID function, the client library generates a corresponding transfer plan set and then calls the local transfer plan executor. The executor transmits the required plans to the servers. The client and server executors then cooperate to execute the plan set. Each of the remote plans is transmitted in a single message to the server on which it will execute.

Servers are driven by a work-loop which calls a load routine to receive the next plan and then calls its local transfer plan executor to execute the plan. This server work-loop is executed once for every Swift/RAID function called by the client application and requiring support from the node on which the server is running. When a Swift file is initially accessed, a copy of the *swift_server* process is forked. On a given node, there will be one *swift_server* process for every open Swift/RAID file.

The relationship between the data structures managed by Swift/RAID is shown in Figure 5. The **file_desc** structure roots all data structures supporting operations on the file, and differs for each RAID implementation. The **plan_hdr** structure contains asynchronous I/O masks, time-out values, and pointers to **node** structures. The **node** structures contain plan contexts, pointers to the arrays of *instructions* which define plans, and the sockets used for network communication. It is the **node** structure that contains program status for the local plan serving that node, including fields such as a program counter and halt/run flags. This is illustrated in Figure 5.

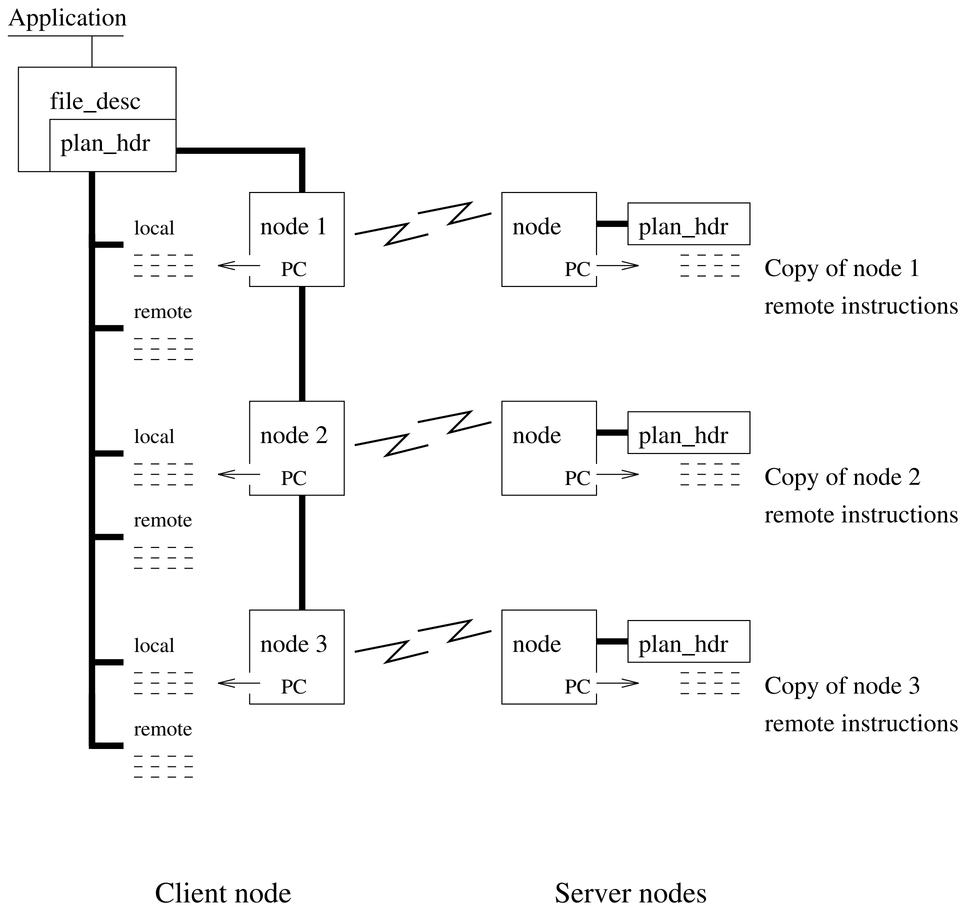


Figure 5: Main data structures used in the implementation of atomic network I/O in Swift/RAID.

The routines **loc_compile** and **rmt_compile** are called by the Swift/RAID library to build a plan set tailored to a given user request. It is this high-level logic which differs among the Swift/RAID variants. The executor executes the resulting instructions by advancing the program counter for each plan through its instructions, dispatching functional code as directed by the op-codes. When an instruction initiates network I/O by either transmitting a data block or awaiting reception of a data block, further execution on behalf of the current plan is blocked and another plan is selected for execution. It is the execution of a distributed pair of instructions, for instance a remote **read disk** and a local **read result**, which actually causes data blocks to be transmitted between the server and the client. Because the complete plan set is computed in advance, unexpected network data I/O *never* occurs, although messages may potentially be lost or repeated.

Processing of a Swift/RAID user request thus consists of two phases: a generation phase in which a plan is assembled, and an execution phase driven by a call to the transfer plan executor. Plans are not discarded until after the entire plan set has completed execution. Experimenting with protocols that do not require immediate acknowledgments is thus possible. Such protocols include sliding window protocols, light-weight timer based protocols, and blast protocols. See Long *et al.* [Long et al., 1993] for a discussion of such protocols with respect to Swift multimedia applications, and Carter and Zwaenepoel [Carter and Zwaenepoel, 1989] for a discussion of blast mode protocols.

Swift/RAID uses low-level communication routines very similar to those used in the original Swift implementation. These routines are based on the UDP **sendmsg** and **rcvmsg** primitives. UDP is the Internet Protocol (IP) based User Datagram Protocol [Cerf, 1993].

4 The Transfer Plan Executor

4.1 Instruction Execution

The *transfer plan executor* is a distributed virtual machine executing on every Swift/RAID network node. The executor virtual machine can concurrently process an arbitrary number of plans. In effect, it is multiprocessed. Although all plan sets for a particular Swift/RAID service are similar, the details of each depend on the specifics of the requested operation.

A simple client-server transfer plan pair is shown in Figure 6 This example shows the cooperating plan pair for the first node in a three node Swift/RAID level 5 write operation. The first instruction in the local plan (**migrate partner**) causes the corresponding remote plan to be transmitted to the server node. Each local **write cmd** instruction then transmits either a section of the user's buffer or a stripe parity buffer to a cooperating remote **write disk** instruction. Each **write disk** instruction writes the data it receives into the correct block of server storage.

The executor executes a work-loop resembling the fetch, decode, and execute cycle of a CPU. This work-loop also performs some simple kernel functions in that it schedules, executes, and blocks all locally executing plans and drives all network communication. Each plan contains a program counter identifying the plan's current instruction. The instruction op-code contains a bit indicating whether an instruction posts an I/O request, in which case the work-loop will block the current plan after executing the instruction.

The executor executes instructions by calling one of two dispatch routines and then advancing the program counter. The dispatch routines execute code corresponding to the instruction's op-code. Figure 7 shows the format of all Swift/RAID instructions. Figure 8 lists Swift/RAID level 5 op-codes. A *pre*-dispatch routine executes instructions that may initiate a network I/O request. A *post*-dispatcher executes instructions that execute after the network I/O completes. Such instructions are called *receive* instructions. Execution of the plan blocks when the first **receive** instruction is encountered.

A high/low protocol stack is used to provide high-level functionality specific to a particular RAID implementation. To support this high/low protocol scheme every instruction contains two op-codes, with

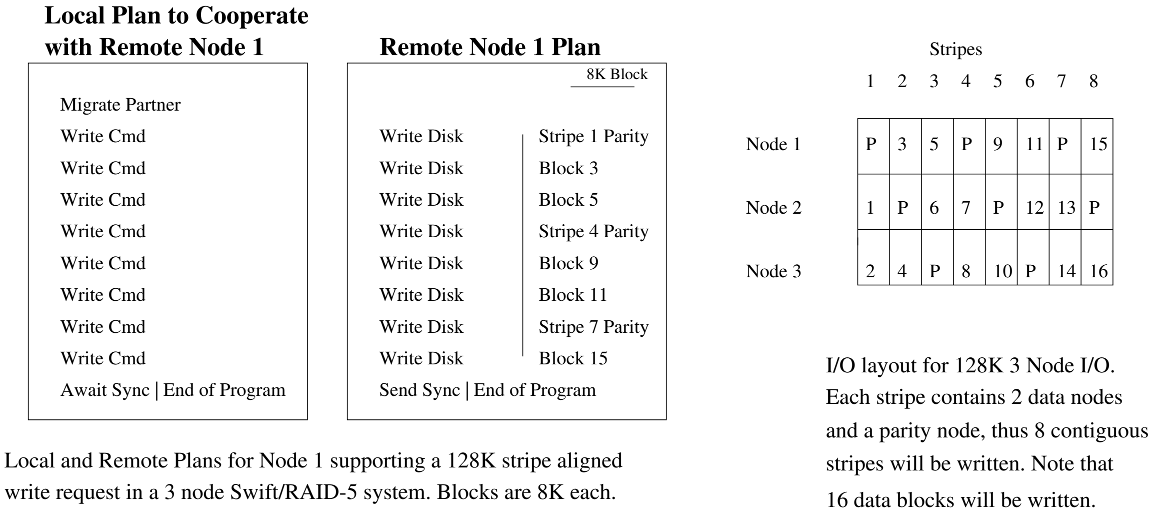


Figure 6: A simple Swift/RAID level 5 blast mode data transfer plan.

Low-level op-code	2 byte Executor op-code.
High-level op-code	2 byte Swift/Raid Library op-code.
Operand 1	4 bytes (Data Buffer Pointer).
Operand 2	4 bytes (File Transfer Location).
Operand 3	4 bytes (Length).
Program location of Self	2 bytes.
Program location of Partner	2 bytes.
Global Event Counter	4 bytes.
Plan ID / Unique ID / Sanity Check	6 bytes.

Figure 7: Instruction fields of all Swift/RAID instructions.

Low-level	Modifiers	RAID-5
Read Disk	End of Program	Write Read
Read Result	Close	XOR Buffer
Write Cmd		XOR Parity
Write Disk		Server Read Parity
Await Sync		Server Send Parity
Send Sync		
Migrate Partner		
Restart		
Delta Wait		
Null		

Figure 8: Swift/RAID operation codes clustered in three categories.

the low-level op-code dispatching low-level transfer plan executor operations and the high-level op-code dispatching high-level RAID operations. A RAID implementation library is required to provide high-level *pre* and *post* dispatchers.

The low-level instructions perform simple I/O operations required of any RAID implementation and perform no parity calculations. The low-level instructions are essentially used to read and write server storage blocks. The **read disk** instruction reads a specified disk block and transmits it to a corresponding **read result** instruction, which typically places the received data in the proper location in a user buffer. The **write cmd** instruction transmits data, often directly from a user buffer, to a cooperating **write disk** instruction which then writes the corresponding disk block. The low-level op-code is occasionally a no-op. The high-level op-codes provide RAID functionality specific to each RAID variant and perform dynamic functions that cannot be precomputed when the instruction is assembled, for instance, setting a buffer address to that of a server allocated parity computation buffer.

4.2 Error Handling

Instructions are generated as distributed client-server pairs with all instructions containing two static program location fields (see Figure 7). The two program locations specify the instruction's program counter within the local plan and the program counter of the partner instruction in the remote plan. These two fields in each instruction support error recovery. When an error occurs during execution of the distributed instruction pair, both sides of the distributed instruction are restarted as a unit. The header of each message received by an executor contains a destination plan identifier and a copy of the remote instruction that requested the message transmission. This remote instruction copy contains the expected program counter of its local partner instruction.

All communication between nodes is supervised by the cooperating transfer plan executors. The executors are responsible for error control, retransmissions, and time-outs. Instruction implementation routines thus never have to deal with acknowledgments, flow-control, or timing issues as the executor work-loop's process of advancing plan program counters includes assuring that required communication has occurred. When a network I/O completes, the executor attempts to advance the relevant plan program counter. The executors do not use any special flow control or acknowledgment messages. If an explicit acknowledgment is required within a plan, a distributed instruction pair must be generated in the plan using the **await sync/send sync** op-codes.

Communication failures result in time-outs, corrupted messages, or lost messages. These events cause resynchronization of the relevant distributed instruction pair via a **restart** instruction, which resets the local and remote program counters of the plans containing the instruction pair. A **restart** is the only instruction that is ever generated on-the-fly within the executor.

All time-outs correspond to a blocked local **receive** instruction. Upon time-out, a **restart** instruction is sent to the cooperating executor. The **restart** instruction contains the program counter of the expected partner instruction. When an executor receives a **restart**, it sets the program counter of the corresponding local plan to the program counter specified within the received **restart** instruction. The **restart** forces a jump to a predefined synchronization location in the remote plan.

If the partner program counter contained within a received instruction is less than the current local program counter, the message is assumed to be a duplicate and is discarded. If the partner program counter within the received instruction is greater than the current program counter, some message are assumed lost, perhaps due to overrun. The received instruction is discarded and a **restart** transmitted to restart the partner to keep it synchronized with the current program counter.

Nothing keeps track of whether a **restart** has been transmitted. If a **restart** is lost, a time-out to the same blocked instruction will recur and another **restart** will be transmitted. After transmitting **restart**, either communication from the expected remote instruction is received, in which case the plans proceed, another unexpected instruction is received, in which case a **restart** is reissued, or a **restart** is received, in which case it is respected and the local program counter reset.

5 The Reimplementation of Swift

The RAID level 5 implementation consists of approximately 4000 lines of **C** code, with the transfer plan executor containing some 1100 lines and the Swift/RAID level 5 library containing 1800 lines. For comparison, the Swift/RAID level 0 library contains some 550 lines. The Swift/RAID level 0 implementation has the same functionality as the original Swift, and was implemented first to develop and debug the transfer plan executor approach.

The versions of Swift/RAID described here used client-server plans generated so that a stop-and-wait protocol was implemented between each cooperating client and server plan. Stop-and-wait is implemented by compiling pairs of instructions which perform explicit synchronization. The simple stop-and-wait protocol provides a base performance level against which to evaluate the more complex protocols with which we have begun to experiment.

Implementation and performance analysis was performed on a dedicated Ethernet network consisting of a SparcStation 2, a SparcStation IPX, a SparcStation IPC, and three SparcStation SLCs. The SparcStation 2 was used as the client.

6 Performance Evaluation of Swift/RAID levels 4 and 5

This section describes our initial performance evaluation of Swift/RAID. The measurements of the prototype are presented in Figures 9 through 14.

The data rates observed for a five node Swift/RAID system and a three node Swift/RAID system were similar. We chose to present the measurements for the five node system as its data rate decreases less under failures.

6.1 Measurement runs

The benchmarks reported in this section are based on programs in which repeated Swift/RAID read or write requests are issued within a loop. These programs were similar to those previously described [Legge and Ali, 1990], except that all I/O transfers always started at the beginning of the Swift file, *i.e.,* a seek to the start of the file was performed before every I/O transfer. The size of the I/O transfer was increased by 8 kilobytes in every iteration of the loop, with the maximum size of a single Swift I/O operation being 800 kilobytes. The Swift files were preallocated, so there were no effects from file extension. All writes were done synchronously.

With single exception noted below regarding the original Swift data, the results reported here were obtained by averaging 50 iterations over each 8 kilobyte transfer size between 8 kilobytes and 800 kilobytes. Performance tests were run on a lightly loaded network. UDP datagram sockets were used, and no overflows occurred during the performance testing reported here.

6.2 Discussion of the measurements

The performance of any RAID system is sensitive to the frequency of write operations, as writes require both parity computations and parity block modification. Optimum write throughput occurs when an entire stripe is written in one I/O, in which case the parity block for the stripe can be written without any additional I/O required other than the parity block write itself. For large multistripe aligned operations, the larger the number of blocks in the stripe, the less will be the relative I/O cost due to writing the parity block. Conversely, the larger the stripe, the greater the relative cost for small writes.

RAID degraded mode operations occur when a node has failed and data needs to be reconstructed from the operational nodes. Degraded mode data rate must also be considered when selecting stripe size. Investigating these performance trade-offs has been an active area of RAID research. Results are sometimes counter-intuitive. Ng and Mattson, for example, note that degraded mode data rate, for some stripe sizes and read-to-write ratios, can actually provide superior data rates when compared to undegraded performance [Ng and Mattson, 1992].

Figures 9 and 10 contrast undegraded performance on a five node network of the original Swift implementation and all three Swift/RAID implementations. Read and write data rates are shown separately. As expected, read data rates are similar for all systems since, in undegradedΦΦΦΦΦundegraded mode, the number of RAID read I/O operations is the same as for non-RAID reads. Read throughput for all systems remains around 600 kilobytes/second.

The write data rate of the original Swift and of Swift/RAID level 0 are comparable. As it was not possible to run the original Swift reliably for 50 iterations at large block sizes, only 10 iterations were used to obtain the original Swift data rate data, while all Swift/RAID data was obtained using 50 iterations. Write throughput for the original Swift and for Swift/RAID level 0 remains between 800 kilobytes/second and 900 kilobytes/second. In Figure 10 Swift/RAID level 4 achieves a write throughput in the order of 300 kilobytes/second and Swift/RAID level 5 in the order of 560 kilobytes/second. This data rate difference illustrates the rationale behind RAID level 5, *i.e.,* Swift/RAID level 4 suffers from all parity blocks being located on the same parity node. We believe our write data rate compares favorably with the 1.15 megabytes/second derived experimentally by [Boggs et al., 1988] for the maximum rate obtainable between a pair of hosts on an Ethernet. Measurements done on a three node system achieved similar throughput.

In Figures 11 and 12 both Swift/RAID level 4 and Swift/RAID level 5 degraded mode performance is shown. In both cases a five node network was operating in degraded node, *i.e.,* with only four nodes. Undegraded data rate is shown for reference. In the case of Swift/RAID level 4, throughput is shown both for the case where the failed node is the parity node and for the case where a non-parity node has failed.

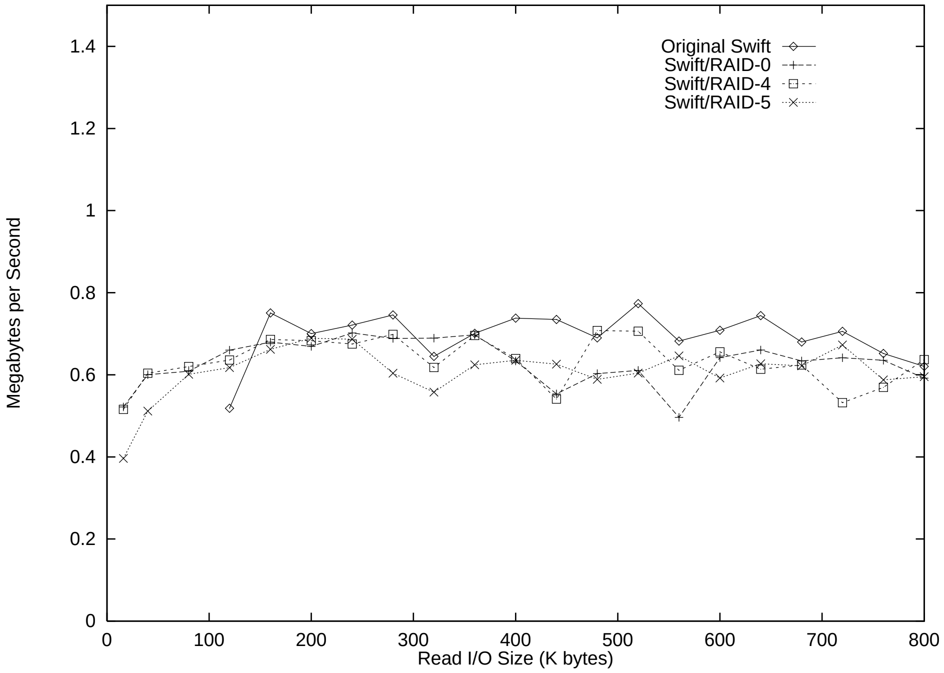


Figure 9: Read performance of Swift/RAID, 5 nodes.

Note that the degraded mode Swift/RAID level 5 performance is significantly better than that of Swift/RAID level 4. As expected, in Figure 11, undegraded Swift/RAID level 4 reads and degraded reads with the parity node failed perform similarly. Swift/RAID level 4 writes when any node has failed all perform similarly, as every node in the stripe other than the failed node must either be read or written. Note that reads with a failed data disk perform essentially the same as writes, since all nodes in the stripe must be read to compute the missing blocks. The Swift/RAID level 5 data in Figure 12 shows that, in the 5 node case, performance in degraded mode differs little from non-degraded mode performance. Degraded writes actually perform slightly better than undegraded writes, reinforcing results in [Ng and Mattson, 1992].

Figure 13 investigates the effect on performance of the number of nodes in a Swift/RAID network. Swift/RAID level 5 write performance is shown for three node, four node, and five node networks, with the performance of the original Swift and Swift/RAID level 0 shown for contrast. For three, four, and five node networks, Swift/RAID level 5 write performance can be seen to increase by about 50 kilobytes/second with each additional node in the stripe.

Figure 14 shows the performance cost of the parity calculations required to support RAID operations. The cost of parity computations was investigated by building versions of Swift/RAID levels 4 and 5 which did no parity calculations, but did perform all required parity block I/O. Even in the final Swift/RAID version, the parity cost can be seen to be substantial.

6.2.1 Reflections on Network Performance

Our experience reinforces several rules of network performance: avoid time-outs and retransmissions, congestion avoidance is better than congestion recovery, and CPU speed is essential for network performance [Mogul, 1993].

Avoid time-outs and retransmissions. A time-out occurs when network bandwidth has been significantly wasted. In general, our experience was that as soon as time-outs occurred throughput dropped substantially,

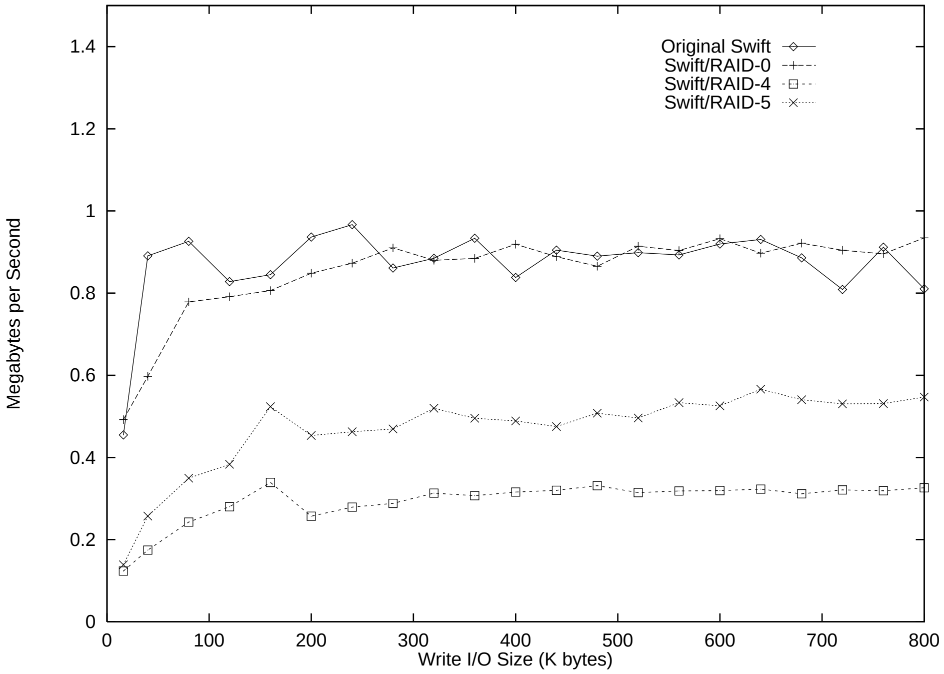


Figure 10: Write data rates of Swift/RAID, five nodes.

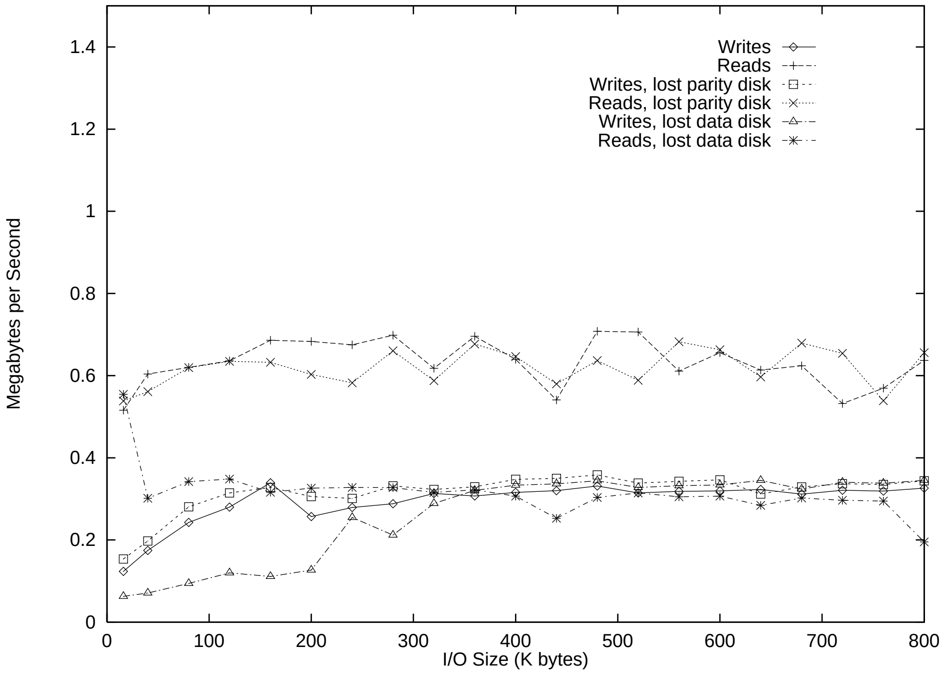


Figure 11: Swift/RAID level 4 data rates in degraded mode, five nodes, one failed.

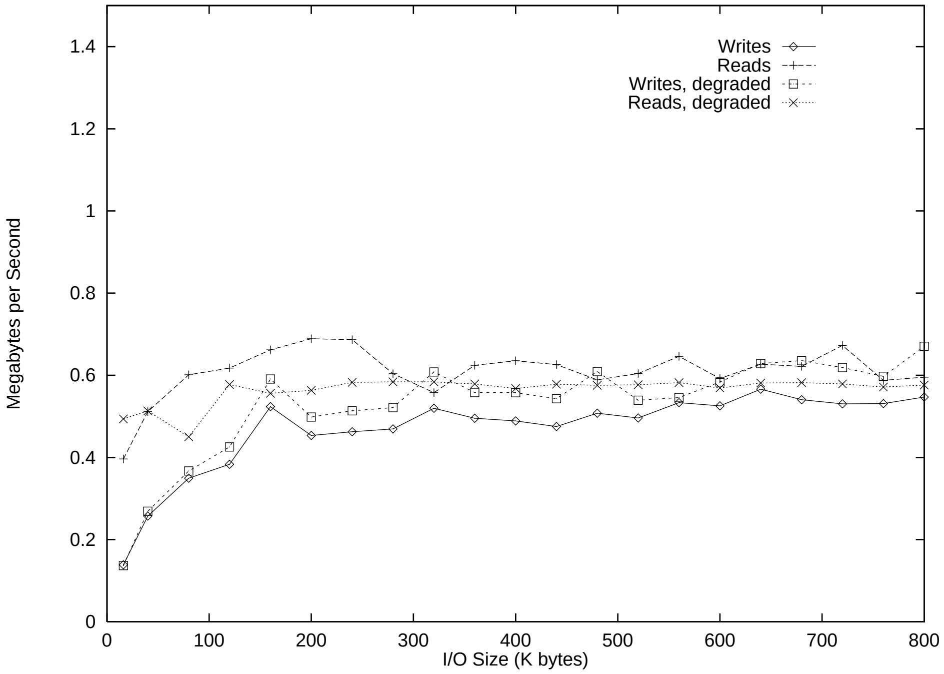


Figure 12: Swift/RAID level 5 data rates in degraded mode, five nodes, one failed.

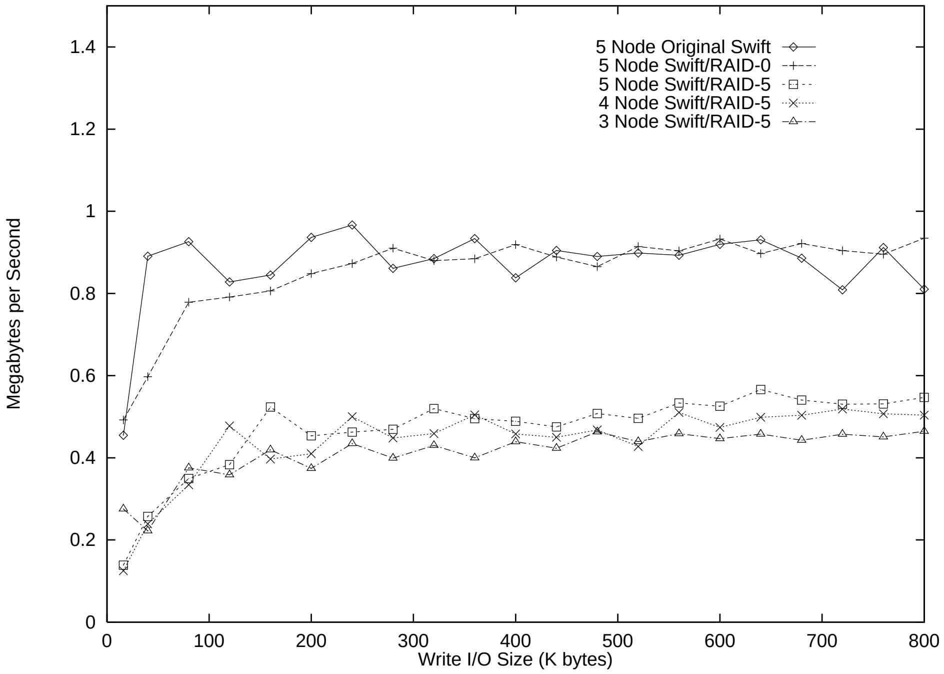


Figure 13: Swift/RAID level 5 write data rates by number of nodes.

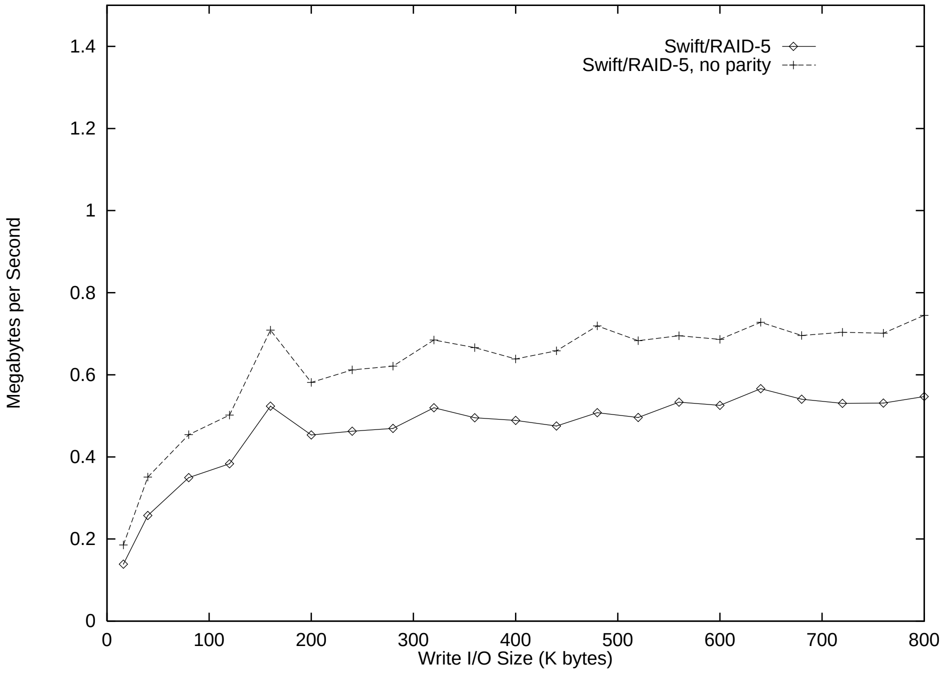


Figure 14: The cost of parity computation for five node Swift/RAID systems.

to less than 100 to 200 kilobytes/second.

Congestion avoidance is better than congestion recovery. The use of stop-and-wait protocols provided congestion avoidance. Initial experimentation with simple blast protocols, where multiple instructions were executed between synchronization, resulted in lower performance than for the simple stop-and-wait protocol once congestion recovery was required, specifically, once UDP socket overflow errors were detected.

CPU speed is essential for network performance. Our experience with the cost of parity calculations confirms that the computational cost of processing packet data may be as significant as the protocol overhead. Naively implemented exclusive-or loops at first resulted in system performance only about 20% of that finally obtained. As network speed increases, the memory and CPU costs of processing data bytes will likely become the bottleneck. Banks and Prudence [Banks and Prudence, 1993] note that memory bandwidth may already be the primary communications bottleneck on networked workstations, and advocate the adoption of *single-copy protocol stacks* to address this problem.

Swift/RAID performs operations directly out of the user's buffers. If a communications protocol other than UDP/IP was used that supported direct network access, Swift/RAID would provide a single-copy protocol. An important advantage of the Swift/RAID approach in providing maximum performance single-copy operations is that the executor extends naturally to providing direct support of device driver code. Instructions can be defined which factor device driver functionality so as to provide the primitive instruction set required to drive the device. A means of having the device interrupt routine activate the executor must also be implemented. Device driver instructions and application instructions can then be mixed within plans as required. Since no domain context switches, data copy operations, or memory management need occur, near maximum performance operations are possible. The *packet filter* [Mogul et al., 1986], where a kernel resident interpreter for a very simple stack-based language is used to route packets to the appropriate user application, was also motivated by minimizing domain crossing operations.

7 Related Work

Variants of the finite state machine approach similar to that described here have been used in the past. In addition to use within various I/O controllers and coprocessors, this approach has often been used in real-time avionics [Glass, 1983]. Discussions of this approach can be found in literature [MacLaren, 1980, Shaw, 1986, Baker and Scallon, 1986]. The use of static precompiled tables is discussed by MacLaren, who refers to this approach as a traditional cyclic executive. Our approach differs from that of MacLaren in that we generate our plans dynamically and our finite state machine is event driven, rather than cycle time driven. MacLaren notes that generating scheduling tables by hand can become difficult, and also notes that "the efficiency of a cyclic executive derives from its minimal scheduling property, and from the very small implementation cost." Our scheduling tables are generated by the logic of the Swift/RAID library. Shaw [Shaw, 1986] describes real-time software as either being based on concurrent interacting processes or as using the table-driven finite state machine approach, which he terms *slice-based*. Shaw notes that the schedulable atomic units we have called *instructions* have often gone under such names as *slices*, *chunks*, or *stripes*. He contrasts the two approaches to real-time software design, and calls for research in including time as a first class programming object.

Variants of this approach have also long been used to implement logically multi-threading servers and drivers in environments where true threads have not been available. Our experience with this approach confirms the observations of other researchers [Allworth, 1981, Beizer, 1983], *i.e.,* this approach can work well but has the flavor of low-level programming, and thus may entail commensurate development and debugging difficulties. An overview of a real-time system implemented using this approach is provided by Baker and Scallon [Baker and Scallon, 1986]. They define their view of the "executive as an independently programmable machine that executes application procedures written in conventional programming languages as if they were individual instructions of a higher level program." According to them the resulting system provides a virtual machine for the system specification, rather than simply a virtual machine for the application program.

8 Conclusions

RAID level 4 and 5 functionality has been added to Swift. We have demonstrated that RAID systems can be implemented in software in a client-server environment. In the network investigated, both Swift/RAID levels 4 and 5 currently obtain peak read rates near 90% of the original Swift. Swift/RAID level 5 obtains a write rate slightly above 50% of the original Swift write rate, while Swift/RAID level 4 obtains only around 35% of the original write rate. The observed five node Swift/RAID level 5 data rates, for transfers larger than 100 kilobytes, ranged from 560 kilobytes/second to 680 kilobytes/second for reads, and from 380 kilobytes/second to 560 kilobytes/second for writes. These indicate that future work with distributed RAID systems is feasible.

Our reimplementation uses a methodology based on atomic and durable sets of data transmission operations. These operations are generated by a run-time library and are interpreted by a distributed finite state machine, the transfer plan executor. This methodology dominates all aspects of the implementation and enabled us to experiment with software RAID implementations on typical Unix research networks.

The distributed table-driven finite state machine approach solves the distributed concurrent programming problem to the degree necessary to implement the different RAID systems. Designing the sequences of required atomic operations and implementing the programs to generate these sequences is difficult. While this process has the flavor of low-level programming and has the associated difficulties encountered when debugging and testing, it has the virtue of working and of raising the level of abstraction by providing atomic I/O operations. For these same reasons, variants of our approach have been used in the past to implement both real-time applications and logically multi-threading servers and drivers.

Our experience confirms the unsettling fact that in producing a high-performance implementation one must selectively avoid the principles of information hiding and transparent layering. As [Padlipsky, 1985] has stated, "Layering makes a good servant but a bad master." We found that to improve the performance of Swift/RAID levels 4 and 5 it was essential for us to investigate what the compiler generated within the parity computation loop and for us to design the Swift/RAID library with transfer plan executor internals in mind.

The Swift/RAID system implements the core functions required for any RAID system. Areas for future enhancement include investigating adaptive burst mode protocols, integrating automatic node rebuild operations, locking services, and enhancing directory service.

References

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

[Baker and Scallon, 1986] Baker, T. P. and Scallon, G. M. (1986). An architecture for real-time software systems. *IEEE Software*, pages 50–58.

[Banks and Prudence, 1993] Banks, D. and Prudence, M. (1993). A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202.

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

[Boggs et al., 1988] Boggs, D. R., Mogul, J. C., and Kent, C. A. (1988). Measured capacity of an Ethernet: Myths and reality. In *Proceedings of the SIGCOMM '88 Symposium on Communications Architectures and Protocols*, Stanford, CA, USA. ACM SIGCOMM.

[Cabrera and Long, 1991] Cabrera, L.-F. and Long, D. D. E. (1991). Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436.

[Cabrera et al., 1993] Cabrera, L.-F., McPherson, J., Schwarz, P. M., and Wyllie, J. C. (1993). A comparison of two log-based implementations of atomicity. *IEEE Transactions on Software Engineering*, 19(10).

[Carter and Zwaenepoel, 1989] Carter, J. B. and Zwaenepoel, W. (1989). Optimistic implementations of bulk data transfer protocols. In *Proceedings of the 1989 ACM SIGMETRICS and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems*, Berkeley, CA, USA.

[Cerf, 1993] Cerf, V. G. (1993). Core protocols. In Lynch, D. C. and Rose, M. T., editors, *Internet System Handbook*, pages 79–155. Addison-Wesley.

[Emigh, 1992] Emigh, A. T. (1992). The Swift architecture: Anatomy of a prototype. Technical report, UCSC Concurrent Systems Laboratory.

[Glass, 1983] Glass, R. L. (1983). *Real-Time Software*. Prentice-Hall.

[Katz et al., 1989] Katz, R. H., Gibson, G., and Patterson, D. A. (1989). Disk system architectures for high performance computing. *Proceedings of the IEEE*, 77(12):1842–1858.

[Kim, 1986] Kim, M. Y. (1986). Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988.

[Legge and Ali, 1990] Legge, G. and Ali, M. (1990). Unix file system behavior and machine architecture dependency. *Software – Practice and Experience*, 20(11):1077–1096.

[Long et al., 1993] Long, D. D., Osterbrock, C., and Cabrera, L.-F. (1993). Providing performance guarantees in an FDDI network. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 328–336, Pittsburgh.

[MacLaren, 1980] MacLaren, L. (1980). Evolving toward Ada in real-time systems. In *Proceedings of the ACM SIGPLAN Symposium on the Ada Language*, Boston.

[Mogul, 1993] Mogul, J. C. (1993). IP network performance. In Lynch, D. C. and Rose, M. T., editors, *Internet System Handbook*, pages 575–675. Addison-Wesley.

[Mogul et al., 1986] Mogul, J. C., Rashid, R. F., and Accetta, M. J. (1986). The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 39–51, Austin, TX, USA.

[Ng and Mattson, 1992] Ng, S. W. and Mattson, R. L. (1992). Maintaining good performance in disk arrays during failure via uniform parity group distribution. In *Proceedings of the First International Symposium on High-Performance Distributed Computing*, pages 260–269, Syracuse, New York. IEEE Computer Society Press.

[Padlipsky, 1985] Padlipsky, M. A. (1985). *The Elements of Networking Style and other essays and animadversions on the art of intercomputer networking*. Prentice-Hall.

[Park and Shaw, 1991] Park, C. Y. and Shaw, A. C. (1991). Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, pages 805–816.

[Patel, 1982] Patel, A. M. (1982). Error and failure-control procedure for a large-size bubble memory. *IEEE Transactions on Magnetics*, 18(6):1319–1321.

[Salem and Garcia-Molina, 1986] Salem, K. and Garcia-Molina, H. (1986). Disk striping. In *Proceeding of the Second International Conference on Data Engineering*, pages 336–342.

[Shaw, 1986] Shaw, A. C. (1986). Software clocks, concurrent programming, and slice-based scheduling. In *Proceedings of the 1986 Real-Time Systems Symposium*, pages 14–18.

[Shaw, 1992] Shaw, A. C. (1992). Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9):805–816.

[Spector et al., 1985] Spector, A. Z., Daniels, D., Duchamp, D., Eppinger, J. L., and Pausch, R. (1985). Distributed transactions for reliable systems. In *Proceedings of the Tenth Symposium on Operating System Principles*, volume 19, pages 127–146, Orcas Island, Washington, USA.