RAIDframe: Rapid prototyping for disk arrays

Garth Gibson, *William V. Courtright II, *Mark Holland, Jim Zelenka

18 October 1995
CMU-CS-95-200

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

*Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Submitted to the 1996 ACM conference on measurement and modeling (SIGMETRICS-96)

Abstract

The complexity of advanced disk array architectures makes accurate representation necessary, arduous, and error-prone. In this paper, we present RAIDframe, an array framework that separates architectural policy from execution mechanism. RAIDframe facilitations rapid prototyping of new RAID architectures by localizing modifications and providing libraries of existing architectures to extend. In addition, RAIDframe implemented architectures run the same code as a synthetic and trace-driven simulator, as a user-level application managing raw disks, and as a Digital Unix device-driver capable of mounting a filesystem. Evaluation shows that RAIDframe performance is equivalent to less complex array implementations and thance is equivalent to less complex array implementations and that case studies of RAID levels 0, 1, 4, 5, 6, and parity declustering achieve expected performance.

http://www.cs.cmu.edu:8001/Web/Groups/PDL

The project team is indebted to the generous donations of the member companies of the Parallel Data Laboratory (PDL) Consortium. At the time of this writing, these include: Data General, Digital Equipment, Hewlett-Packard, International Business Machines, Seagate, Storage Technology, and Symbios Logic. Symbios Logic additionally provided a fellowship. The Data Storage Systems Center also provided funding through a grant from the National Science Foundation under grant number ECD-8907068. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the PDL Consortium member companies or the U.S. government.

Keywords: RAID, disk arrays, simulation, modeling, software engineering, directed acyclic graphs

1. Introduction

Disk arrays are an effective method for increasing I/O system performance [Salem86, Kim86]. By incorporating redundancy into arrays, systems are able to survive disk faults without loss of data or interruption of service [Lawlor81, Patterson88, Gibson93]. Popularized by the RAID taxonomy and driven by a broad spectrum of application demands for performance, reliability, availability, capacity, and cost, a significant number of redundant disk architectures have been proposed. These include designs for emphasizing improved write performance [Menon92a, Mogi94, Polyzois93, Solworth91, Stodolsky94], array controller design and organization [Cao93, Drapeau94, Menon93], multiple failure toleration [ATC90, Blaum94, STC94], performance in the presence of failure [Holland92, Muntz90], and network-based RAID [Cabrera91, Hartman93, Long94]. Finally, the importance of redundant disk arrays is evidenced by their pronounced growth in revenue, projected to exceed $5 billion this year and to surpass $13 billion in 1998 [DISK/TREND94].

In this paper, we are concerned with the process of developing and evaluating a new array architecture. In general, ideas for new architectures are derived using back-of-the-envelope analytical models and evaluated by detailed simulation experiments. To maximize evaluation accuracy, simulators are sometimes built with sufficient detail to manipulate actual data—often by beginning with code from a running storage system [Chen90b, Lee91, Kistler92]. In our experience, the specification and implementation of a detailed array simulator, even if it does not closely model a running system, is an arduous task. To reduce the development cost and complexity of such a simulator, a designer may too often limit the simulator's design and implementation to one or a small number of array architectures. This can greatly complicate the task of modifying the simulator to model another, significantly different, array architecture.

As an example of the complexity involved in modifying a simulator, we examined simulator code changes we made in the development of our first new array architecture [Holland92]. We began with RaidSim, a 92 file, 13,886 line detailed array simulator derived at Berkeley from an implementation of a functional device driver in the Sprite operating system [Chen90b, Lee91]. To this simulator we added new array functions, replaced previously unessential simple mechanisms, and augmented statistics recording, workload generation and debugging functions. During these changes one file was deleted, 11 files with 1,204 lines of code were added and 46 files were modified, changing 1,167 lines and adding 2,742 lines. Collectively, to do the research reported in one paper, the number of simulator lines added or changed, 5,113, was equivalent to 36% of the size of the original simulator and affected half of RaidSim's modules. With this development cost in mind, we set out in 1994 to construct an array evaluation system that reduced the cost and complexity of experimenting with new array architectures.

This paper introduces RAIDframe, a framework for rapidly prototyping redundant disk arrays. Based on a model of RAID operations as directed acyclic graphs (DAGs) and a simple state machine (engine) capable of executing operations represented as DAGs, RAIDframe is designed to allow new array architectures to be implemented with small, localized changes to existing code. Central to RAIDframe is the programmatic abstraction of an architecture as a set of directed acyclic graphs, one for each distinct class of user requests, whose nodes are primitive operations such as disk read or buffer exclusive-or (XOR). Architecture features that cannot be expressed in request DAGs are usually satisfied by RAIDframe's flexible, dynamic address mapping mechanism, its extensible disk queueing policies or its policy configurable cache, although some changes, such as real-time disk optimizations, may require the definition of new DAG primitives.

RAIDframe allows a single implementation to be evaluated in three distinct environments. First, it provides a discrete event simulator with configurable array parameters and disk models. To exercise this simulator, RAIDframe offers a synthetic and trace-driven load generator. Second, RAIDframe implementations and its load generator can be executed in a user process using the UNIX "raw" device interface to access real disks instead of simulated disk models. Finally, to allow real applications to be run against a file system mounted on a RAIDframe-described array architecture, the same implementation code that runs in simulation or as a user process can be run as a device driver in the Digital UNIX operating system on Alpha workstations.

This paper describes the motivation and structure of RAIDframe, describes six case studies of array architectures and reports RAIDframe's storage performance efficiency and its evaluation of the performance of each case study.

2. RAIDframe: Rapid Prototyping for Arrays

2.1 Directed Acyclic Graphs for Specifying Access Sequencing

To facilitate rapid development of and experimentation with advanced array architectures, we have developed a software architecture called RAIDframe. Because advanced array architectures broadly exploit four techniques—a carefully tuned sequencing of disk accesses, a layer of mapping indirection, architecture-aware caching, and device-specific optimizations—RAIDframe separates the specification of these aspects of an architecture from the mechanisms needed to execute any array design.

Central to the design of RAIDframe is the notion of providing an array architect with a highly focussed and flexible abstraction for specifying the sequencing of the primitive operations (e.g. disk reads and writes) which collectively form RAID operations. Since some RAID architectures differ only in mapping, our access sequencing methods are isolated from mapping interpretation. Our experience with RAID controllers designed as operating system device drivers is that mapping and sequencing logic is the likeliest part of an architecture to be modified during experimentation, but is often interspersed throughout the code, making modification time-consuming and error-prone.

The development of RAIDframe's access sequencing abstraction owes much to prior storage systems. A powerful approach for localizing access-specific control is suggested by IBM's channel command words and channel programs [Brown72]. Although this model serializes accesses, its success indicates that the range of primitives needed to support storage control is small and may directly correspond to operations provided by lower level devices. Similar abstractions are found in SCSI chip control scripts [NCR91] and network-RAID node-to-node data transfers [Cabrera91, Long94]. A more flexible example of a storage control abstraction specifically developed for RAID architectures is the parallel state table approach in TickerTAIP, a distributed implementation of RAID level 5 [Cao93]. RAIDframe expands on the example of TickerTAIP, simplify the expression of potentially concurrent orderings of operations, by using directed acyclic graphs (DAGs) of primitive actions. We believe that a graphical abstraction has the added benefit of compactly and visually conveying the essential ordering aspects of a RAID architecture.

Using DAGs to model RAID operations has a number of additional advantages. First, DAGs can be compactly represented in isolation from the engine that interprets and executes them. This localizes and delineates the code modified to experiment with RAID architectures. Second, since DAGs represent RAID architectures as well-defined connections of primitive operations, it should be possible to apply powerful reasoning techniques to the correctness of an implementation. Third, the structural simplicity of DAGs provides RAIDframe with the opportunity to automate the handling of error conditions; that is, respond correctly to errors without intimate knowledge of the RAID architecture implemented by the system.

RAIDframe's approach, specializing the system structure to localize portions most likely to change during array experimentation, is complimentary with approaches emphasizing rapid prototyping implementation languages. For example, implementing RAIDframe itself in Tcl scripts, analogous to the implementation of HP's AutoRAID simulator, should further enhance rapid prototyping [Wilkes95].

The remainder of this section is devoted to the use and structure of RAIDframe. Case studies of array architectures implemented in RAIDframe are presented in the following section.

2.2 Multiple Evaluation Environments

Array architectures implemented in RAIDframe can be evaluated in three distinct execution environments: an event-driven simulator, a stand-alone application controlling UNIX "raw" disks, and a Digital UNIX device driver capable of mounting a standard file system on a set of disks. In all three environments, the code unique to a disk array architecture (mapping, caching, DAGs, primitive operations, and disk queueing) is reused without change. This multiplicity of evaluation alternatives provides considerable value to an array developer. Simulation allows modelling of unavailable hardware and rapid evaluation of traces. Because the same code, loaded in-kernel, can support a file system in a production operating system, a designer can be confident that simulation is not based on unrealistic abstractions. The in-kernel implementation additionally allows direct benchmarking of real applications (up to the CPU limitations imposed by RAIDframe's software computation of parity). While the out-of-kernel RAIDframe execution environment is a performance compromise in the spirit of microkernel operating system design [Accetta86], experiments run at user-level on attached but unmounted disks execute with little or no dependence on the local operating system. User level execution also offers accurate modeling, by direct execution, of the interactions between application threads and system scheduling [Ganger93]. Relative to an in-kernel implementation, however, user-level RAIDframe has decreased responsiveness to asynchronous events like disk completion and added messaging, system call and context switching overheads. Although it is possible to serve files to real applications by co-locating user-level RAIDframe in an user-level filesystem server, our current implementation provides only the synthetic and trace-driven load generation tools available in simulation.

2.3 RAIDframe Internal Structure

RAIDframe provides extensibility through separation of architectural policy from execution mechanism. *Policy*, such as redundancy update sequences and data layout, which varies with RAID architecture, has been isolated from *infrastructure*, code which does not change with RAID architecture. The primary infrastructure module is the DAG execution engine. This engine is responsible only for fully exploiting the allowable concurrency within a DAG; that is, the engine has no knowledge of the architecture embodied in the DAG. Figure 1 illustrates the structure of RAIDframe.

RAIDframe's engine also incorporates a simple and uniform mechanism for handling error conditions in the array. When any condition occurs that prevents a node in a DAG from completing successfully, the engine suspends execution of uninvoked nodes in the indicated DAG, waits for all in-flight nodes associated with that DAG to complete, destroys the failed DAG, and retries the request beginning from DAG selection. Since DAG selection is based on array state as well as requested operation, a different DAG will be selected on retry. While error handling in RAIDframe is not currently independent of architecture, we are extending this error handling model—quiesce, change state, and retry—into a mechanized error recovery system, capable of providing recovery for n-fault-tolerant arrays [Courtright94].

The policies implementing a particular array architecture are available to RAIDframe's infrastructure as a set of modules are:

- **mapping library**: This library contains the layout routines which determine the placement of data and redundancy information in the array. The routines convert logical (user) addresses into physical (disk) addresses and identify associated stripe boundaries and redundancy units. The routines are typically short (5 lines of C code).

- **graph selection library**: A "graph selection" algorithm is required for each architecture. This algorithm, implemented as a C routine, determines which graph from the graph library is to be used to execute a specific user request (type, layout map), given the current state of the array.

- **graph library**: This library contains the routines, such as `CreateSmallWriteDAG()`, which are capable of creating graphs if called by graph selection. Each routine receives type and physical mapping information and returns a pointer to a graph which is tailored for that request. Adding new graphs requires installing new or extending existing creation functions.

- **primitives library**: This library contains the functions which abstract single device operations (e.g. `XOR`, $\mathtt{DISK}_{\mathtt{RD}}$, etc.) from which graphs are created. Primitives delineate the failure domains that RAIDframe accommodates; that is, when an operation fails, the device associated with it is considered failed as well. Primitives are required to independently detect and recover from soft errors.

- **disk queue library**: RAIDframe allows the disk queue management routines (written in C) to be extended. It assumes only a general queueing interface (enqueue, dequeue).

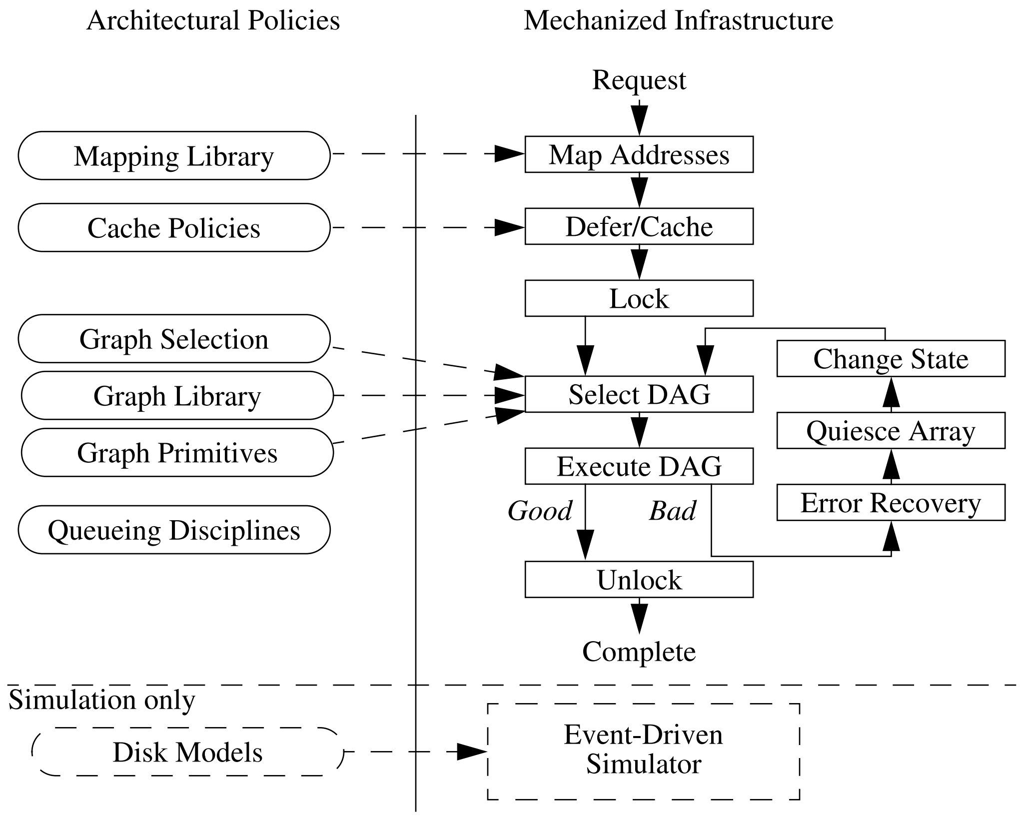

Figure 1: Typical RAIDframe control flow

In this example, when a request arrives in the system, it is first sent to the mapping module to compute the set of physical disk locations affected by the access. This produces a data structure describing, for each stripe touched by the access, the mapping of addresses in the RAID address space to physical disk units within each stripe. This request may be satisfied or deferred in the cache. When a request is forwarded from the cache for physical disk access, its parity stripes are locked to assure that concurrent writes to the same stripe do not conflict in their parity updates. The access is then converted to a DAG and submitted for execution. If a failure occurs during the execution of a DAG, recovery local to the failed primitive leads to quiescing of the DAG engine and modification of the global array state. When terminated requests are reissued, DAGs appropriate for avoiding the failure will be invoked.

- **cache policies**: These are the replacement and writeback policies used by the cache. These policies are structured around a set of boolean triggers and callback routines. On callback, the trigger delivers the list of blocks that caused it to fire. Blocks may be cached according to physical or logical addressing.

- **disk geometry library**: This library contains disk specifications used by the simulator. These specifications include layout parameters (tracks per cylinder, number of zones, etc.) as well as performance parameters (rpm, seek times, etc.).

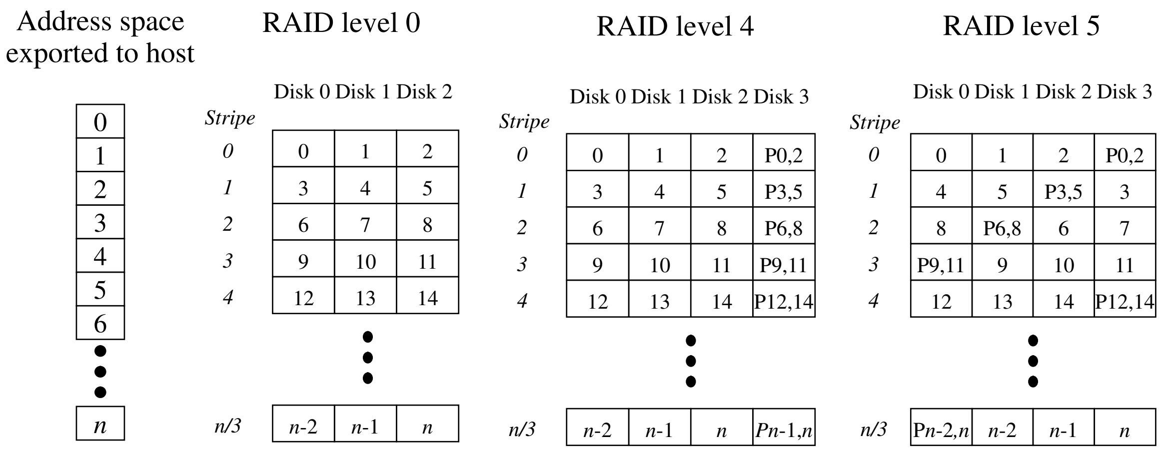

Figure 2: Mapping logical to physical addresses

Logical addresses 0 to n map to physical addresses and parity locations in RAID levels 0, 4, and 5. Pi,j denotes the parity computed over data units i through j.

3. Case Studies in Extensibility

In this section we present six array architectures implemented in RAIDframe. We begin with three of the well-known RAID levels [Patterson88]: RAID levels 0, 4, and 5. Next, we present RAID level 1, the most studied redundant array implementation, also known as disk mirroring [Bitton88, Gray90]. Next, we present declustered parity, a variant of RAID level 5 that reduces the on-line failure recovery performance penalties and RAID level 6 [ATC90, STC94], a double failure tolerant variant of RAID level 5. A full description of these architectures is beyond the scope of this paper; our purpose here is to outline enough architectural detail to facilitate description of the RAIDframe implementation and evaluation.

3.1 RAID Levels 0, 4, and 5

Figure 2 illustrates the mapping of data in RAID level 0, 4, and 5 disk arrays [Lee91]. A RAID controller, whether implemented as a stand-alone subsystem or as a device driver in the host, exports the abstraction of a linear address space; the array appears to the system as one large disk that supports multiple concurrent accesses. The controller maps blocks in this RAID address space to physical disk locations to service read or write requests. In RAID level 0, data units are block interleaved (striped) over the array's disks without redundancy. In RAID level 4, an additional disk contains the bitwise-XOR (parity) of data units at the same offset into each disk. This organization is sufficient to protect the array's data from any single disk failure: any requested unit of data can be recovered by reading the surviving units in the stripe and XORing them together. RAID level 5 provides the same fault-tolerance but rotates parity units over all disks to distribute the parity update workload. While many data and parity layouts are possible [Lee91], the layout shown, called left symmetric, allocates data units to disks round-robin over all disks, interposing parity units between data units according to the parity rotation.

User requests to a fault-free RAID level 0 are translated directly into a DAG of one access on each disk. In RAID levels 4 and 5, however, DAGs must maintain the integrity of redundancy information as well as reconstruct data when single disk faults are present. The five distinct DAG templates used in RAIDframe to implement RAID levels 0, 4 and 5 are diagrammed in Figure 2.

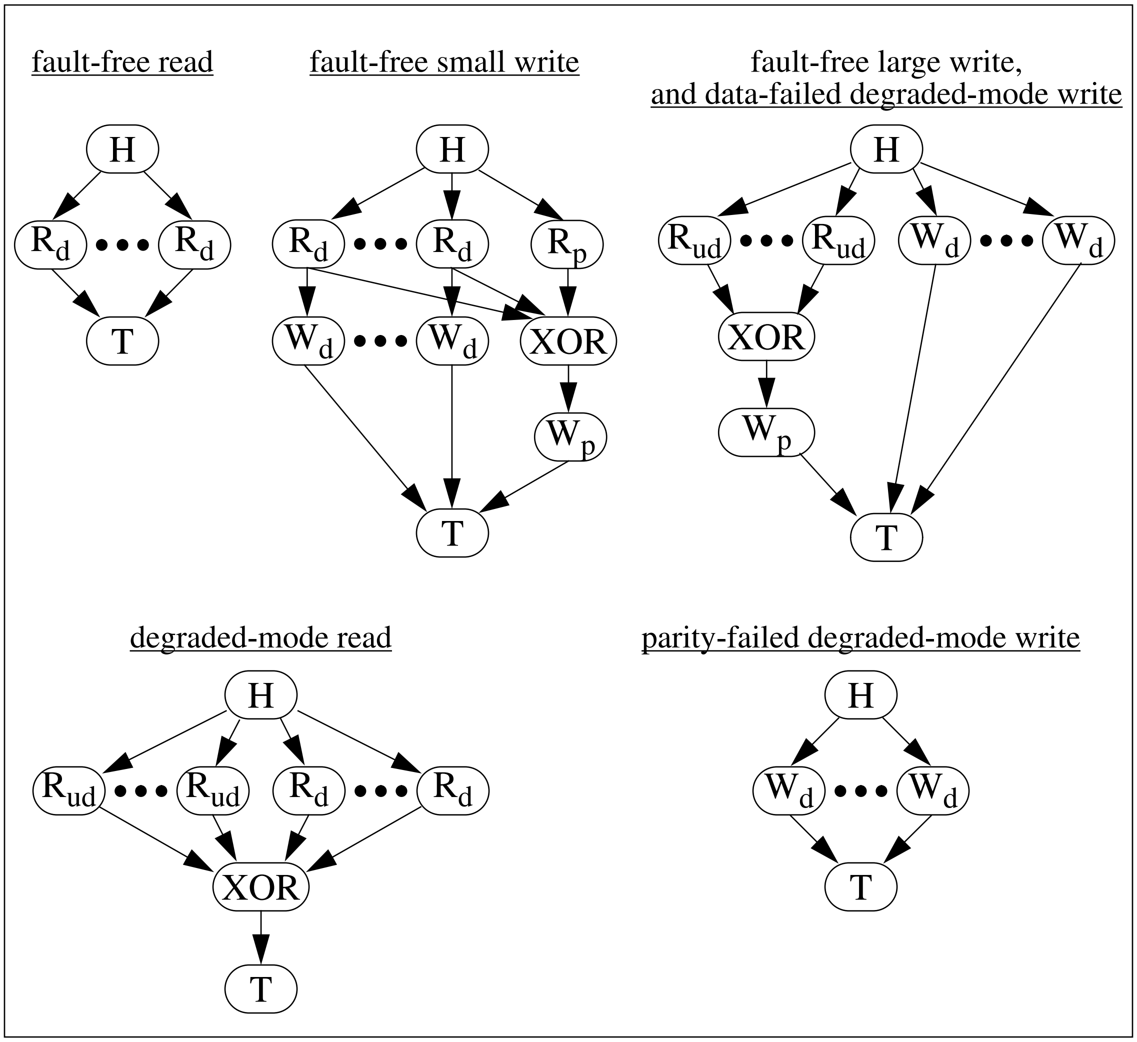

Figure 3: DAG templates used to implement RAID levels 0, 4 and 5.

Nodes labelled "H" and "T" are DAG header and terminator nodes. "R" and "W" nodes invoke disk reads and writes. "XOR" nodes implement an XOR computation over a set of input buffers. "R" nodes are able to lock a disk arm—this provides a performance advantage in operations which use the "fault-free small write" graph in which a write immediately follows the read of a disk block. RAID level 0 uses the "fault-free read" and "parity-failed degraded-mode write" DAG templates while RAID levels 4 and 5 use all five templates. Subscripts are: "d" for addressed data, "ud" for unaddressed data, and "p" for parity.

Edges in a DAGs indicate direct (data or control) dependencies; no node may fire until all of its antecedents have completed. The nodes labelled H and T are, respectively, the header and terminator nodes which obtain data to be written or deliver data read, respectively. Nodes labelled R and W are disk reads and writes, and are annotated (for the reader's convenience only) with a subscript indicating what logical object (parity, data) is being read or written. Nodes labelled XOR compute the bitwise XOR over the data buffers indicated by their input arcs.

The "fault-free read" template has one read node per stripe unit accessed, and simply reads the data to into the buffer supplied in the request. A RAID level 0 write uses the "parity-failed degraded-node write" DAG.

The “large-write” template is used in fault-free RAID level 4 and 5 when more than half of the stripe under consideration is being written. It computes new parity by reading the portions of the stripe that are not being overwritten by the user and XORing this data with the new data supplied by the user.

The “small-write” template is used in a fault-free RAID level 4 and 5 when less than half of the stripe is being written. Each affected data unit is read and then overwritten with new data, without allowing any queued disk requests to be serviced between the read and the write. That is, in parallel, the old parity unit is read, the new parity is computed, and this new parity is written atomically with respect to the read.

The degraded-mode templates are used in RAID levels 4 and 5 when a portion of the stripe being updated contains a failed unit (data or parity). In the read case, the missing data unit is reconstructed by reading all surviving units in the parity stripe and XORing them together. When the operation is a write, a distinction is made between the case where it is the data that has failed and the case where it is the parity that has failed. In the former case, the old data cannot be read, and so writes must read the unmodified data to reconstruct the missing information. This is equivalent to using the fault-free large-write technique previously described, except that the write to the failed unit is suppressed. In the latter case (parity failed instead of data), the entire parity computation is suppressed, and the write proceeds as if the array were non-redundant.

3.2 RAID Level 1

The most implemented, most studied, and most optimized redundant disk array architecture is mirroring, also known as RAID level 1 [Solworth91, Bitton88, Polyzois93, Gray90]. Basically, two copies of every data unit are kept on two different disks. This architecture doubles storage costs, offers reads a choice of copies, and simplifies updating redundant data and recovering from failures.

Our RAIDframe implementation describes a basic RAID level 1 architecture with each data disk duplicated entirely, all writes concurrently scheduled to both disks, and adds a shortest queue scheduling optimization. Shortest queue is an approximation of the much harder to implement shortest seek scheduling policy that produces quite similar results [Wilkes95].

RAID level 1 DAGs are particularly simple: reads use RAID level 0 read DAGs and writes look like RAID level 0 DAGs with two concurrent copies of each access. The mapping code looks like the disks are divided into two collections, each striped just like RAID level 0.

3.3 Parity Declustering

In RAID levels 4 or 5, a user read of a data unit that resides on a disk which has failed can be satisfied by reading all other units in the associated parity stripe and computing their XOR. For a fixed user workload, these additional accesses cause the I/O rate observed at each surviving disk to be increased by approximately 60-80% [Holland94b]. The severity of the resulting performance degradation draws into question the use of such arrays in many highly available applications. To address this on-line failure recovery problem, Muntz and Lui [Muntz90] proposed an extension to RAID level 5 called parity declustering which decouples the number of units in a parity stripe from the number of disks in the array, and distributes the contents of each parity stripe. Illustrated in Figure 4, their idea has since been extended by mechanisms evaluated by analytical modeling and event-driven simulation [Holland92, Merchant92, Schwabe94, Ng92b]. These studies suggest that parity declustering should be able to improve user performance during on-line recovery and to dramatically reduce the duration of on-line reconstruction.

An implementation of parity declustering on C disks differs from an implementation of RAID level 5 on $G < C$ disks only in the mapping of units to disks. In our RAIDframe description of parity declustering, the request DAGs are the same as those used in RAID level 5, as are all other modules except the mapping module. Changes in the mapping module depend on the implementation technique; we have used the internet-published block designs approach [Holland92].

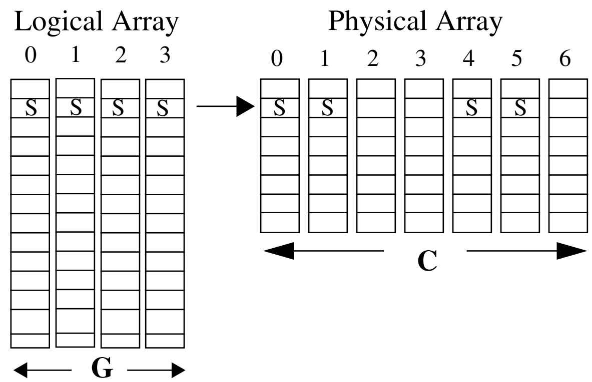

Figure 4: Parity declustering's logical to physical mapping

Parity declustering maps a logically narrow array (G disks) to a physical array which is wider (C disks). Mapping a narrow stripe onto a wide array has the desirable property that reconstructing a failed unit from the narrow stripe does not involve all disks in the physical array.

3.4 RAID Level 6

RAID levels 4 and 5 and parity declustering are tolerant of a single disk failure. If two disks are simultaneously failed in one array, they make no guarantee that data will not be irretrievably lost. In very large arrays, or in arrays requiring very high reliability, multiple concurrent failures may need to be tolerated [Blaum94, Burkhard93]. The most common multiple-failure tolerating array organization, RAID level 6, also known as *P+Q*, uses two redundant units per stripe, the *P* unit and the *Q* unit, to store a double-erasure-correcting Reed-Solomon code computed over the data portion of the stripe. The *P* unit contains the bitwise XOR over the data portion of the stripe, and the *Q* unit contains a linear non-binary code over the same data. A *P+Q* array achieves double failure tolerance, at the expense of maintaining an encoding on two redundant disks.

RAID level 6 differs from RAID level 5 in the mapping of logical addresses to physical, in the structure of the DAGs generated to perform reads and writes, and in the introduction of a new primitive, the *Q* computation node. The mapping functions describe a simple extension to RAID level 5 differing only in the addition and rotation of the *Q* redundancy units and the reduction by one of the number of user data units in a stripe. RAID level 6 DAGs are also extensions of RAID level 5 DAGs, though these extensions require more effort. While fault-free reads are unchanged versions of the corresponding RAID level 5 DAGs, Figure 2 shows how fault-free writes are changed for RAID level 6. The extra complexity in RAID level 6 DAGs arises from the requirement to operate in three states — fault-free, a single failed drive, or two drives failed. For example, when two disks are lost, a given stripe could have two data units lost, a data unit and *P* lost, a data unit a *Q* lost, or *P* and *Q* lost. Table 1lists the distinct DAG templates for RAID level 6 with a guide to when each is used.

3.5 Reconstruction

Reconstruction of lost data is implemented in RAIDframe using a disk-oriented algorithm [Holland94b]. A single reconstruction thread is logically located in parallel with the RAID execution engine. When invoked, this reconstruction thread issues, through the locking and DAG layers, a low-priority read request for the next unit on each disk required for reconstruction. As each reconstruct read completes, its data is XORed into the accumulating "sum" for the indicated stripe, and the next read request for that disk is issued. When the last unit associated with a particular stripe has been read and summed, the reconstruction thread issues a low-priority request for the now reconstructed data to be written to a replacement or spare disk. This algorithm allows reconstruction to keep one low-priority disk request in the queue for each physical disk at all times, maximizing the efficiency of reconstruction without significantly penalizing user responsiveness.

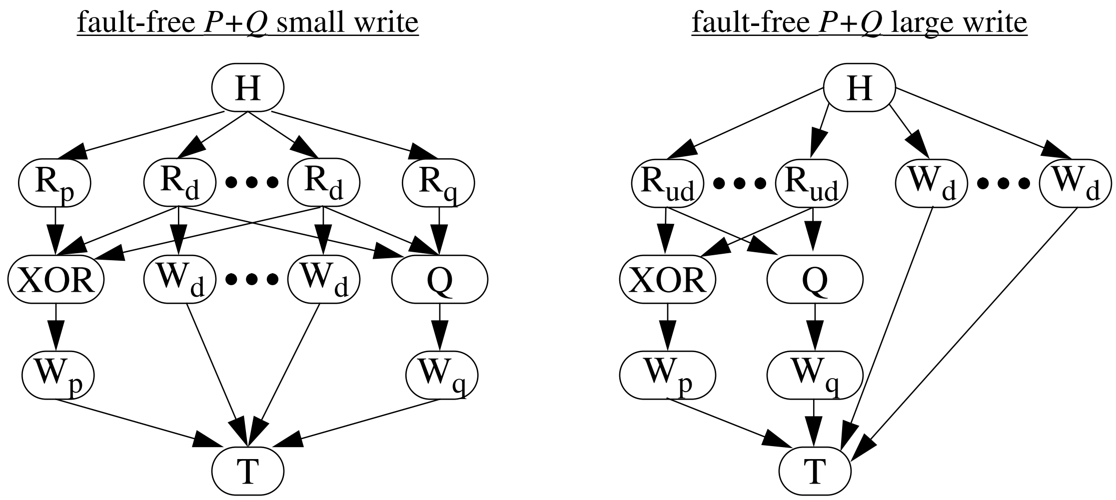

Figure 5: Fault-free write DAG templates for RAID level 6

The DAG templates above use the same nomenclature as Figure 2. A “Q” node computes Reed-Solomon check symbols identified by subscript “q”. Note that a small write, therefore, involves a least three drives each doing a read-modify-write sequence.

3.6 Case Study Prototyping Costs

The description of most of our example architectures was done early enough to directly influence the design of RAIDframe. The addition of RAID level 1 (mirroring), however, was done while this paper was being written and a record of effort was kept. The overall effort was 460 minutes during which 370 lines of code were produced or modified. Broken down, the time until first compilation attempt was 90 minutes, with first successful compilation 15 minutes later. Read DAGs first ran 55 minutes later, writes DAGs 105 minutes later, read scheduling was implemented after 145 minutes more and degraded mode was operation in an additional 50 minutes.

While we do not have development times for the other architectures, we know the order in which they were added to RAIDframe and the resulting code changes.Table 2 shows the code reuse exhibited during development of these architectures. We measure code reuse as the ratio of architecture specific code over infrastructure code at the time of development (including previously added architectures).

4. Experiments

4.1 Evaluation Configuration

Figure 6 illustrates our hardware environment. The host system is a 150 MHz DEC 3000/500 running Digital UNIX version 3.2, and has five DEC KZTSA fast-wide-differential SCSI adapters on its 100 MB/s Turbochannel bus. These adapters connect to five shelves of disks in a DEC Storageworks 800 cabinet. Each shelf contains a DEC DWZZA fast-wide-differential to fast single-ended SCSI converter, and three one-gigabyte Hewlett-Packard model 2247 disk drives. In our evaluations, we use a single parity group consisting of fifteen disks on five busses, neglecting the dependent failure mode that would occur should a SCSI bus controller fail [Gibson93].

DAG Template	Conditions for invocation				
	P	Q	Acc data	Unacc data	% of stripe
RAID level 5 fault-free read	S	S	0	0	N/A
RAID level 5 degraded read	S	S, F	1	0	N/A
RAID level 5 degraded read with P replaced by Q	F	S	1	0	N/A
degraded read using both P and Q	S	S	1	1	N/A
RAID level 5 fault-free small write plus Q RMW	S	S	0	0,1,2	< 1/2
RAID level fault-free large write plus Q write	S	S	0	0,1,2	≥ 1/2
small write with RMW of P suppressed	F	S	0	0	< 1/2
large write with write of P suppressed	F	S	0	0	> 1/2
small write with RMW of Q suppressed	S	F	0	0	< 1/2
large write with write of Q suppressed	S	F	0	0	> 1/2
recover unaccessed data, then degraded write	S	S	1	1	N/A
degraded write updating both P and Q	S	S	1,2	0	N/A
non-redundant write	F	F	0	0	N/A
degraded write using P	S	F	1	0	N/A
degraded write using Q	F	S	1	0	N/A

Table 1. The RAID level 6 DAG templates and conditions for the invocation

“S” indicates a surviving disk, and “F” a failed disk. “0”, “1”, and “2” refer to the number of disks of the indicated type that are failed. “< 1/2” and “≥ 1/2” indicate, respectively, that less than 1/2 and greater than 1/2 of the data portion of the affected stripe is being updated by a write operation. “Acc data” and “Unacc data” refer, respectively, to the portion of the stripe being updated by a write operation, and the portion not being updated, if any. RMW is a read-modify-write sequence.

RAIDframe has both a synthetic and a trace-drive load generator. The synthetic generator conforms its load to a script containing a variable number of access profiles with individual occurrence probabilities. Each profile defines a deterministic or exponentially distributed access size with a given mean and alignment. Access addresses are randomly generated throughout the entire address space, or with a given probability, within a single locality specified with each profile. Access types are either read, write or sequential (the same as the last access with its address advanced).

For the experiments reported in this paper we use RAIDframe’s trace-driven workload generator. This allows us to apply identical, high-concurrency access sequences to an architecture executing in simulation, at user-level and as a device driver. Traces understood by RAIDframe contain an explicit sequence of tuples: (thread id, delay time before issuing this request, read or write, block address, number of blocks, and a requester-waits/requester-does-not-wait flag).

Architecture	Lines of Code	New Files	% Code Reuse
RAID level 0	34,311	122	—
parity declustering	2,416	7	93.4
RAID level 5	360	3	99.1
RAID level 4	139	2	99.6
RAID level 6	3,632	7	91.1
RAID level 1	408	2	99.0

Table 2. Cost of creating new array architectures

At the time of this writing, RAIDframe has been extended to include a total of six architectures (with three more in progress and expected by publication). Beginning with a baseline implementation which supported RAID level 0, the above architectures were added in which they appear in the table. "Lines of Code" represents lines of code either added or modified to the previously added architecture. "% Code Reuse" represents the number of lines of unchanged (reused) code as a percentage of the total at that time.

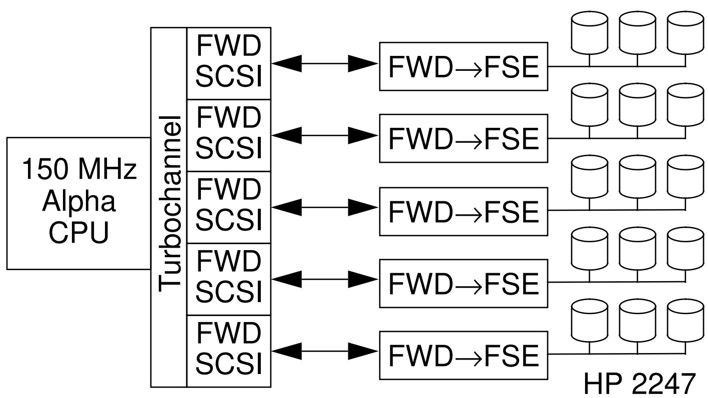

Figure 6: Test system for evaluating RAIDframe.

The measurements reported in this paper are drawn from microbenchmark experiments. We use microbenchmarks that are broadly familiar to readers of RAID related papers: random small accesses all of the same type. Specifically, our tests use 1, 2, 5, 10, 15, 20, 30, 40 threads to issue blocking 4 KB operations on random 4 KB aligned addresses with no intervening delay. We show results for 100% reads and 100% writes separately. All results display the average of three experiment runs, each with a distinct seed.

4.2 RAIDframe Efficiency

Although RAIDframe is structured as an engine that interprets architectural specifications to implement an array, its structure does not substantially impede the performance of its accesses. To evaluate the overhead involved in RAIDframe, we compare its in-kernel RAID level 0 implementation to a simpler pseudo-device disk striper. This striper was developed independently, based on code from 4.4BSD-Lite, to provide high performance access for a file system research project [Patterson95].

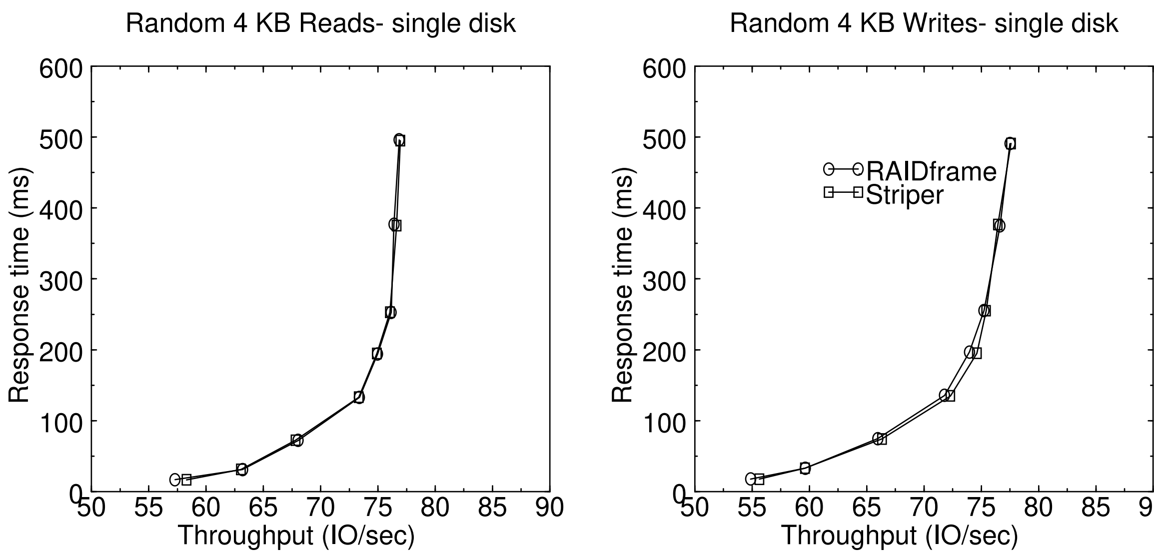

Figure 7: Single disk performance of striper and RAIDframe.

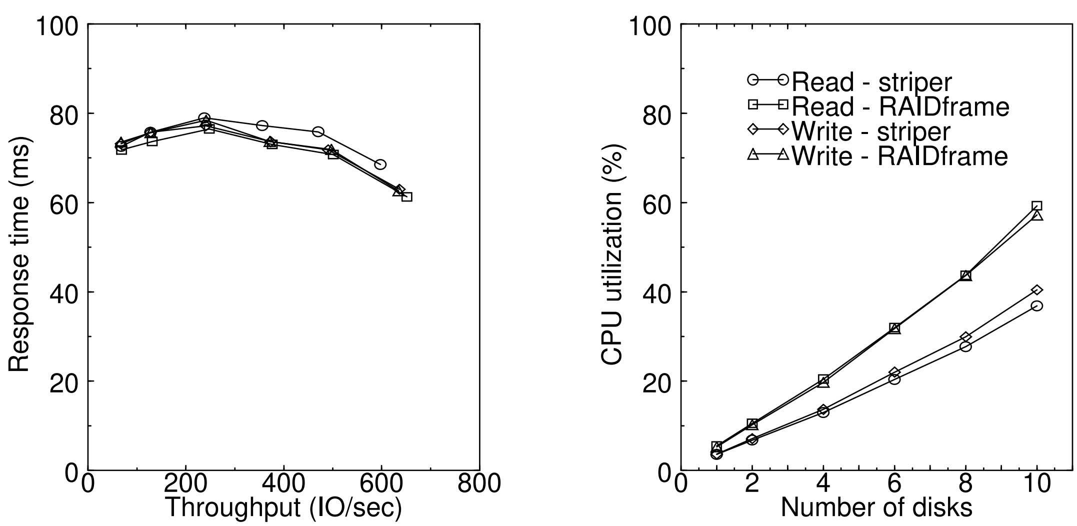

Figure 8: RAID level 0 performance of striper and RAIDframe.

We first consider its simplest configuration — managing a single disk. Figure 7 shows the negligible difference in disk throughput and average access response time for our microbenchmark set running against a single disk. While no difference is apparent in I/O performance, RAIDframe is consuming more of the host machine's CPU cycles; the striper consumes 2.9% - 4.6% of the host CPU while RAIDframe consumes 4.2% - 6.7%.

Next we consider the impact of RAIDframe's higher demand for CPU cycles on its performance in larger arrays. Figure 8 shows both CPU utilization and total throughput versus average response time for our microbenchmarks running on 1, 2, 4, 6, 8, and 10 disks with 5 user threads per disk. RAIDframe's computational overhead for RAID level 0 (no parity computations) causes it to consume 50% more processor cycles than the striper at all loads. However, this heavier computational load does not substantially reduce RAIDframe's ability to exercise an array efficiently.

4.3 Microbenchmark Case Study Evaluations

In this section we report measurements from simulation, user-level RAIDframe and in-kernel RAIDframe for each of the case study architectures described in Section 3. Since most of these case study architectures are well known, so is their relative performance. Specifically, based on the simple performance model given by Patterson, Gibson and Katz [Patterson88], we expect small, fault-free read accesses to achieve the same performance in all architectures except RAID level 1, whose shortest queue discipline should improve throughput and decrease response time [Chen90, Bitton88]. For fault-free small writes, we expect minimum average response time for the parity-based architectures to be about twice that of the direct write architectures (RAID levels 0 and 1). Moreover, we expect the ratio of the maximum throughputs achieved in a workload of 100% small, fault-free writes in a 10 disk array to be
1:10/6:10/4:10/2:10 corresponding to RAID level 4, RAID level 6, RAID level 5 (and declustered parity), RAID level 1, and RAID level 0, respectively [Patterson88].

Figure 9 shows that the request throughput versus average request response time for our microbenchmarks bear out these expectations in all three evaluation environments. Moreover, because there is almost no difference in the measurements between in-kernel and user-level RAIDframe performance, array researchers unable or unwilling to port RAIDframe's in-kernel implementation to their operating system can be confident of the validity of user-level performance results.

While simulation results demonstrate all the correct trends, the absolute numbers are much higher than in the other two real system measurements primarily because RAIDframe does not model the amount of time corresponding it its execution. In simulation, RAIDframe treats all computation as taking zero time (infinitely fast processor). In fact, the CPU utilization during in-kernel RAIDframe tests gets as high as 62% over all microbenchmarks.

5. Conclusions

RAIDframe is a software redundant disk array implementation structured for rapid prototyping. Its structure separates the large body of rarely changing execution mechanism from the smaller, much more interesting architectural policies. RAIDframe provides policy control over operation sequencing, operation definition, data and parity mapping, cache management, queue discipline, and, in simulation, disk implementation. Of particular interest to architectural experimentation is the combination of operation sequencing, expressed as directed acyclic graphs in RAIDframe, and layout. In our experience most array architectures are defined primarily by their layout and operation sequencing. Currently we have libraries of RAIDframe implementations for RAID levels 0, 1, 4, 5, 6, and parity declustering arrays.

In our experience modifying and experimenting with an architecture in RAIDframe is much faster and less error-prone than doing the same in a detailed simulation environment, such as raidSim. Moreover, RAIDframe uses the same policies and mechanisms running as a simulator, as a user process controlling real disks through the UNIX raw device interface, and as an in-kernel device driver that can mount a usable file system in the Digital UNIX operating system. While its rapid prototyping structure does make significantly heavier demands on its hosts CPU than a much simpler disk striper, it is able to obtain comparable storage performance from an array of up to 10 disks.

RAIDframe's simulator and user level implementations, including the library of array architecture described in this paper, are available for public use on our internet web pages.

While our motivation for developing RAIDframe was to simplify the process of developing, modifying and evaluating array architectures, having it provides new opportunities for array research. First, of course, is its use for developing new array architectures or experimenting with hybrids of existing narrowly focused architectures. Next, approximately concurrently with our development of RAIDframe we have been developing mechanisms for automating the effect of errors on array controller software. By constraining operation sequences and adding to RAIDframe's execution engine rollforward and rollback functions, we intend to virtually eliminate from a designer's concerns the problem of completing requests in flight at the time of an error. Particularly interesting in this effort is the potential to prove correctness of implementation policies, up to the

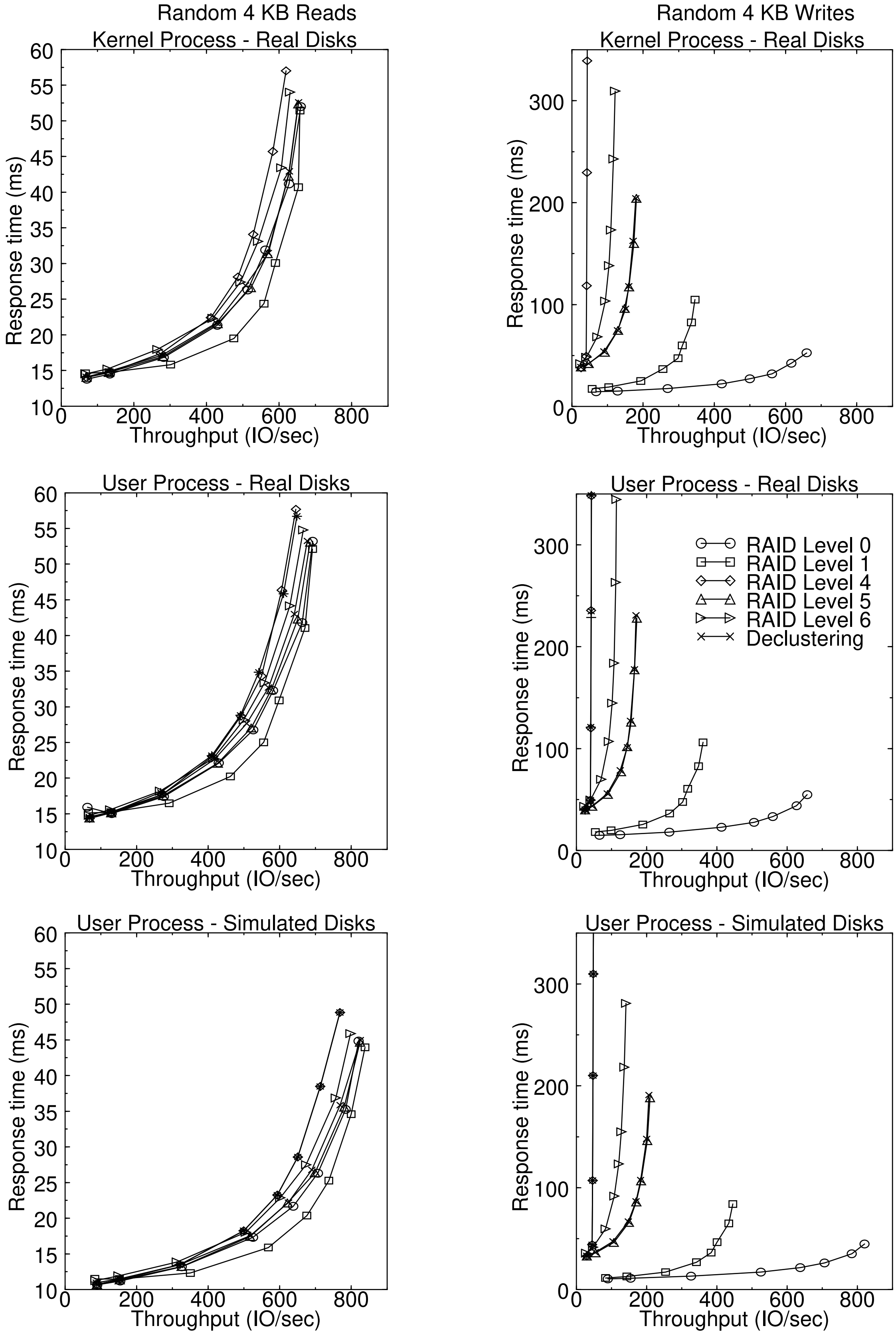

Figure 9: Case study performance of microbenchmarks in RAIDframe.

correctness of the unvarying engine. Next, since operation sequences are expressed in RAIDframe by simple DAGs, it is possible to automatically manipulate them. We currently decompose a request into DAGs for each full stripe, full stripe unit in a partial stripe, and individual block in a partial stripe unit, then compose a single large DAG from these constituent DAGs. Static or dynamic optimization of composed DAGs is certainly possible. Next, RAIDframe can be viewed as an exercise in interpretive language development. As such, it should be amenable to being "compiled" into a globally optimized instance for each collection of architectural policies.

6. Acknowledgments

Aside from the authors, Khalil Amiri, Claudson Bornstein, Robby Findler, LeAnn Neal-Reilly, Daniel Stodolsky, Alex Wetmore, and Rachad Youssef have been active participants in the development of RAIDframe. We also thank Ed Lee who was the driving force behind raidSim, from which we have learned much.

The project team is indebted to the generous donations of the member companies of the Parallel Data Laboratory (PDL) Consortium. At the time of this writing, these include: Data General, Digital Equipment, Hewlett-Packard, International Business Machines, Seagate, Storage Technology, and Symbios Logic. Symbios Logic additionally provided a fellowship. The Data Storage Systems Center also provided funding through a grant from the National Science Foundation under grant number ECD-8907068. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the PDL Consortium member companies or the U.S. government.

7. References

[Accetta86] M. Accetta, et. al "Mach: A New Kernel Foundation for Unix Development," Proceedings of the Summer 1986 USENIX Workshop, p. 93-113, 1986.

[ATC90] *Product Description, RAID+ Series Model RX,* Array Technology Corporation, Boulder, CO, 1990.

[Bitton88] D. Bitton and J. Gray, "Disk Shadowing," *Proceedings of the 14th Conference on Very Large Data Bases*, 1988, pp. 331-338.

[Blaum94] M. Blaum, J. Brady, J. Bruck, and J. Menon, Evenodd: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures, *Proceedings of the International Symposium on Computer Architecture*, 1994, pp. 245-254.

[Brown72] D. Brown, R. Gibson, and C. Thorn, "Channel and Direct Access Device Architecture," *IBM Systems Journal,* 11(3), pp. 186-199, 1972.

[Burkhard93] W. Burkhard and J. Menon, "Disk Array Storage System Reliability," *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1993, pp. 432-441.

[Cabrera91] L.-F. Cabrera and D. Long, "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates," *Computing Systems*, vol. 4 no. 4, 1991, pp. 405-439.

[Cao93] P. Cao, S.B. Lim, S. Venkataraman, and J. Wilkes, "The TickerTAIP parallel RAID architecture," *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 52-63.

[Chen90] P. Chen, et. al., "An Evaluation of Redundant Arrays of Disks using an Amdahl 5890," *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 74-85.

[Chen90b] P. Chen and D. Patterson, "Maximizing Performance in a Striped Disk Array," *Proceedings of International Symposium on Computer Architecture*, 1990, pp. 322-331.

[Courtright94] William V. Courtright II and Garth A. Gibson, "Backward error recovery in redundant disk arrays." Proceedings of the 1994 Computer Measurement Group (CMG) Conference, Vol. 1, December 4-9, 1994, pp 63-74.

[DISK/TREND94] DISK/TREND, Inc. 1994. *1994 DISK/TREND Report: Disk Drive Arrays*. 1925 Landings Drive, Mountain View, Calif., SUM-3.

[Drapeau94] A.Drapeau, K.Shirriff, J. Hartman, E. Miller, S. Seshan, R. Katz, D. Patterson, E. Lee, P. Chen, and G. Gibson, "RAID-II: a High-Bandwidth Network File Server," *Proceedings the 21st Annual International Symposium on Computer Architecture,* pp. 234-44, 1994.

[English92] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing storage device. In *Proceedings of the 1992 Winter Usenix Technical Conference*, pages 237–251, January 1992.

[Ganger93] G. Ganger, and Y. Patt, "The Process-Flow Model: Examining I/O Performance from the System's Point of View," *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, pp. 86-97, 1993.

[Gibson93] G. Gibson and D. Patterson, "Designing Disk Arrays for High Data Reliability," *Journal of Parallel and Distributed Computing*, vol. 17, 1993, pp. 4-27.

[Gray90] G. Gray, B. Horst, and M. Walker, "Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput," *Proceedings of the Conference on Very Large Data Bases*, 1990, pp. 148-160.

[Hartman93] J. Hartman and J. Ousterhout, "The Zebra Striped Network File System," *Proceedings of the Symposium on Operating System Principles*, 1993.

[Holland92] M. Holland and G. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 23-25.

[Holland94b] M. Holland, G. Gibson, D. Siewiorek, "Architectures and Algorithms for On-line Failure Recovery in Redundant Disk Arrays," *Journal of Distributed and Parallel Databases*, vol 2, pp. 295-335, 1994.

[Kim86] M. Kim, "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, vol. 35 no. 11, 1986, pp. 978-988.

[Kistler92] J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Transactions on Computer Systems*, vol. 10 no. 1, 1992, pp. 3-25.

[Lawlor81] F. D. Lawlor, "Efficient mass storage parity recovery mechanism," IBM Technical Disclosure Bulletin

[Lee90] E. Lee, "Software and Performance Issues in the Implementation of a RAID Prototype," University of California, Technical Report UCB/CSD 90/573, 1990.

[Lee91] E. Lee and R. Katz, "Performance Consequences of Parity Placement in Disk Arrays," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 190-199.

[**Long94**] D. Long, B. Montague, and L.-F. Cabrera, "Swift/RAID: A Distributed RAID System," *Computing Systems*, 3(7), pp. 333-359, 1994

[Menon92a] J. Menon and J. Kasson, "Methods for Improved Update Performance of Disk Arrays," *Proceedings of the Hawaii International Conference on System Sciences*, 1992, pp. 74-83.

[Menon93] J. Menon and J. Cortney, "The Architecture of a Fault-Tolerant Cached RAID Controller," *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 76-86.

[Merchant92]A. Merchant and P. Yu, "Design and Modeling of Clustered RAID," *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1992, pp. 140-149.

[Mogi94] K. Mogi and M. Kitsuregawa, "Dynamic Parity Stripe Reorganizations for RAID5 Disk Arrays," *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, 1994, pp. 17-26

[Muntz90] R. Muntz and J. Lui, "Performance Analysis of Disk Arrays Under Failure," *Proceedings of the Conference on Very Large Data Bases*, 1990, pp. 162-173.

[NCR91] *NCR 53C720 SCSI I/O Processor Programmers Guide,* NCR Corp., Dayton OH, 1991.

[Ng92b] S. Ng and R. Mattson, *Uniform Parity Group Distribution in Disk Arrays*, IBM Research Division Computer Science Research Report RJ 8835 (79217), 1992.

[Patterson88] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the ACM Conference on Management of Data*, 1988, pp. 109-116.

[Patterson95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, Jim Zelenka, "Informed Prefetching and Caching," to appear in *Proceedings of the 15th Symposium on Operating Systems Principles*, Dec. 1995.

[Polyzois93] C. Polyzois, A. Bhide, and D. Dias, "Disk Mirroring with Alternating Deferred Updates," *Proceedings of the Conference on Very Large Data Bases,* 1993, pp. 604-617.

[Ruemmler94] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *IEEE Computer*, 27(3):17-28, 1994

[Schwabe94] E. Schwabe and I. Sutherland, "Improved Parity-Declustered Layouts for Disk Arrays," draft submission to the *Symposium on Parallel Algorithms and Architectures*, 1994.

[Solworth91] J. Solworth and C. Orji, "Distorted Mirrors," *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1991, pp. 10-17.

[STC94] Storage Technology Corporation, *Iceberg 9200 Storage System: Introduction*, STK Part Number 307406101, Storage Technology Corporation, Corporate Technical Publications, 2270 South 88th Street, Louisville, CO 80028

[Stodolsky94] Daniel Stodolsky, Mark Holland, William V. Courtright II, and Garth Gibson, "Parity-Logging Disk Arrays," *Transactions on Computer Systems*, 12(3), August, 1994, pp. 206-235.