The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

SIMULATING RAID STORAGE SYSTEMS FOR PERFORMANCE ANALYSIS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

University of Cape Town

By
Sameshan Perumal
November 2007

Supervised by
Dr P. S. Kritzinger

© Copyright 2007
by
Sameshan Perumal

University of Cape Town

Acknowledgements

To my family, Mom, Delisha and Marlan, for believing in me.

To my friends for their valiant efforts at keeping me sane.

To Thamaray, for being there.

University of Cape Town

Abstract

Redundant Array of Independent Disks (RAID) provides an inexpensive, fault-tolerant storage solution using commodity hard-drives. RAID storage systems have recently surged in popularity amongst enterprise clients, as they provide economical, scalable, high-performance solutions for their storage requirements.

The performance of RAID systems is negatively affected by the overhead required to manage and access multiple drives, and multiple disk failures can result in data loss. As RAID has developed, various improvements have been devised by both academia and business to address these shortcomings. These improvements have suggested improved architectures to increase performance and new coding techniques to protect against data loss in the case of drive failure.

Evaluating the effect on performance of these improvements is greatly simplified by the use of discrete-event, software simulations. The RAID Operations Simulator for Testing Implementations (RÖSTI) was developed to support such simulation experiments. RÖSTI is a modular, extensible, RAID-focused simulation environment, built on the OMNet++ framework, and able to run on both the Microsoft Windows™ and GNU/Linux™ platforms.

In this environment, RAID storage systems can be modelled using the graphical tools provided by OMNet++. The RAID model adopted by RÖSTI is modular in nature, with the intention that extensions can be added transparently. The configuration of these models, once specified, is also achieved through the RÖSTI graphical user interface. The configured simulation models can then be executed, and the results analysed from within RÖSTI.

The work of Courtight et. al. [CGHZ96b] on representing RAID operations as Directed Acyclic Graphs (DAGs) served as the basis on which extensive validation of RÖSTI was performed. This validation was also extended to the various caching schemes that are implemented in RÖSTI. Test runs of the system were performed and the results compared against expectations as additional verification of RÖSTI.

RÖSTI currently supports modelling of RAID 5, RAID 6 and SPIDRE protection schemes, as well as LRU, LFU and ARC caching schemes.

University of Cape Town

Contents

University of Cape Town

University of Cape Town

University of Cape Town

Chapter 1

Introduction

University of Cape Town

Over the recent history of computer systems, there has been a phenomenal increase in both raw processing power (as predicted by Moore's Law) and primary memory (RAM) capacity, but this has not been adequately matched by the performance of secondary memory (disk drives). Neither capacity nor transfer rates have experienced serious improvements, yet disk requirements are soon anticipated to reach PetaByte[1] levels [KBC+00].

Single Large Expensive Disks (SLED) have previously been the solution of choice amongst organisations with large storage requirements, as the Mean Time to Failure[2] (MTTF) is high, and reliability and persistence of storage is of utmost importance to most organisations. They do suffer from problems however. SLEDs are expensive and can require large amounts of power to operate. More importantly SLEDs have limited bandwidth, which means recovering a backup from tertiary storage[3] to secondary storage could take prohibitively long. This last factor is especially important for online applications, where constant availability is essential, and the non-accessibility of backed-up data could be crippling.

A solution to this problem is provided by the widespread availability of commodity hard disk drives for personal PCs. They offer a number of benefits: they are cheap, with a lower cost per Megabyte (MB) than SLEDs; an individual disk now has a MTTF comparable to most SLEDs; they require far less power; they conform to a uniform access standard, typically SCSI (Small Computer Systems Interface), IDE (Integrated Drive Electronics) or SATA (Serial Advanced Technology Attachment); and they have built in controller logic which performs both error detection and correction functions.

The concept of a Redundant Array of Inexpensive Disks (RAID) was introduced to harness the potential of these commodity hard drives: Patterson et. al. [PGK88] established the RAID taxonomy in 1988. RAID overcomes the capacity limitations of commodity disks by exposing an array of such low-capacity disks as a virtual SLED.

Such arrays offer flexibility over SLEDs, in that capacity can be increased incrementally, and as desired, by adding more disks to the array. Since each individual disk in a RAID system consumes little power, and the cost of replacement of a single disk is small compared to the overall cost, RAID systems have low associated running costs. Finally, given the fact that each disk in the

[1] $1\text{PB} = (1024)^6$ B

[2] The average time for a single disk in a group to fail.

[3] High capacity, low performance storage solutions, such as Tape Drives

array can perform transfers independently of the others, due to the built-in controllers, there exists the possibility of increased I/O bandwidth due to parallelism.

All these advantages are not without a significant problem: reliability. The Mean Time To Failure (MTTF) of an array of disks is inversely proportional to the number of disks in the array, given by [PGK88]:

$$\text{MTTF of RAID} = \frac{\text{MTTF of Single Disk}}{\text{Number of Disks in Array}}$$

To illustrate the effect of this, consider an array of 100 disks, each with a MTTF of 30 000 hours. The MTTF of any one disk in the array is then 300 hours or 12.5 days. Ensuring that data loss does not occur in a RAID is thus essential. Almost all current RAID architectures focus on preventing data loss from a single disk failure. They are generally susceptible to data loss if a second failure occurs before the first is repaired. The two main strategies employed are mirroring [BG88] and striping with parity [LKB87]. Both these approaches utilise redundancy information to achieve reliability.

The applications for RAID (Redundant Array of Independent Disks) [PGK88] in Enterprise Storage Systems (ESS) continue to grow as applications demand ever more secondary storage. New protection schemes are being used in commercial systems to meet this demand.

These schemes increase the number of disk failures that are tolerable before data loss occurs. In organisations with large arrays of disks, this is of particular importance, since the Mean Time To Failure (MTTF) of the array decreases as the number of disks increase. The increased number of disks across which data can be distributed can also improve data transfer rates for certain types of I/O operations, as well as allowing multiple operations to occur in parallel.

The benefits offered by these new schemes are offset by a number of factors: write operations take longer, since more protection information must be updated on each operation; certain schemes reduce the level of parallelism possible; a greater proportion of the available storage space is used to store protection information; rebuilding lost data after a disk failure is a longer and more complicated procedure; and the complexity of the associated controller increases, which introduces a greater possibility for errors in operation.

University of Cape Town

Motivation

Our work here came about as a result of collaboration with the IBM Zürich Research Lab, which is involved in developing new, more effective schemes for RAID storage. These schemes aim to decrease the potential for irrecoverable data loss with a minimal increase in capacity and processing overhead. The first such scheme under development is **Sector Protection through Intra-Drive REdundancy** (SPIDRE), described in Section 2.3.6 (p. 13).

The goal of developing the **Raid Operation Simulator for Testing Implementations** (RÖSTI) was born out of the desire to implement and test the performance of these new RAID schemes using a common, extensible simulation environment. This simulator allows for advanced RAID schemes, such as those under development by IBM, to be performance tested prior to implementation, thus allowing for more cost-effective development. The extensibility of the simulator is thus essential to this goal.

Our secondary goals are to ensure that RÖSTI is configurable in a simple, graphical manner to allow inexperienced users access to its capabilities. A further goal is to assist in first-order analysis[4] of results produced by RÖSTI, thus allowing simulations to be fine-tuned as part of the simulation process.

Contribution

Our contributions to the area of RAID storage system research are as follows:

- Development and validation of a RAID simulation environment (RÖSTI) that allows RAID simulation models to be specified, configured, executed and analysed. RÖSTI supports the RAID 5, RAID 6 and SPIDRE protection schemes[5], and the LRU, LFU and ARC caching algorithms[6]. This simulation environment was further designed to be modular and extensible[7].
- Provision of a Graphical User Interface (GUI) to this simulation environment, thereby allowing inexperienced users to utilise the functionality provided without requiring a detailed knowledge of the system internals[8].
- Creation of a validation strategy for RAID implementations focused on high-level behaviour rather than low level implementation[9]. This strategy uses the extensive work done by Courtright et. al. [CGHZ96a] on formalising RAID Controller operations, and applies it to ensuring the correctness of a particular implementation.
- Outlining a possible avenue of research into a formalisation strategy for RAID Controller operations[10]. This strategy would allow the individual disk operations required for a given array operation to be automatically inferred from a high-level description of the RAID scheme (RAID 5, RAID 6, etc.) in use.

University of Cape Town

[4] Initial examinations of results that are used to refine or redefine a simulation experiment.
[5] Section 5.6.3 (p. 78)
[6] Section 5.6.3 (p. 79)
[7] Section 5.4 (p. 60)
[8] Figures 35 (p. 88), 63 (p. 118), and Chapter 7 (p. 91)
[9] Section 8.3 (p. 97)
[10] Appendix A (p. 133)

Part I

Background

University of Cape Town

Chapter 2

Redundant Array of Independent Disks (RAID)

University of Cape Town

Describing the nature of a RAID system requires the use of specialised terminology, covered in Section 2.1. These systems are composed of a number of individual components, that frequently interact in complex ways. Section 2.2 provides an overview of this aspect of RAID. A taxonomy of the various RAID schemes available is presented in Section 2.3 and their associated drawbacks are discussed in Section 2.4. Finally the issue of caching, its impact on RAID systems and an overview of some widely implemented caching policies is presented in Section 2.5.

2.1 Terminology

RAID technology usually requires the distribution of data across a number of disks. We define here those terms used to more clearly describe the various ways in which this can be accomplished.

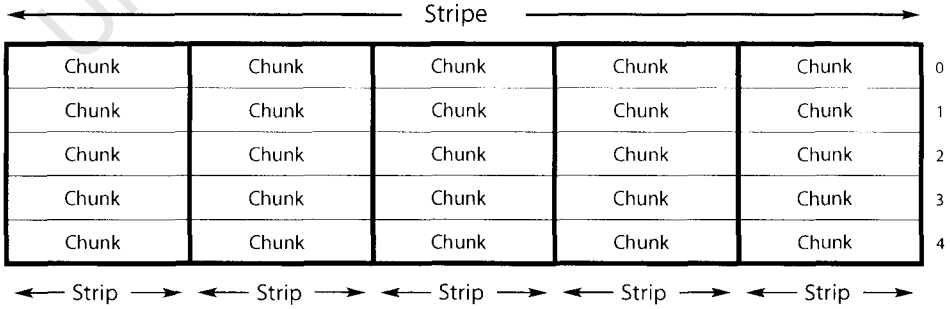


Figure 1: Illustration of the relation between Stripes, Strips and Chunks. The numbers to the far right indicate numbering of chunks within a strip.

Restricting ourselves to one-dimensional arrays[1], we term the ordered set of all disks participating in a RAID configuration an **Array**. Referring to Figure 1, each column (outlined in thick black) represents a single disk in the array.

[1] See Section 2.2.4 (p. 8)

Each row represents a **Stripe**, such that a **Stripe** spans all disks in the **Array**. A **Stripe** represents the smallest unit of protection in an **Array**– any lost data within a **Stripe** can be recovered using only the surviving data within that **Stripe**.

A **Stripe** consists of a sequence of **Strips** corresponding to the disks in the **Array**. A **Strip** is the smallest unit of data that will be accessed on a disk, and the *strip size* is thus chosen to match the normal size of a physical disk sector, usually 512KB. For most common RAID schemes (RAID 1 - 5), these are the only terms that are necessary.

For more complex schemes, such as SPIDRE[2] and EvenOdd [BBBM94], it is necessary to introduce one additional term. Each **Strip** can be logically segmented into a contiguous series of one or more **Chunks**. A **Chunk** represents the smallest unit of data on which a RAID Controller can operate. The purpose and size of each chunk is dependent on the RAID scheme used.

2.2 RAID Implementation

RAID implementations can exist in either software or hardware, with some occasional implementations using hybrid combinations. The majority of commercial implementations utilise a dedicated hardware implementation, however, since the overhead imposed on the CPU by having to handle RAID logic diminishes the performance gains of RAID significantly. The majority of such systems are connected to a single host system using either a commodity disk interface (IDE, SATA or SCSI) or enterprise level storage interfaces (Fibre Channel, iSCSI). Whichever solution is chosen, the RAID appears as a single disk to the host system which is accessed normally over the given channel.

2.2.1 Hardware

In a hardware RAID system, all processing and management of the RAID array is offloaded to a dedicated processor in hardware, referred to as the RAID Controller. This controller performs parity checking, management of recovery from disk failures, and the physical striping of data across multiple disks. Internally, drives are attached using IDE, SATA or SCSI interfaces. Hardware solutions present the RAID array to the host system as a single disk. Configuration of various RAID parameters (such as stripe size, strip size, RAID mode) is handled by external utilities that interface with the hardware. Parallel performance of such a system is effectively equivalent to a single server, with a single large disk, serving requests from multiple clients.

2.2.2 Software

In a software RAID system, a software driver performs all necessary operations to treat an array of drives as a RAID. The CPU load increases because of this overhead, and can thus present a performance bottleneck in multitasking situations. Drives in the array are connected via one of the commodity disk interfaces mentioned above. The software driver allows the the Operating System to access the array as a single, large disk.

[2] See Section 2.3.6 (p. 13)

University of Cape Town

2.2.3 Parallelism in RAID

Since clients are connected to the RAID via a serial access channel, parallel access by multiple clients is not explicitly supported. Some implementations offer multiple access channels, but each channel can only access a designated part of the RAID. This is achieved by partitioning the RAID and allowing each channel exclusive access to a subset of the partitions. If the partitions are on physically separate volumes, true parallel access is supported, since each channel can independently access its associated partitions as exclusive disk access is possible. If all disks are shared by all partitions, this is not possible.

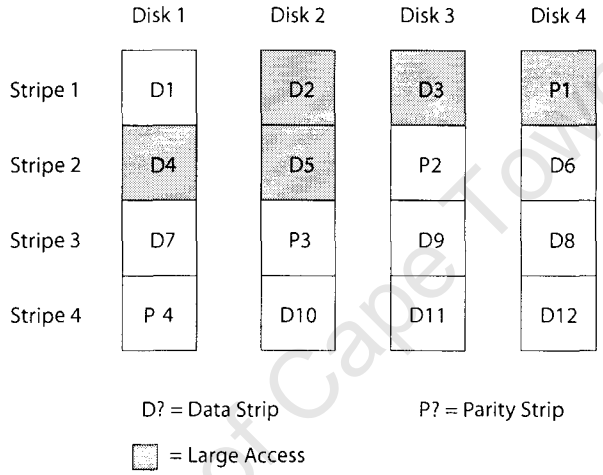


Figure 2: A single large I/O request that accesses all disks in the array. No other request can be executed while the large request is being serviced.

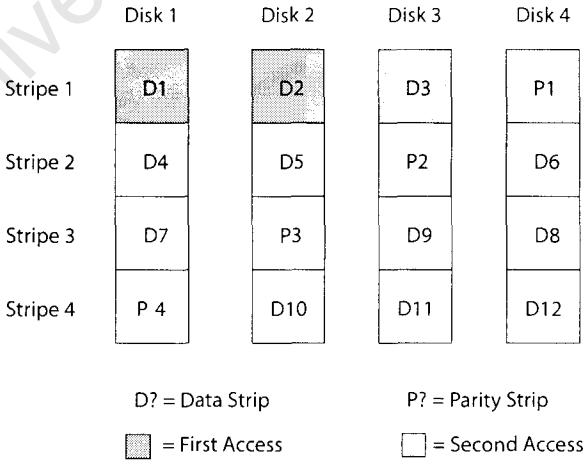


Figure 3: Two small I/O requests that do not require shared access to any disk, and can thus be executed in parallel.

The extent to which parallelism is possible in an array is thus primarily a question of how queued requests are scheduled to run in parallel. This can be answered by examining the nature of I/O requests presented to the system. In particular, a distinction can be made between large requests (that require access to at least a full stripe) and small requests (which require access to only a subset of strips within a stripe).

University of Cape Town

All large accesses must be performed in serial, whereas most small accesses can be parallelised. Large accesses can only be serviced one at a time, since all disks are necessarily involved in the transfer and disks are themselves serial. Hence, none of them can be used until the transfer is complete, as illustrated in Figure 2.

Small accesses, on the other hand, can be serviced in parallel if each access addresses different disks, as illustrated in Figure 3. It follows that the smaller the accesses, the greater the degree of parallelism possible, with the maximum number of simultaneous accesses determined by the number of disks in the array.

2.2.4 Logical to Physical Mapping

As described previously, data in RAID arrays is divided into discrete blocks (*strips*) that are then *striped* across a series of drives in the array. This striping requires a well defined system to determine where each strip will be physically stored. The only requirement of such a system is that each strip in a given stripe must be placed on physically independent disks.

Given this criteria, the most commonly used system matches the number of strips in a stripe to the number of available physical disks, as illustrated in Figure 2. This system has the advantage of requiring the minimal number of parity strips for a given array size. A significant downside, however, is that rebuild operations[3] on the array cannot take advantage of parallelism as each stripe reconstruction uses all disks in the array.

University of Cape Town

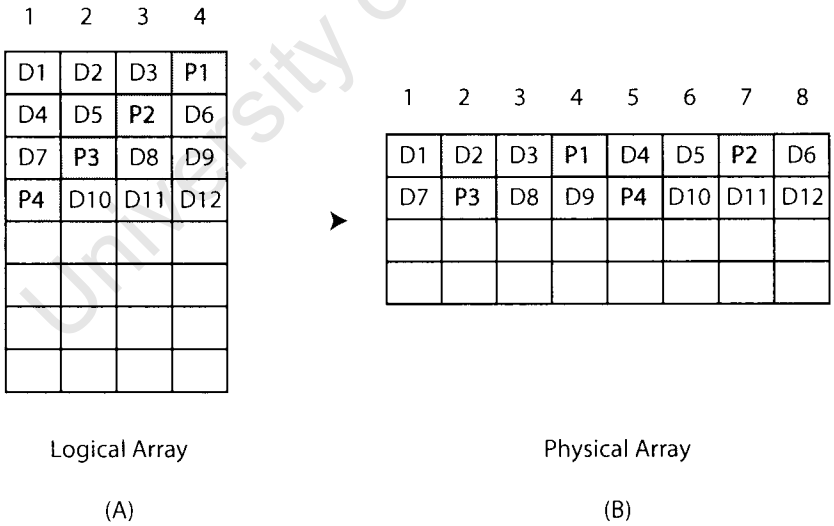


Figure 4: Parity Declustering mapping a logical array mapping (Figure A) to a physical array mapping (Figure B). Note that the physical independence of data and parity strips in a stripe is maintained, since each is located on an independent physical disk.

An alternate solution to this problem was presented by Holland et. al. in their work on Parity Declustering [Hol94]. In this solution, the number of disks in the array is greater than the number of strips per stripe. The array layout is then logically identical to the previous case in Figure 2. However at a physical level, the strips can be mapped to the individual disks in a variety of ways. The most straightforward of these is illustrated in Figure 4. Each disk is partitioned into a series

[3] See Section 2.4.2 (p. 17)

of equal sized blocks, as in a normal RAID array, and the stripes are then laid out in a sequential, rotating pattern across them.

The important aspect of this layout is that related data and parity strips are still kept on separate disks, thus ensuring that the normal RAID recovery procedures can be applied. The primary advantage such a layout offers is a much higher maximum bandwidth during rebuild operations. This follows since there are more disks in the array, and not all are utilised to rebuild a given stripe. This also allows for the possibility that multiple stripes can be rebuilt in parallel. The major disadvantage is the reduced data capacity of the array compared to a traditional layout, due to the extra parity strips.

2.3 RAID Taxonomy

2.3.1 Non-Redundant arrays - RAID Level 0

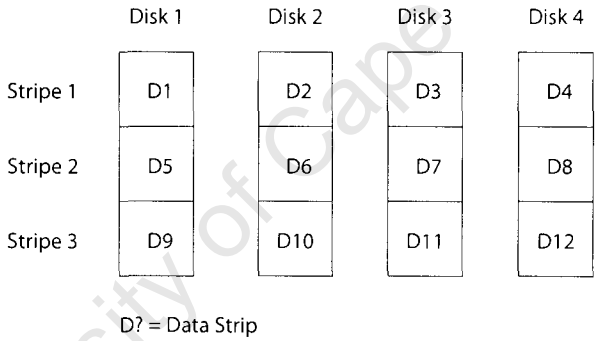


Figure 5: Data Layout in RAID 0 scheme

The scheme illustrated in Figure 5 has come to be known as RAID 0, though it is generally acknowledged that is not a true RAID. A single failure will render the data across the entire array useless, since each disk stores part of every file. It is thus necessary to store error correcting information, so that it is possible to recover from at least one disk failure.

2.3.2 Mirroring - RAID Level 1

In the case of mirroring[4], reliability is achieved by simply duplicating all data across two or more disks. This provides complete reliability with minimal repair and recovery time - in the event of a single failure, any of the duplicates can be used for reads, while writes can be mirrored across the remaining disks . This scheme offers the possibility of greater bandwidth through parallelism, since two reads of different sectors can be assigned to two different disks - both reads occur at the same time, effectively doubling the bandwidth. With more than two disks, multiple reads can be scheduled simultaneously.

Mirrored disks also suffer the smallest write penalty, since the cost of any write is simply the maximum cost for any of the individual disk writes. If the disk spindles and heads are synchronized[5],

[4]Initially referred to as disk shadowing by Britton et. al. [BG88]

[5]At any given moment, the heads of each disk are over the same logical sector

University of Cape Town

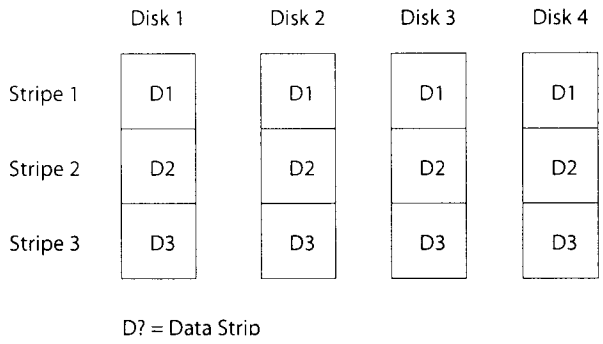


Figure 6: Data Layout in RAID 1 scheme

there is no write penalty, since all disks move in unison, acting as one large disk. Mirroring has the highest overhead of all (100%), however, since each disk in addition to the primary disk is used solely for redundancy - none of its capacity is available for useful storage.

Another issue is that recovering a failed disk involves copying the entire disk to a replacement - this is not only time consuming, it also reduces the performance of the RAID during reconstruction, which may not be acceptable in certain real-time applications. However, if more than 2 disks are used, one of the clones can simply be taken off line and used to recover the failed disk in a short period of time.

2.3.3 Striped Data with Parity - RAID Levels 2 - 4

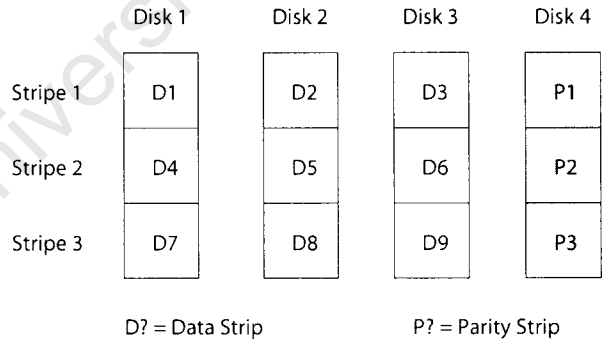


Figure 7: Data Layout in RAID 4 scheme

In order to prevent data loss in RAID systems, other than level 1, it is necessary to incorporate some sort of redundancy into the system. The simplest and most widely adopted scheme provides single error correcting parity [PGK88] using XOR operations, and is able to prevent single disk failures. This technique forms the basis of RAID 2-4.

RAID level 2 uses additional disks to store Hamming Code data, which is used to determine which disk in the array failed. However, most modern drive controllers can detect which disk in the array has failed, thus eliminating the need for these extra redundancy disks. This allows more of the total capacity to be utilised for data storage versus redundancy.

Since disk failure can now be detected by the controller, parity can be stored on a single separate

University of Cape Town

disk, and a single failed disk (parity or data) can be reconstructed from the remaining disks. This is referred to as RAID 3 in the taxonomy. The two possible failure scenarios are illustrated in Figure 8.

Data	*Parity*
0 1 0	1
1 1 1	1
1 1 0	0

Normal Data Layout

Data	*Parity*
0 1 0	?
1 1 1	?
1 1 0	?

Failed Parity Disk

Data	*Parity*
0 1 ?	1
1 1 ?	1
1 1 ?	0

Failed Data Disk

Figure 8: Illustration of the 2 possible failure scenarios

The reconstruction simply involves reading the data from all the undamaged disks for each stripe, calculating the parity of that data, and then writing this value to the replacement disk. If the failed disk was the parity disk, the recovery is done by simply recomputing the parity. If the failed disk was a data disk, the scheme still works since the XOR operator is commutative. Figure 9 illustrates this using the state represented by the Failed Data Disk scenario in Figure 8.

OldData			*Parity*		*RecoveredData*
0	1	$\oplus$	1	=	0
1	1		1		1
1	1		0		0

Figure 9: Reconstruction of lost data

RAID 4 still distributes data across disks, with a parity disk for redundancy, but now data is partitioned across the disks in strips. The new parity must be computed and written to disk to complete a write request. Hence, each write must access the parity disk before it completes. Since multiple write requests will be queueing for this single parity disk, a bottle neck is created in RAID levels 3 and 4.

University of Cape Town

2.3.4 Rotating Parity with Striped Data - RAID Level 5

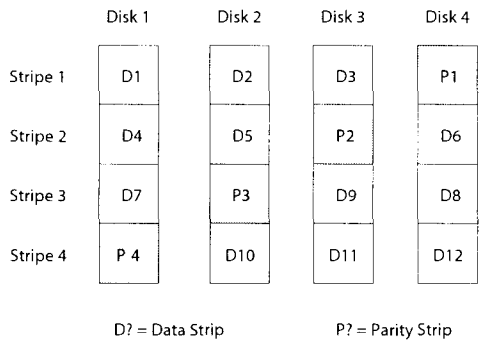


Figure 10: Data Layout in RAID 5 scheme

RAID 5 improves on RAID 4 by distributing the parity information across all disks in the array.

thus reducing the bottleneck created by a single parity disk. This scheme allows multiple simultaneous writes if the writes are to different stripes, and there are no common clusters between the writes.

RAID 5 is the most common scheme used commercially, as it offers the best balance between data integrity, storage performance and overhead (the amount of total free space that must be devoted to storing redundant parity information).

2.3.5 Dual Disk Failure Protection - RAID Level 6

All the RAID levels described thus far offer protection from only a single disk failure. These schemes work well in deployments where data integrity must be balanced against storage performance, but are in insufficient in situations where data integrity is of primary importance. RAID 6 addresses this by providing protection from up to two disk failures.

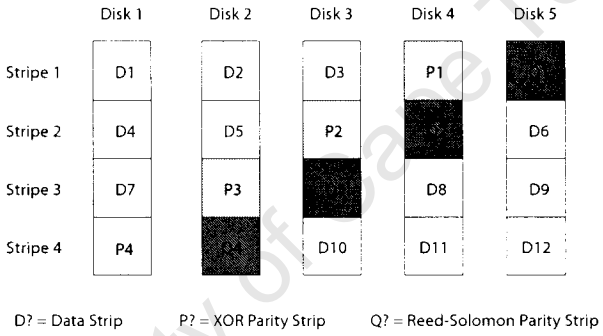


Figure 11: Data Layout in RAID 6 scheme

RAID 6 achieves this using two different, independent parity strips per stripe. The first parity strip is the same XOR parity used in RAID level 5. The second is a Reed-Solomon code across the data strips within the stripe. Like RAID 5, the parity strips are rotated to reduce bottleneck effects caused by concurrent accesses to these strips.

RAID 6 Reed Solomon Coding

The Reed Solomon Code used to provide the parity protection for the second disk in RAID 6 is the result of work by Reed and Solomon [RS60]. Their coding technique is based on the algebra of Galois fields [Anv07], specifically the $\mathbf{GF}(2^8)$ Galois field. This is a finite field with 2^8 elements[6] with several important properties:

1. Addition (+) is represented by the bitwise XOR operator.
2. Multiplication is represented by boolean polynomial multiplication modulo the irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$, which is equivalent to the operation of a linear feedback shift register.

[6]The size of the field is chosen so as to no limit the maximum number of usable drives (255), while still ensuring reasonable bounds on the computations required.

University of Cape Town

3. The normal algebraic commutative, associative and distributive laws of addition and multiplication apply.

4. Raising an element to a power is congruent mod 255.

5. Field generators, g^n, exist that generate elements of the field without repetition until all elements have been exhausted with the exception of the identity element.

Consider each strip in a given stripe as a vector of bytes, with $\mathbf{D_0}, \mathbf{D_1}, \mathbf{D_2}, \ldots, \mathbf{D_{n-1}}$ representing the data strips, $\mathbf{P}$ representing the RAID 5, XOR parity and $\mathbf{Q}$ representing the RAID 6, Reed-Solomon parity. We can then compute $\mathbf{P}$ and $\mathbf{Q}$ for n data disks as follows:

$$\mathbf{P} = \sum_{i=0}^{n} \mathbf{D}_i \tag{1}$$

$$\mathbf{Q} = \sum_{i=0}^{n} g^i.\mathbf{D}_i \tag{2}$$

where g is any generator of the field.

In the event that 1 data disk and the $\mathbf{P}$ drive are lost, the lost data can be recomputed through the equation:

$$\mathbf{D}_x = (\mathbf{Q} + \mathbf{Q}_x)/g^x \tag{3}$$

In the event of 2 data disk failures, the following set of equations can be used to recover the missing data $\mathbf{D}_x$ and $\mathbf{D}_y$

$$\mathbf{D}_y = \mathbf{P} + \mathbf{P}_{xy} + \mathbf{D}_x \tag{4}$$

$$\mathbf{D}_x = \frac{g^{-x}.(\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x}.(\mathbf{P} + \mathbf{P}_{xy})}{g^{y-x} + \mathbf{I}} \tag{5}$$

where $\mathbf{I}$ is the multiplicative identity element.

In all other cases, lost data can be recovered as in the equivalent RAID 5 case.

2.3.6 Sector Protection through Intra-Drive REdundancy (SPIDRE)

SPIDRE is a RAID protection scheme developed at the IBM Zürich Research Lab. SPIDRE addresses the real-world problem of strip-failures in disks. A strip-failure occurs when a small contiguous surface area of a disk platter becomes unreadable. This area is usually smaller than a RAID strip[7], and since the rest of the disk is still operational, it is only a single RAID stripe that is compromised. This type of burst failure is differentiated from the atomic failure of an entire disk, in that some portion of the data on the disk is recoverable.

In a RAID 5 array operating in degraded mode, such a failure would result in immediate data loss in the array. The only way to prevent this would be to utilise a RAID scheme designed to protect against more than a single disk failure, such as RAID 6. However, this introduces additional overhead, both in terms of number of disks and in storage performance. SPIDRE provides an compromise solution by recovering strip-failure related burst errors with minimal overhead.

[7]See Section 2.1 (p. 5)

University of Cape Town

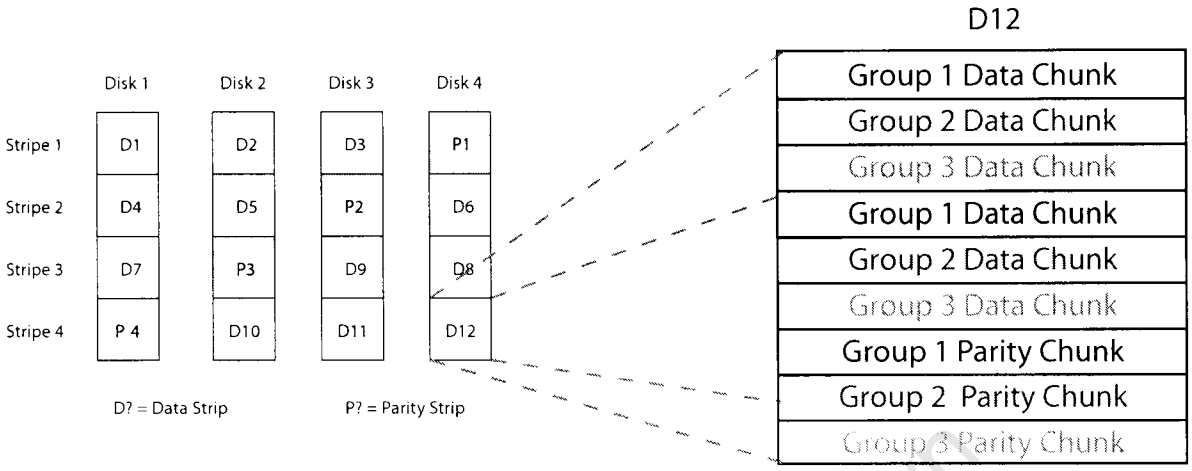


Figure 12: Illustration of the SPIDRE parity scheme applied to a RAID 5 array.

SPIDRE proposes an alternate solution to this problem. Instead of using an extra parity disk, SPIDRE adds XOR parity information to each strip in a stripe. This parity is arranged as in Figure 12 and can be used to recover lost data chunks in a strip.

The important part of this scheme is that there are several XOR protection groups (labeled as such in the diagram), that are interleaved. Without interleaving, only a single missing chunk could be recovered. as is the case with RAID 5. Interleaving allows for recovery from strip-failures that span several chunks.

In the illustrated SPIDRE configuration, the loss of any 3 consecutive chunks in a given strip (due to strip-failure) can be recovered using the same recovery algorithm used in RAID 5. The number of groups per strip, as well as the number and size of chunks are determined by examining strip-failure data from real-world disks.

The storage efficiency of SPIDRE is directly related to the number of sequential bytes that can be recovered and the size of a strip. For instance, given a strip size of 64KB and a maximal strip-failure length of 256B[8], we would require a 256 interleaved protection groups, each with $\frac{64\times1024}{256} = 256$ chunks. This would provide protection against any strip failure of less than 256 bytes within that strip, while requiring an overhead of only 256 bytes. Thus the focus on burst-failures within a strip allows the SPIDRE overhead to be minimised to exactly the number of recoverable consecutive bytes in a strip required.

2.4 Issues with RAID

Within RAID systems, there are a number of problematic areas: the physical layout of data across the array; improving reliability; recovery and repair after disk failure; design and correctness of RAID controllers and architectures; and performance of RAID architectures in specifically problematic areas. A summary of these issues, and related work in these areas, follows.

[8]This would be established from real-world data.

University of Cape Town

2.4.1 The Small Write Problem

A significant problem with RAID systems arises in their application to On-Line Transaction Processing (OLTP) systems. These systems have disk access patterns that typically consist of read-modify-write cycles. With the exception of RAID 1, this causes several problems for a RAID system. Firstly, a write in a striped array requires reads of both data and parity blocks, computation of a new parity, and writes of both new data and new parity - 4 times more accesses than for a single disk. Another problem is that these accesses are small, and hence only a few blocks within a specific stripe are altered, yet the parity disk for the entire stripe is unavailable during the update - this effectively reduces the performance of the array, by reducing the parallelism possible.

This problem was initially tackled by [SCO90], who proposed a scheme wherein writes were buffered until a sufficiently deep queue had developed to minimise the write penalty. The problem with this approach is that a disk failure could lead to data loss unless the buffers used are fault tolerant.

The work of Menon et. al. [MM92] attempts to solve this problem using a technique referred to as *Floating Data and Parity*. Each cylinder in a disk is set to contain either data or parity, and for each such cylinder, an empty track is set aside. During the update cycle, rather than overwrite the old data, the new data is instead written to the rotationally closest free block. This allows the read-compute parity-write to be executed without an extra rotational delay. The main problem with this approach is that undermines large block reads, since logically sequential data need not necessarily be stored sequentially on the disk.

The overhead required by this scheme is very low, but fault tolerant array controller storage is required to track data and parity locations. Variations on this technique are the log structure filesystem (LFS) [RO91] and the distorted mirror approach [SO91], which uses the 100% overhead of mirroring to maintain a copy of each block in both fixed and floating storage. All the above schemes require significant amounts of controller memory to handle buffering and store location information.

The most promising solution thus far arises from the work of Holland and Stodolsky et. al. [SGH93, SHG93]. They present a system wherein parity updates are buffered, then written to a log when sufficient are accumulated to allow for efficient disk transfer - data updates are written immediately. This log is then periodically purged, with all parity updates in it being written to disk. This scheme ensures data reliability, since: if a data disk fails it can be recovered from the parity and remaining data disks; if the parity or log disk fails, then the parity disk can be reconstructed from the remaining data disks, and the log disk can be emptied.

One problem with this approach is that unless the array controller has fault tolerant buffers, a failure could result in data loss. Another is that the log disk could easily become a bottleneck in the system - this can be solved by distributing the log disk across all disks, much as is done with the parity disk in RAID 5. The most problematic aspect of this approach, and one that does not appear to have been addressed, is how user response will degrade during reconstruction of a failed disk. Various schemes to address this for other RAID configurations are presented in Section 2.4.2.

A fairly recent solution presented by Haruo [YG99], *Fault-tolerant Buffering Disks*, uses disks to provide double buffering, which increases write speed, as well as backup for fault tolerance and

University of Cape Town

read speed improvements. The system does not scale well, and has a higher overhead than other systems, but for small arrays increases both throughput and response time.

2.4.2 Reliability

Due to their decreased MTTF, preventing data loss in RAID systems is a very important consideration. Redundancy is the primary fail safe, and many schemes exist to achieve this. Immediately after a failure, it is necessary to perform some form of recovery. Finally, the failed disk needs to be replaced and reconstructed to restore the system to full working efficiency. This section covers advances in these areas.

Redundancy Schemes

In order to prevent data loss in RAID systems, other than level 1, it is necessary to incorporate some sort of redundancy into the system. The simplest, and most widely adopted, system uses single error correcting parity [PGK88], using XOR operations, and is able to prevent single disk failures.

EVENODD [BBBM94] is an alternate scheme that prevents two disk failures, and is efficiently implementable in hardware. The layout described to prevent bottlenecks restricts the array to a maximum of 259 disks, however.

Other schemes include balanced incomplete block designs (BIBD) [HG92], which attempt to uniformly distribute data and parity across a disk, and coding methods proposed by Gibson [HGK+94] which protect against arbitrary numbers of failures, but have overheads that increase exponentially w.r.t. prevented failures. Additionally, the schemes are fairly restrictive on array dimensions for optimal redundancy usage.

Finally, a scheme proposed by Alvarez [ABC97], DATUM, allows for recovery from an arbitrary number of disk failures, using an optimal amount of redundant storage, whilst still allowing flexible array configurations, and ensuring both parity and data are evenly distributed across the disks.

Recovery

Recovery is necessary immediately after a disk failure, to ensure that operations in progress at that time are not lost completely. There are currently three approaches to this problem. The first and most prevalent solution, forward error correction, attempts to handle errors dependent on the current state of the system and the state of execution of the requested disk operation. This method requires enumeration of all possible states of the system, and handcrafting execution paths for each. This is both time-consuming and error prone.

An alternate approach, backward error recovery [CG94], uses an approach popular in transactional systems which are required to support atomicity of operations. The approach is to set out a small number of execution paths for each of the possible states of the system (error-free, disk failed, etc.). Each execution path is composed from a set of simple, reversible operations, which are logged as they execute. When a failure occurs during execution of one of these paths, the execution is rolled-back by executing the inverse operations in reverse order. When the original state is

University of Cape Town

recovered, an alternate execution path, appropriate to the current state of the system, is used to retry the failed operation.

The final alternative, roll-away error recovery [Cou97], uses a similar scheme to backward error recovery, but adds commit barriers to simulate two-phase commit in transactional systems. The idea is that when a failure occurs, execution is allowed to continue up to a commit barrier, but not beyond. If the completed state is not reached by this, then an alternate execution path is chosen, and the operation is reattempted. This style of error recovery is popularised by the RAIDframe system [CGHZ96a], which is discussed in Section 4.2.1 (p. 42).

Reconstruction

A major consideration when recovering from a failed disk is the effect on user response times, i.e. does the reconstruction of the failed disk degrade the performance of the RAID? In the usual case this is very true, since reconstructing any single disk requires access to all the other disks to reconstruct every stripe, effectively rendering the RAID inaccessible during each such access. While stripe reconstruction can be interleaved with user requests to allow the RAID to continue operation, access latency will rise unacceptably and the Mean Time to Repair (MTTR) will increase. The longer it takes to repair the RAID, the more likely it is a second disk will fail before the first is recovered, resulting in data loss.

An innovative solution to this is presented in [HG92, HGS93, HGS94, Hol94]. The authors suggest that performance degradation during reconstruction can be reduced by sacrificing some of the data capacity of the RAID toward redundancy. The crux of their work is the idea of a virtual topology that is mapped to the physical disk topology of the system. Assuming that there are 7 disks available, the normal, intuitive solution would be to treat all 7 as a large array with striping across all 7, and distributed parity. An alternative would be to treat this as an array of 4 disks, in a RAID 5 configuration. The authors discuss ways of mapping such a virtual topology to the physical one[9] to allow this behaviour.

The benefit of this approach is that if a single disk fails, only particular stripes in the virtual array will be affected. Importantly, since each stripe only consists of 4 virtual disks, reconstructing affected stripes only requires read accesses to 3 other real disks. This greatly diminishes the bandwidth required for reconstruction, and allows accesses to the other 3 disks[10] to occur simultaneously. Additionally, the mapping scheme mentioned above ensures that the virtual sectors are uniformly distributed, thus ensuring that no single disk gets overburdened during reconstruction. The only requirement is increased redundancy overhead, since instead of a ratio of 6:1 of data to parity sectors, the ratio is now 3:1.

The latest development in this field has been *data-reconstruction networks* [Yok00]. The idea is that disks are connected in subnetworks which are then interconnected to form one large network. Reconstruction of a single failed disk is localised to a subnetwork, hence reducing the impact on the whole network. Additionally, the overlapping of subnetworks allows the recovery of more than 1 failed disk, dependent on the architecture.

Other considerations during reconstruction are how to handle user requests. If a read is requested of an as yet unreconstructed sector, the sector can recovered on the fly using parity and the

[9] See Section 2.2.4 (p. 8)

[10] Remember that 1 disk has failed, and is in the process of being reconstructed

University of Cape Town

remaining data disks. However, once this data is calculated, the option arises to store it (which requires buffering), write it to disk (which could upset the scheduling algorithm in use) or discard it (which entails having to recalculate it later during the reconstruction). The choice of which approach to take is dependent on the implementation.

If a write is requested during reconstruction, then this new value is simply written to the replacement disk, and the appropriate stripe is excluded from the rebuild list, which reduces reconstruction time. Alternatively, all stripes are reconstructed in order, and new data is simply overwritten - this has the advantage of significantly less bookkeeping, but there is a lot of unnecessary work performed.

2.5 RAID Caching

Due to the relatively slow access and transfer times of secondary storage systems in general, caching plays an important role in improving the performance of these systems. The most important function of a cache is to store recently accessed data in memory. Subsequent requests for this data can then be served from memory much faster than accessing it on disk.

As the cache is finite in size, its contents must be managed so as to ensure sufficient space for new additions whilst still retaining data which is most likely to be accessed in future. The various techniques for achieving this are discussed in Section 2.5.1 (p. 18).

The cache can also perform a predictive function, by examining the list of recently accessed data and requesting adjacent data from disk to store in memory. This predictive behaviour leverages the spatial locality of most workloads, where data is generally accessed in physically adjacent groups.

As in all connected systems with a speed disparity, the cache can also act as a buffer to mask the speed differential. This can be accomplished by allocating a section of the cache to buffer data that is being sent faster than it can be processed by the RAID system. This section can also be used to hold intermediate results until a sufficiently large amount has been accumulated to efficiently send to the host in one transmission.

University of Cape Town

2.5.1 Caching Schemes

A cache is finite in size and divided into a number of equal-sized, contiguous sections, referred to as *pages*. These pages are atomic and form the basis of all cache operations.

When a cache is initialised, it is empty and all pages are available to hold data. As data is added to the cache, the number of free pages decreases until the cache is full and no pages are available. From this point on, whenever new data is added to the cache, it is first necessary to free one or more pages that contain data before the new data can be successfully added.

The process of freeing pages is referred to as destaging. If a page to be destaged has been marked as *dirty*[11], its value is first written to disk. The page's entry in the cache page table (which holds a list of all pages in the cache) is then modified to reflect that it is free and available to store

[11] See Section 2.5.2 (p. 21)

data. Note that the data in the page is not actually deleted, as the value stored therein will be overwritten when the page is next used.

The method used to determine which cache page to free when the cache is full is referred to as the *cache policy*. There are a variety of cache policies that are tailored to specific uses, but all derive to some extent from either the LRU or LFU policies, which are discussed below. We also present the ARC policy, which attempts to synthesise the best aspects of LRU and LFU to provide a very good, general purpose policy. For a detailed overview of some of the operational details of these schemes, refer to Section 8.4 (p. 111).

Optimal Page Replacement Policy

The optimum efficiency criteria for a cache is the rate at which pages are recycled. In other words, the more infrequent the need to destage a cache page, the better the policy is performing. This arises from the desire to minimise the number of slow disk accesses necessary, which are incurred whenever a page of data must be brought into the cache from disk.

Early research into page replacement algorithms that maximised this criteria, particularly the First-In-First-Out (FIFO) algorithm[12] led to the discovery of **Belady's anomaly**: For certain page replacement algorithms, increasing the cache size could increase the page recycle rate.

This led to the definition of an optimum cache replacement policy (OPT) [SGG02], which is characterised by the following algorithm:

> Replace the page that will not be used
> for the longest period of time.

Clearly, this algorithm is not practically implementable since it requires knowledge of future workload behaviour. It does, however, provide a baseline against which other policies can be compared.

LRU

The Least Recently Used (LRU) policy is an attempt to approximate the OPT policy by using the events of the *recent past* to predict events in the *near future*. Under LRU, each page has associated with it the time it was last *accessed*. An access here is defined as a read or write request that alters the contents of that page without requiring it to be destaged.

When a page must be chosen for replacement, the LRU policy picks the cache page with the *earliest* access time, i.e. the page that has not been used for the longest period of time. This scheme then works under the presumption that a page that has not been used for a long time will not be required in the near future.

One of the problems with the LRU scheme is that the possibility of thrashing exists. Thrashing occurs when a page that has just been destaged is then requested and has to be read back into the cache. It occurs with workloads that repeatedly access a working set of data that is just

[12] Where the first page added to the cache is the first page freed.

University of Cape Town

larger than the cache size. In this case, LRU is very inefficient as the page recycling rate increases greatly.

LRU is also susceptible to cache pollution by sequential workloads. In this scenario, the sequential workload causes all pages in the cache to be destaged and replaced by the sequential data. As long as the sequential stream lasts, existing pages (that should potentially remain in the cache) are destaged on each request. In addition, once the sequential stream transitions to some other access pattern, the cache is left devoid of any useful recency information with which to determine policy behaviour.

LFU

The Least Frequently Used (LFU) policy takes a different approach to LRU in that it considers recent behaviour rather than recent history. This utilises the assumption that if a page has been used frequently in the past, it is likely to be used again even if it has not been used recently.

Under LFU, each page has associated with it a counter of the number of times it has been accessed since being read in from disk. When a page must be chosen for replacement, the LFU policy chooses the page with the smallest frequency count.

A major weakness in LFU is also related to the occurrence of thrashing, though under different circumstances. A workload that comprises a large number of frequently accessed pages, together with an equal number of random requests will experience cache starvation as the cache becomes filled with the frequently accessed pages.

In this situation, none of the frequently accessed pages are eligible for destaging, and so the recently used, random accesses must be continuously swapped into and out of a single cache page. Once again this is not ideal, as the page recycling rate increases here too.

ARC

The Adaptive Replacement Cache (ARC) policy was developed by IBM [MM03b]. It attempts to strike a balance between the characteristics of LRU and LFU by dynamically adapting to the workload presented.

An ARC cache uses a page table that is twice the size of the actual number of pages in the cache. This page table is split into two equal sized lists: `L1` which contains those pages that have been accessed exactly once; and `L2` which contains pages that have been accessed more than once.

`L1` and `L2` are further divided into `T1`, `B1` and `T2`, `B2` respectively. `T1` and `T2` contain information related to the actual pages in the cache, and as such the total size of `T1` + `T2` is always equal to the cache size. `B1` and `B2` are used to record historical data, and do not reference actual cache pages. ARC also uses a target size `target_T1`, which represents the optimum size of `L1` and is updated constantly during execution.

The replacement policy for ARC uses the LRU criteria:

> Replace the LRU page in `T1`, if `T1` contains at least `target_T1` pages;
> otherwise, replace the LRU page in `T2`.

University of Cape Town

The adaptive nature of cache is achieved by varying `target_T1` in response to the observer workload. The adaptation rule, which is applied whenever a new page is added (ie. an actual cache miss), is:

> Increase `target_T1`, if the new page exists in the history B1;
> similarly, decrease `target_T1`, if the new page exists in the history B2.

By applying these two rules to the cache, ARC is able to adapt to changing workloads and hence avoids the pitfalls inherent in both LRU and LFU. It is scan-tolerant, in that sequential workloads do not pollute the cache, and it is less susceptible to thrashing as the `target_T1` metric attempts to ensure there is enough space for both frequently and recently used pages. For a more in-depth explanation of ARC, refer to [MM03b, MM03a].

2.5.2 Write Caching

Across all caching schemes a distinction can be drawn between Write-Back and Write-Through caching. Both these terms refer to the situation where a write operation is requested and the data to be written is sent by the host to the RAID Controller.

In a Write-Back cache, this data is added to the cache and the relevant pages flagged as dirty. A signal is immediately sent to the host indicating the success of the requested write. At some later time, usually when pages are flushed from the cache, the updated data is actually written to disk. This scheme has the advantage that multiple updates to the same data will result in a single disk access that reflects only the last, correct value.

By reducing the overall number of disk accesses required, this increases performance in the system, but at the cost of reliability. Unless the cache is implemented in non-volatile RAM[13] (NVRAM), any data not written to disk will be lost in the case of a power failure. This scheme also proves problematic when used in multi-level RAID schemes[14] as cache coherency cannot be guaranteed.

In a Write-Through cache, the data is added to the cache and immediately written to disk. Only once the disk request has succeeded is the host signalled to indicate a success. Since this scheme requires more disk accesses that Write-Through caching, it performs poorly by comparison. It does have the advantage of offering much better reliability across power failures, as writes are always persisted immediately to disk.

University of Cape Town

[13] RAM that can persist its contents across power failures, such as NAND flash RAM.
[14] See Section 5.4.4 (p. 68).

Chapter 3

Simulation of Systems

University of Cape Town

In this chapter, we present an overview of the discipline of simulation. We introduce the discipline, and evaluate both it benefits and shortcomings. We further present the fundamental aspects of simulation and outline the principles of conducting a simulation study. Finally, we discuss the use of simulation environments, present a number of available simulation environments, and justify the choice of one for the development of our simulator.

3.1 Introduction

In areas such as operations research and management science, simulation is a tool that allows business processes to be analysed and optimised without disrupting the daily activities of the business. In engineering disciplines, simulation allows products to be developed and tested before investing in the manufacturing process. In scientific disciplines, theoretical models can be simulated to determine their validity and future behaviour.

A simulation attempts to imitate the behaviour or operation of some process for the purpose of studying some aspect of it. The process being simulated is referred to as a system, and in order to correctly mimic its operation it is necessary to develop a model of this system. This model usually takes the form of mathematical relationships, which calculate observable outputs based on input parameters, and logical relationships, which determine which portions of the model are exercised at a particular point in time [LK82].

In order to develop this model of the system, it is necessary to make a set of assumptions regarding the operation of the system. These assumptions may be based on a theoretical idea of how the system should work, or may be derived from an empirical investigation of the system. These assumptions seek to approximate the system, which is usually too complex to model in its entirety, by concentrating on areas of particular interest or importance. The results produced by a simulation are thus only estimates of the true behaviour of the system.

For certain models that are sufficiently simple, it is possible to determine an analytical solution for the system model. This solution is obtained by using mathematical methods[1] to obtain exact results from the system model. This is obviously a significant improvement over the estimated

[1] Such as algebra, calculus or probability theory

results produced by simulation, but such analytic solutions are usually not possible for realistic system models, due to associated complexity.

For most other models, simulation is necessary to obtain information from a system model. Simulation, in our context, involves using a computer to numerically evaluate a given model over a stipulated time period. This evaluation involves a number of steps, and at each step data is gathered that is later used to estimate the desired characteristics of the model[LK82]. These estimates can then be used to refine the model, change the simulation parameters or implement changes in the corresponding real-world system.

3.2 Evaluating the Benefits of Simulation

Simulation and analytic solutions are the most common approaches to analysing a given system model. Each has their own advantages and drawbacks, as outlined by Law and Kelton [LK82]. Given our focus on simulation of systems, we present first the advantages that simulation of system models provide:

1. Most mathematical models of real-world systems are too complex to evaluate analytically, thus simulation is often the only viable option.
2. Simulating an existing system allows hypothetical operating conditions to be applied and the performance of the system to be estimated.
3. Simulation is best suited to evaluating alternative designs of a single system (or alternative operating parameters for a single system) against a given set of requirements to determine the most suitable.
4. It is possible to maintain much tighter controls over experimental conditions while conducting a simulation of a system when compared to experimentation with the system itself.
5. Simulation allows us to manipulate time within an experiment. Thus we can compress time for long simulations, allowing for results to be obtained sooner. Conversely, it is possible to dilate time for processes that occur over a very short time, allowing a detailed study of their operation in expanded time.

Simulation is not a magic cure, however, and there are a number of drawbacks that must be carefully considered. In many situations, an approximated analytic solution of a simplified model may be of more use, simpler to construct and less expensive than the equivalent complex simulation. The considerations against simulation are:

1. Simulation models can be time-consuming to develop, and complex simulations require large amounts of computation time and power, which can prove expensive.
2. A simulation that uses stochastic models[2] produces only estimates of the true behaviour of a system, whereas an analytic solution can produce exact results. It thus stands to reason that if an analytic solution is possible, it would be preferable to simulation. However, analytic solutions are not always possible.

[2]See Section 3.3.1 (p. 24)

University of Cape Town

3. Stochastic models that produce estimates of a model's characteristics require several simulation runs to be executed for each set of input parameters, to allow statistical analysis of the results. This requirement means that simulations are better suited to comparing a finite set of alternatives, rather than optimising over a range of parameter values.

3.3 Simulation Fundamentals

3.3.1 Systems and Models

A system is defined by Schmidt and Taylor [ST70] to be a collection of entities (components), e.g. people or machines, which act and interact toward the accomplishment of some goal. A simulation of a system requires a model of the system to be developed. This model describes the functions of the various components of the system, as well as the interaction between these components. The granularity of this model is highly dependent on the particular objectives of the study, as well as the accuracy of results required. For instance, a simulation of a commodity computer hard disk drive would model the various components of the drive, such as the drive arm, the platters and the read/write head. However, in a simulation of a storage system, the disk drive would form one small component within the overall model.

A model of a system usually involves a number of variables whose values changes over time. These variables are referred to as state variables, and the state of the system is then defined to be the values of all state variables at a given time. In the simulation of a storage system, examples of such variables would be: the number of I/O requests being processed; the time of arrival of each request; and the collection of data stored in the cache.

Continuous and Discrete Systems

Depending on the areas of interest and the model employed, a system may be classified as either continuous or discrete in nature. For a continuous system, the state variables change continuously with respect to time. In a discrete system, the state variables change only at a finite number of points in time.

For example, a model of storage systems that is interested in investigating the Quality of Service of the system will be continuous, as variables such as the throughput, bandwidth utilisation and failed requests change continuously. Conversely, a model that focuses on the actual requests in the system and how they are processed is discrete in nature, since each individual request is simulated atomically. Hence, the state variables only change when commands are executed to satisfy each request.

Deterministic and Stochastic Simulations

A further distinction can be made between deterministic and stochastic models. A deterministic model given a set of initial conditions will always produce the same resultant state at a given instant in time. A deterministic model contains no random variables, hence the results it produces are always the same.

University of Cape Town

For the same set of initial conditions, a stochastic model will produce different states for the same time instant dependent on the random number seeds chosen[3]. Since a stochastic model contains one or more random variables, the output data it produces are also random and serve only as an estimate of the true behaviour of the system. Consequently, the results of a stochastic simulation require more interpretation than those for a deterministic one.

The simulation models we present in this work are discrete and stochastic. They model the processing of individual I/O requests, over a period of time, using random variables.

3.3.2 Advancing Simulation Time

The state of a dynamic discrete simulation model changes over time, and hence the state variables must be periodically updated at specific times. The concept of simulation time must thus be introduced to determine when to update individual variables. This simulation time is stored in an independent variable, and must be updated over the course of the simulation. The units in which simulation time is measured is determined by the simulation environment. There is also usually no relationship between the length of the simulation time required to complete a simulation (the period of time under investigation) and the actual time required to run the simulation to completion on a computer system (the execution time).

There are two main approaches to advancing simulation time: *fixed-increment time advance* and *event-driven time advance*. Each method is responsible for incrementing the value of the simulation time by a specific amount, which in turn determines when and how simulation variables are updated.

Fixed-Increment Time Advance

In this approach, a fixed time interval Δt is chosen at the start of the simulation. This interval represents the value by which the simulation time will be advanced after each update. After each time increment, the simulation must check whether any events were scheduled to occur during the elapsed interval of length Δt. All such events are then assumed to have occurred at the end of the interval, and the routines associated with these events are executed as though this were the case. This scheme is illustrated in Figure 13.

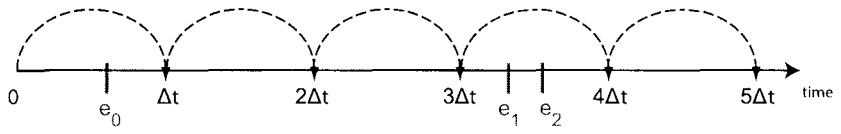


Figure 13: In the Fixed-Increment time advance approach, events are only processed at fixed increments. Events occurring between increments must be moved to an appropriate boundary. In the figure above, both events e_1 and e_2 will be processed as though they had originated at time $4\Delta t$. (Figure adapted from [LK82])

The requirement that events be processed as though they had occurred at the end of an interval has a number of disadvantages. Multiple events that occur in a given interval are treated as

[3]See Section 3.3.6 (p. 31)

University of Cape Town

having occurred at the same simulation time, which ignores the actual ordering of events within the interval. This requires that the simulator must have some method of determining in which order to execute these apparently simultaneous events. Calculations that are dependent on the time an event occurred will hence be incorrect.

The interdependence between events may also be disrupted by these simultaneous occurrences. For example, a computation associated with event A may rely on the elapsed time since the last occurrence of event B. If both A and B occur in the same interval, but at different times, the computation for A may be incorrect.

The above problems may be mitigated to some extent by reducing the size of Δt, but this increases the number of interval advances required for the simulation to complete. This in turn requires more checks for event occurrences, which finally increases execution time.

Due to these problems, fixed-increment time advance is not an appropriate choice for simulations where the inter-arrival time between events is not known, or varies greatly. This scheme is, however, well suited to scenarios where events occur on a regular schedule. Financial models where the simulation state is updated every year is a natural application for fixed-increment time advance, as are weather simulations and astrophysical simulations.

University of Cape Town

Event-Driven Time Advance

Event-driven time advance is a scheme that attempts to address the shortcomings evident in fixed-increment time advance. With this approach, the concept of an interval is scrapped. Instead, all events generated by the simulation are appended to a queue, commonly known as the *Future Event Set* (FES) or *Future Event List*, which is sorted by time of occurrence. Whenever the simulation enters an idle state (no event is being processed), the FES is searched for the *most imminent* event. The simulation time is then advanced to the time this event occurs, and the event is processed. This processing may lead to other events being generated, and these are added to the FES. When all relevant processing is done, the simulation then enters another idle state, the FES is searched for a new event, and the loop repeats. This scheme is illustrated in Figure 14.

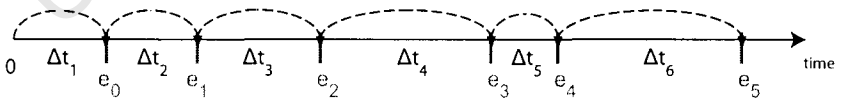


Figure 14: In the Event-Driven time advance approach, events are processed at the time they are generated. Thus the simulation time is advanced by differing amounts depending on the elapsed time between events (the Δt_i's above). (Figure adapted from [LK82])

Simulations using event-driven time advance have an implicit stop condition that is triggered when the simulation is in an idle state (no events are being processed) and the FES is empty (there are no future events to be processed). This is in contrast to the fixed-increment scheme, which cannot detect whether requests will be issued in future, and hence must have an explicit stop condition built into the simulation.

Another advantage of the event-driven approach is that periods of inactivity in the system are not simulated, since the time is only advanced to points where processing must be performed (when an event has occurred). This reduces the simulation overhead, and hence execution time. Further,

since all events are processed as they occur, the inter-dependence between events is preserved, in contrast to the case with fixed-increment time advance.

3.3.3 Workloads

Once a model of the system of interest has been created, it is necessary to consider the workload associated with the system. The workload refers to the series of inputs that are passed to the system, and to a large extent determines the results that a given simulation run will produce. Typical types of workloads that are processed by RAID systems include:

- On-Line Transaction Processing (OLTP) workloads: these workloads consist of many small requests to random areas on disk, and generally involve a read-process-write pattern applied to this data. This workload is typical of financial systems where large numbers of clients are making small changes to their accounts.
- Web Server workloads: these workloads typically consist of medium sized read-only requests to random areas on disk, as web pages are delivered to clients.
- Batch Processing workloads: these workloads consist of large, sequential data accesses that are primarily read requests, since they primarily relate to the processing of large volumes of data.

An appropriate analogy for the relationship between system models and workloads is the typical computer system: the system model represents the hardware and the workload the software. Each is defined separately, but both are required to achieve the required output. Further, while the model changes infrequently, the workload is routinely altered in order to investigate varying aspects of the system. Thus in any simulation study, determining the characteristics of the required workload is of paramount importance.

A number of methods are available to help determine possible workloads for a given system. If the system under investigation is a real-world system, measurements of the current load on the system can be made and used to extrapolate similar workloads. Historical data captured over a period of time can also be used to help determine appropriate workloads. Alternatively, a theoretical model of the workload can be built and then used to exercise the system model. This last option is particularly useful for abstract, theoretical systems where no data on usage is available.

For the particular case of storage systems, both methods are equally feasible. The workload for such a system consists of a set of Input/Ouput (I/O) requests that are submitted to the system. These requests are for particular segments of data stored within the system that are to be fetched (read) or stored (written). Each request is defined by a number of parameters, details of which can be found in Section 5.5.1 (p. 70). A series of requests, each with appropriately configured parameters, determines the workload presented to a storage system.

Since most storage systems of interest exist in the real-world, we have considered two forms of workloads to use with our system models. The first uses traces obtained from real systems. A trace is a record of all the I/O requests, and their parameters, submitted to a system over a period of time. Such a trace may be used directly to exercise a system model that corresponds exactly with the original system in terms of configuration. More usually, the system model differs from the original system and the trace must be manipulated before being used.

University of Cape Town

This manipulation could entail altering the parameters so that they map on to the system model. For instance, if the collected trace was from a storage system with a small capacity, increasing all requested data sizes by a factor of x would allow the trace to be used with a system model configured with a capacity x times larger. Alternatively, the times between requests could be expanded or contracted by a constant factor, thus allowing a correspondingly longer or shorter period of time to be simulated.

Such manipulations of traces should be carried out with caution, since altering a trace might significantly change the nature of the associated results. In particular, drawing conclusions from these results based on the nature of the original trace may be incorrect. Trace manipulations must thus be carefully considered and analysed to ensure the underlying behaviour represented in the trace is preserved.

The second form of workload we have considered is the synthetic workload. A synthetic workload uses a theoretical model to determine the parameters of the individual requests in the workload. Each parameter is represented by a random variable whose value is drawn from some predetermined statistical distribution, such as those presented in Table 6 (p. 138). These distributions are themselves configured by the choice of a number of parameters.

Determining which distributions to use and how to configure them is an important consideration before using any synthetic workload. These choices can be determined by modelling the characteristics of a series of traces, as is done in the ESSWA project [Sik06]. Alternatively, a particular set of distributions may be chosen which artificially stress some component of the system, or exercise a particular subset of the available operations. For instance, a workload that consists of only write requests may not be common in practice but may be necessary in order to evaluate the performance of various RAID schemes during write operations.

However the workload is determined, it must be remembered that it is only the combination of an appropriate workload with a given system model that can provide meaningful simulation results. The simulation software we have developed allows for both the above workload options to be utilised, but only the user can ensure that a representative workload is used.

University of Cape Town

3.3.4 Simulation Execution

The process of discrete, event-driven simulation can be generalised to apply to simulation of any system model [LK82]. Almost all such simulations have a common set of core units, and a common flow-of-control between these units. The organisation of these units promotes a structured, partitioned approach to developing, debugging and extending simulation code. The functions of each of these units are:

- **System state:** Refers to all the state variables that are used to describe the system at a given time. Depending on the architecture used these variables may be collected in a single location or distributed amongst various code objects.
- **Future Event Set (FES):** This is a collection of all the events that are scheduled to occur during the execution of the simulation. These events are sorted by the simulation time they are scheduled to occur at.

- **Statistical counters:** Encompasses all variables that are used to record statistical information about the simulation. These variables are updated as appropriate after each timestep.
- **Simulation clock:** The variable or object responsible for maintaining the current simulated time within the system.
- **Initialisation:** This encompasses all code executed prior to the start of the simulation. The *system state* and *future event set* are both initialised and the simulation clock is set to 0.
- **Simulation loop:** This is the main loop of the simulation program and is responsible for controlling the operation of the simulation. The simulation core calls the *timing routine* to update the simulation time and retrieve the next event to be processed. It then calls the appropriate event routine to handle the pending event. Once the event has been processed, the simulation terminating conditions are checked to determine whether to end the simulation or continue processing events.
- **Timing:** Removes the next event from the *future event set*, advances the simulation clock to the time at which this event should occur and returns the event to the *simulation core*.
- **Event handling:** Performs the appropriate processing and update of system state variables and statistical counters for a given event. Generates future events and adds them to the FES. In object-oriented simulations, each object in the simulation has such a routine for processing particular types of events.
- **Statistical Output:** A post-simulation subroutine that outputs the values of all statistical counters for later analysis.

This basic structure applies to most discrete, event-driven simulations, however individual implementations may merge or subdivide units based on their requirements. In particular, modern simulations tend to be object-orientated and distributed in design. This increases the flexibility of the simulation in terms of extensions and changes, but it also leads to structures that are notably different from the one we have presented. However, the functionality described and the interactions between the functions still remain broadly similar.

University of Cape Town

3.3.5 Events versus Messages

The traditional approach to simulation leans heavily on the concept of events. An event is any occurrence within the system which requires a series of actions to be performed. These actions can be any combination of: processing of new data; updating the simulation state; updating the statistical counters; or generating a further series of events. Events are usually described in terms of type, and generally do not have information associated with them. For instance an event indicating the arrival of a new I/O request to a storage system must be handled by an appropriate process to determine the specifics of the request.

Such a system works well for simulations where the entire system state is accessible by all event routines. However, modern simulations distribute functionality across multiple objects rather than having a number of independent processes, and the internal state of these objects are often hidden. In such a scenario, the event model is a poor fit to the architecture. Additionally, features

such as polymorphism mean that the simulation loop may not be aware of all the events that a particular object can handle.

The solution to this is the concept of messages. A message is similar to an event in that it indicates a system occurrence that requires processing. However, messages are passed between objects rather than being raised globally. A message also has an associated state that is used to pass information between objects in the simulation. This change has a number of consequences.

Firstly, events are categorised and distinguished solely by type. This means that subcategories of events cannot be differentiated unless each is represented by a new event type. By example, consider an event indicating a I/O request entering the system. It would be necessary to create two new event types for each possible request, in order to differentiate between read and write requests. Messages overcome this problem by allowing the internal state of the message to be examined to determine the appropriate action. Thus, an IORequest message might contain a field to indicate whether it is a write or read request, rather than requiring two different message types.

In an event based system, when an event is triggered the simulation must decide how to appropriately handle it. This is usually solved by having each process in the simulation register handlers for the events it is interested in. This requires that the simulation software must keep track of the relation between events and the corresponding registered processes, as well as deciding which of the registered processes for an event should be invoked. By contrast, messages are a directed communication between two objects. This means that when a message is dispatched, the recipient and delivery time is known, and there is no ambiguity in how to handle it.

Another consideration stems from an event's lack of state information. This requires that the individual processes in the simulation need to maintain a notion of state[4]. This in turn lends itself to utilising a looping execution strategy, wherein the process is constantly alternating between waiting for events and handling them. The necessity of state information increases the memory requirements for a simulation, and the looping strategy requires an explicit terminating condition to exit the loop. A side effect of this looping strategy is that a mechanism to allow the passage of simulation time must exist, since the processes are always active. Such a mechanism means that an event could occur at any point during the loop. Accounting for this complicates the expression of functional logic within the loop.

A messaging system, by contrast, is able to avoid this problem, since messages themselves contain state information. Thus the objects in the simulation do not need to maintain state information. Further, objects can be developed using a simpler message-handling approach, where a specific method is called to handle an incoming message. Once the message has been dealt with, the method exits. When the handling of a message requires time to be advanced – a delay in satisfying an I/O Request, for example – a message can simply be scheduled for delivery at a later time. This approach ensures that objects in a message-based simulation are only activated when a message arrives. Additionally, since messages are handled instantaneously – there is no time advance mechanism as in an event-loop system – a new message cannot arrive whilst handling an old one. This allows the logic for handling a message to be more clearly expressed, since detecting other message arrivals is not necessary.

The messaging paradigm is also a good representation of directed communication between various

[4]The alternative is to have a global state that is manipulated by all processes but this is a poor, inextensible solution comparable to the use of global variables in software.

University of Cape Town

components of the system. It is also well-suited to illustrating the implicit logic at work, as the flow of messages between objects and the types of these messages can be used to describe the operation of the system. The event system, on the other hand, is simpler to implement and use in small simulations as the messaging infrastructure does not need to be constructed and managed.

Finally, messaging allows for abstract, black-box type development of component objects. This is possible by treating the set of messages that an object consumes, and the associated set of messages it generates as a contract between the object and other objects it communicates with. Two objects that both satisfy the same contract can then be interchanged without requiring changes to other objects they communicate with.

3.3.6 Random-Number Generation

A fundamental aspect of any stochastic simulation is the use of random numbers drawn from specified distributions. In practice this involves generating a series of numerical values for a random variable, such that the frequency of each of these values satisfies the Probability Density Function (PDF) of the associated distribution. A series of such values is referred to as a random-number stream.

A random-number generator is software that generates such a stream. When developing simulations, it is preferable to use generators which possess a particular set of characteristics. Most importantly, the values that are generated should conform to the chosen statistical distribution and be independent of each other (statistically uncorrelated). In terms of implementation in software, it is important that generators execute quickly (so as not to create a bottleneck) and utilise a minimum of storage space.

It is also important that the generator is able to reproduce a given random-number series. This requirement is useful for reproducing error conditions for debugging purposes and verifying the operation of the simulation. It also allows the same series of random input data to be used in several simulation runs, either to increase the precision of results or to compare different configurations.

Generating true random numbers that conform to these requirements is prohibitively difficult, in large part due to the deterministic nature of computers. The accepted compromise is to use pseudo-random generators. These are arithmetic based generators where the entire random number stream is determined by a given *seed value*. The numbers generated are thus not truly random, but they are sufficiently uncorrelated to pass a series of statistical tests that check for randomness.

All such pseudo-random generators share one noteworthy problem. After a given number of values have been generated in a series, the series begins repeating. The number of values required for this to occur is referred to as the period of the generator. The quality of a generator is primarily determined by this period, since a small period implies a small number of usable random values.

Most random-number generators generate streams for the Uniform Distribution over the interval $(0, 1)$, referred to as U(0,1). However, it is more usual to require the use of other statistical distributions, such as Exponential, Geometric or Poisson. There exist a number of methods to use the random-number stream from a U(0,1) generator to generate a value for a random variable

University of Cape Town

from another distribution[5]. Most notable are the:

- **Inverse-Transform** method, which uses the inverse of the required distribution function.
- **Composition** method, which describes the required distribution function (F) as a function of a number of other distribution functions $(F_1, \ldots, F_n)$, from which the required random variable can be more easily sampled.
- **Convolution** method, where the desired random variable (X) is described as a sum of other independent identically distributed (IID) random variables which can be more easily generated than X.
- **Acceptance-Rejection** method, where a number of possible values are generated until one satisfies a given condition.

The most commonly used random-number generators are *linear congruential generators* (LCGs). LCGs were introduced by Lehmer [Leh51] in 1951. LCGs are defined by the recursive formula:

$$Z_i = (aZ_{i-1} + c) \,(\text{mod } m) \tag{6}$$

where the Z_i's represent the consecutive values in the series. The choice of the various parameters a, c and m determines the quality of the generator and significant research has been conducted to find such values so as to maximise the period of the LCG. Other notable generators are: the Mersenne Twister by M. Matsumoto and T. Nishimura [MN98b], which has a period of $2^{19937} - 1$; and the CMRG by L'Ecuyer [PLK02], which has a period of about 2^{191} and can provide a large number of streams that are guaranteed independent.

3.4 Developing a Simulation Study

Law [LK82] presents a series of steps that are typical of the development of any simulation study. These steps are reproduced below:

1. **Formulate the problem**: Determine the objectives of the study, describe the scope of the alternate system designs to be studied and the criteria that will be used for evaluation.
2. **Collect data and define a model**: Determine the specifics of the system under consideration, including details of its operation, input parameters and output dependencies. Use this information to create a representative system model that can be simulated.
3. **Validate the model**: Compare the model to real instances of the system and consult people involved in using the system to ensure the validity of this system model.
4. **Construct a simulation**: Utilise an appropriate simulation library to develop a running simulation of the above system model. Conduct testing and debugging to ensure the simulation operates correctly.

[5]See Chapter 7 in [LK82] for details on these methods.

University of Cape Town

5. **Validate simulation with pilot runs**: Conduct pilot runs with chosen test data to compare the simulation results with expectation based on real systems. Adjust the model or input parameters where necessary based on feedback from these runs.

6. **Design experiments**: Determine the particular aspects of the system and the variances in their behaviour that need to be investigated. Develop input data that exercises these aspects in appropriate ways.

7. **Make production runs**: Conduct simulation runs based on the experiments of interest and collect generated results.

8. **Analyse output data**: Analyse results from the experiments to address the questions posed by the experiments.

9. **Document and implement results**: Document the assumptions, experiments and results for later reference. Implement results in the real system where appropriate, or use results to refine further experiments.

Our focus in this work has been on the design and development of an Enterprise Storage System simulation environment (step 1). Our study has therefore been focused on designing an appropriate, extensible model of such a system (steps 2-3), creating a simulation environment based on this model (steps 4-5) and developing a set of support tools. Steps 6-9 would typically be carried out by users of the system, who have a particular investigation in mind and wish to simulate several storage system configurations to derive an answer. As such, these steps are not directly addressed in our work, though Sections 6 and 7 present the support tools we have created to aid them.

University of Cape Town

3.5 Choosing a Simulation Environment

When developing a simulation, there are two main options available to the developer: constructing a simulation from scratch in a general purpose language such as C++; or utilising a simulation environment that provides the necessary management, statistics collection and support functionality required. Our reasons for choosing to use a simulation environment are outlined in Section 3.5.1, an overview of the simulation libraries we considered appears in Section 3.5.2 and our reasons for settling on the OMNet++ environment are presented in Section 3.5.3.

3.5.1 Advantages of Using a Simulation Library

Developing a simulation and the associated support environment in a general purpose language is feasible in situations where the system model itself is fairly small, or where specific optimisations or constraints are necessary. It is also of use in cases where physical hardware must be included in the simulation process. This approach can become complex, however, and has a tendency to divert development focus away from the simulation model as the supporting environment routines need to be tested and maintained.

By contrast, a simulation library offers a number of advantages over the above approach. Most importantly, it allows development to focus on the implementation of the system model, as the

supporting environment is provided for the developer to use. In addition, most quality simulation libraries have been validated and tested, both by the developing company and a community of users whose input further improves the software. This user community also provides a forum in which problems can be posed and solutions provided by knowledgeable developers who are well versed in both simulation and the particulars of the library.

A primary, but often understated, advantage of a simulation library is the expert knowledge it implicity provides. This is most apparent if the library imposes structural constraints on how simulation models are implemented and executed under the provided environment. This enforced structure encourages the use of best practices in simulation, such as:

- The use of a robust pseudo-random number generator with a large cycle-length[6] and a statistically valid distribution of the generated values.
- The management of event queues such that events are delivered to the appropriate recipients, and the associated simulation time is correctly updated.
- Providing various housekeeping functions that ensure that resources are allocated and freed correctly, and that contention for resources is correctly handled.
- Centralising simulation activity logging and statistics collection, providing a single interface usable throughout the simulation environment.

Finally, simulation libraries usually ship with example simulations illustrating various features of the environment. These examples not only flatten the learning curve, but can also serve as a simulation primer for the novice developer. They can also illustrate particular features of the library that simplify some aspect of simulation, or show how best to implement a particular pattern within the environment.

University of Cape Town

3.5.2 Available Simulation Libraries

Having outlined above the reasons for using a simulation library, we undertook an initial investigation of two simulation environments to determine which was most appropriate for our needs. Given the very large number of environments available, we chose to compare environments with very different design and execution ideologies: CSIM and OMNet++[7]. A description of each of these follows.

CSIM

CSIM[8] is a proprietary simulation environment that is widely used in business simulation. It is a Process-Oriented environment, targeting models with explicit producers and consumers of resources rather than models with multiple, independent, interacting components.

[6] The number of unique values generated in a sequence before the sequence begins repeating.
[7] Other alternatives, such as NS2 (`http://www.isi.edu/nsnam/ns/`), were not considered.
[8] `http://www.mesquite.com`

CSIM is implemented as a library of classes and procedures which implement all of the structures and operations necessary to implement a simulation model. CSIM thus requires that the end user be responsible for configuring, executing and controlling the resultant simulation.

As a process-oriented simulation environment, a typical CSIM simulation involves:

- Defining a set of processes that produce, consume or process resources.
- Allocating a set of facilities, that represent consumable resources.
- Executing the model logic, wherein processes queue, interact and compete for available resources.

This focus on autonomous processes competing for resources is not a natural match to the data-centric operation of I/O systems. Moreover, CSIM lacks support for loosely coupled systems, as each process must know precisely which resources it requires. This contrasts with an I/O request, where a process' request for data may be routed by an underlying process (the Operating system, for instance) to any one of a number of I/O devices.

An advantage that CSIM offers is a built-in statistics gathering and processing function. This capability is based around the various objects CSIM provides for constructing a simulation, and hence the statistical analysis can be automatically computed. A downside to this approach is that this mechanism cannot be automatically applied to custom objects. This requires users to either instrument their code manually for the statistics they wish to collect (which negates the advantage offered) or to use only the objects provided by CSIM.

The latter approach is problematic, in that it requires models to adhere to the CSIM process-oriented, producer-consumer model view, rather than selecting an appropriate idiom for the system at hand. This is constricting and can lead to cumbersome models that must work around the library, rather than with it. CSIM is also a proprietary library, with paid for support and development. This is useful in that bug fixes can be requested and effected immediately, but it prevents end-users from studying and extending the core functionality where necessary.

While the library has undoubtedly benefitted from the experience of the developing company over the course of many years, its lack of flexibility and enforced structure make it less appealing for the development of simulations with a wide range of applicability, or which do not easily fit the design paradigm it espouses.

OMNet++

OMNet++[9] [Pon93] is an Open-Source, object-oriented, modular, discrete event simulation library originally developed at the Technical University of Budapest. It is also available as a commercially supported distribution known as OMNEST.

OMNet++ was initially designed as a simulator for communication networks and the associated protocols, such as IPv4, IPv6 and MPLS (Multiprotocol Label Switching). The modular architecture and flexible nature of the environment have subsequently given rise to a number of other application areas from hardware architectures to business processes.

[9] `http://www.omnetpp.org`

University of Cape Town

The design of OMNet++ is focused around its primary application area of communicating systems. In such systems, it is the flow of information between distinct, functional objects that is of interest. This has resulted in a simulation environment tailored around message-oriented simulation.

A typical OMNet++ simulation consists of a number of modules (independent objects responsible for generating and processing messages), connected by a set of communication channels, along which the actual messages flow.

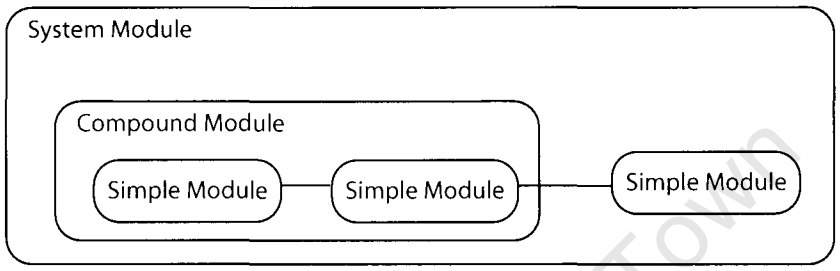


Figure 15: Hierarchy of modules in an OMNet++ simulation.

Modules map directly to C++ objects, and can be classified as either simple or compound. Simple modules perform the actual processing on messages, while compound modules serve as containers for organising the simulation into a hierarchy of nested modules, as illustrated in Figure 15.

Gates abstract the connection between modules, such that every module defines a number of gates. Channels are then bi-directional connections between any pair of gates. The notion of a gate allows a module to send a message to a particular gate without knowing the recipient module.

This anonymous messaging decouples modules, which in turn allows for a much more modular, extensible design of simulations. It also allows for reusability, since a module can be generically defined by a C++ module (for example a disk module) and then connected to any channel that provides it with messages it can understand.

The architecture of the simulation model is specified using a custom description language called NED (Network Description language). NED is editable either as a text file, or using the provided graphical editor, GNED. NED further extends modularity of OMNet++, by separating the simulation architecture description from the actual elements of the simulation.

Another major feature of OMNet++ is the Graphical User Interface (GUI) it provides. This interface greatly simplifies the user interaction with a simulation, as well as aiding understanding of the processes at work in said simulation. This GUI is dynamically generated from the simulation model description, and is implemented in TCL/TK [HLM+97]. The choice of TCL/TK further adds to the extensibility of OMNet++, as it allows GUI extensions to be written and added to the simulation environment.

Finally, a compiled OMNet++ simulation executable can store several independent simulation models, and the choice of model to use can be specified at runtime. This feature allows one to ship a single large executable containing several models as a standalone simulation tool.

University of Cape Town

3.5.3 Suitability of OMNet++

After an initial comparative study of the available simulation environments, OMNet++ was chosen as our simulation environment of choice. The message oriented simulation models espoused by OMNet++ is a good fit for the RAID I/O simulations, as outlined above, creating a natural correspondence with the movement of I/O Requests within a RAID system. This correspondence is further enhanced by OMNet++'s modular support, where a simulation model can be partitioned into modules that are functionally equivalent to their real-world counterparts. It is thus possible to create modules representing RAID Controllers, Disks and Data Sources, and to integrate these various modules into an overall simulation model, in the same way that the equivalent physical components are integrated into an overall system.

Completeness of Simulation Library

An important characteristic of a simulation environment is the presence of a comprehensive library of functions that encompass the range of common tasks involved in developing a simulation. These tasks include those directly related to the execution of a simulation, such as random number generation and timekeeping, and those that fulfill a support role, such as data manipulation. In this regard, OMNet++ offers the following noteworthy capabilities:

- A statistically sound Random Number generator based on the Mersenne Twister [MN98b] algorithm, which has a period of $2^{19937}-1$, and a 623-dimensional equidistribution property is assured. Using this base Uniformly distributed generator, OMNet++ also provides a comprehensive list of other distributions from which random numbers can be drawn (see Appendix C).
- A comprehensive simulation timekeeping functionality that is integrated with the entire simulation framework, thus allowing message transport effects associated with latency and bandwidth to be incorporated into a simulation model.
- Comprehensive, simulation-wide logging to a centralised output location, with the data partitioned according to the specific simulation execution.
- Extensive data manipulation functionality of the above output data using provided GUI tools. Data manipulation is also facilitated during simulation using adaptive histogram data collection classes.
- Parallel and distributed simulation execution over multiple machines, and across networks, by integrating with the Akaroa framework [EPM99].

OMNet++ is open-source software, and as such is maintained by the very same community of developers who use it to develop simulations. This offers a number of advantages over proprietary environments, chief amongst which is that bugs are identified and corrected much more quickly. The collaborative nature of the environment encourages contributions from many simulation practitioners, which improves both the quality and ease-of-use of the environment.

This input also represents a pool of knowledge that is encapsulated in the structure enforced by the environment, which encourages componentisation of code, reusability and an understanding of

University of Cape Town

the fundamentals of discrete event simulation. It also increases the usefulness of the accompanying documentation, which is both generated and consumed by the same group of users.

Simulation Portability

OMNet++ is completely implemented in C++, with only standard libraries as dependencies. This allows it to be completely cross-platform, with versions currently targeting Microsoft Windows^{TM}, Apple OS/X^{TM}, Sun Solaris^{TM}, FreeBSD and GNU/Linux^{TM}. This allows simulations to be developed for all these platforms using a single code base.

User Interface

OMNet++ provides two complementary interfaces to simulations developed using it. The first is a standard command-line interface that outputs status messages during simulation execution, but allows for no user interaction. This interface is ideal for executing batches of preconfigured simulations, and is similar to standard interfaces of other simulation environments.

OMNet++ also offers the unusual option of a Graphical User Interface (GUI), which visually represents the various modules of the simulation. The low-level support for this is provided by the TCL/TK scripting language [HLM+97], which allows powerful GUIs to be created at runtime. This GUI is dynamically generated from the model being simulated, and as such requires no extra work on the part of the programmer. During simulation execution, the interface animates the movement of messages within the system and provides control over the passage of simulation time (allowing time to be paused, sped up or slowed down).

When paused, various aspects of the internal simulation state can be interactively changed, with these changes taking effect once simulation time is restarted. This facility is useful in several scenarios: debugging a simulation model during development; developing an understanding of a simulation model authored by another developer; and initiating events to alter the state of a simulation for analysing once-off events.

University of Cape Town

Multi-Level Models

OMNet++ encourages hierarchical development of simulation models using nested components, which separate functionality into independent modules. This allows the developer to break a simulation model down into a series of ever simpler components, each representing a specific subset of the functionality of the entire system. This top-down approach to designing a simulation model is a well recognised design pattern.

It also allows for the representation of very detailed models, since each component can be refined internally without affecting its interactions with other components. Moreover, once a sufficiently detailed simulation module has been developed, it can be nested in any number of other simulation models without requiring modifications.

Chapter 4

Previous Work

University of Cape Town

As RAID systems are being developed, their performance is analyzed using either simulation or analytic methods. Simulators are generally handcrafted, or built on top of existing tools such as CSIM [1] and OMNet++[2] which may entail code duplication and potentially unreliable systems as explained by Holland [HG92]. On the other hand, RAID architectures, once defined, are difficult to verify correct without extensive simulations. In addition, simulators need input and therefore workload measurement tools.

Tools can be divided into one of three categories:

1. In the first category the tools are used to do *measurements* of drive storage systems to determine either the performance characteristics of a drive or to measure the I/O activity.
2. The second category is about the *design and development* of RAID redundancy procedures. One such tool is RAIDframe [CGHZ96a] which uses Directed Acyclic Graphs (DAGs) to define RAID architectures in terms of primitive operations, together with the roll-away error recovery scheme, to enable quick development and testing. The system has shown exceptional code reuse, and has been utilized by other, independent researchers, e.g. [ABC97].
3. The third category of software tools are for modelling the *performance* of RAID storage systems. There are several in this category and we will describe what we discovered about each of them in the following sections.

4.1 Measurement tools

4.1.1 Rubicon

Rubicon is a disk workload characterization tool made available in the public domain by HP[3], with the ability to perform many different types of analysis[4] on disk traces[VK03]. In general,

[1] Developed by Herb Schwetman of Mesquite Software Inc., Austin, Texas, www.mesquite.com.

[2] Developed by András Varga as part of his PhD work at the University of Budapest, www.hit.bme.hu/phd/vargaa/

[3] http://tesla.hpl.hp.com/public_software/

[4] Ranging from simple rate measurement (I/Os per second) to correlations between I/O streams, spatial and temporal locality measures and self-similarity properties.

Rubicon reads a sequence of disk trace records, performs some analysis on them, and outputs the result of the analysis. The user can configure Rubicon in several different ways. Firstly, the disk trace can be filtered in order to select subsets for analysis (e.g. single out a single logical volume), with multiple filtered streams undergoing analysis in parallel. New analysis functionality can be easily added to the system. The output generated is independent of the analysis; by adding a new "reporter" module, the same analysis can be reported in several different styles (e.g. as an Excel spreadsheet).

4.1.2 DIXTrac

DIXTrac [SG99] (DIsk eXtraction) is a program that can quickly and automatically characterize disk drives that comply to the Small Computer System Interface (SCSI). Such characterizations include data about mechanical delays, on-board caching and pre-fetching algorithms, command and protocol overheads, and logical-to-physical block mappings. It is not the only attempt at measuring disk characteristics, but it seems by far the most successful, probably supported by the fact that the disk model parameterizations accompany the DiskSim simulator discussed later in Section 4.3.1 (p. 45). Similar work is reported by B.L. Worthington *et. al*[WGP95] and M. Aboutabl *et. al.*[AAD97]

DIXTrac runs as a user-level application on Linux, using the `/dev/sg` interface to pass SCSI commands directly to the device driver. It requires no special hardware or operating system support.

DIXTrac's disk characterization process can be divided into five logical steps. First it discovers the basic physical geometry characteristics (e.g., numbers of LBAs, cylinders and surfaces) by translating random and targeted LBAs. Second, it finds out where any media defects are located. Third, explicitly avoiding defective regions, it figures out the sparing scheme (e.g., the allocation of spare sectors) used and the locations of any space reserved by the firmware. Fourth, it determines the boundaries and number of sectors per track for each zone. Fifth, the re-mapping mechanism used for each defective sector is identified.

The information in the disk map is necessary for the remaining steps, which involve issuing sequences of commands to specific physical disk locations. Second, mechanical parameters such as seek times, rotational speed, head switch overheads, and write settling times are extracted. Third, the cache management policies are determined. Fourth, command processing and block transfer overheads are measured. Since these overheads rely on information from all three of the prior steps, they must be extracted last.

DIXTrac has been used to characterize different disk models from various manufacturers, including IBM. Performing a characterization consists of simply pointing DIXTrac at the disk of interest. The characterization process generally requires less than 3 minutes to complete. Using the extracted characteristics to parameterize the DiskSim simulator gives rise to very close matches between simulated and measured disk performance.

Although the DIXTrac software is not offered for general use the authors are providing a database of over 20 validated disk characterizations on the Web[5]. IBM is a member of the Carnegie Mellon Parallel Data Laboratory Consortium which may mean that they would characterise a particular drive for IBM if so requested.

[5]http://www.ece.cmu.edu/ schindjr/DIXTrac and http://www.pdl.cmu.edu/DiskSim/diskspecs.html

University of Cape Town

Finally the technical report by J. Schindler G. R. Ganger [SG99] which describes DIXTrac contains a wealth of information about drive buffer management, the layout of data on the physical media and other disk operations.

4.1.3 I/O Stat

The AIX `iostat`[6] command is used for monitoring system input/output device loading by observing the time the physical disks are active in relation to their average transfer rates.

The `Disk Utilization Report` generated by the `iostat` command provides statistics per physical disk. The report contains the following:

- The percentage of time the physical disk was active or its utilization.
- The amount of data transferred (read or written) to the drive in KB per second.
- The number of transfers per second that were issued to the physical disk. A transfer is an I/O request to the physical disk. Multiple logical requests can be combined into a single I/O request to the disk.
- The total number of KB read.
- The total number of KB written.

Similarly the `Adapter Throughput Report` contains:

- The amount of data transferred (read or written) in the adapter in KB per second.
- The number of transfers per second issued to the adapter.
- The total number of KB read from the adapter.
- The total number of KB written to the adapter.

4.2 Tools for designing RAID systems

Apart from redundancy, there are three other issues at stake when designing RAID systems:

- The correctness or otherwise of the procedures.
- Given a particular workload, what effect the design will have on the system performance.
- Once one is satisfied with the answers to the above two questions, it remains to implement the design in software without destroying the credibility of the results obtained in the first two steps.

[6] `http://linux.die.net/man/1/iostat`

University of Cape Town

4.2.1 RAIDframe

One general approach to designing RAID systems is presented by Courtright et. al. [CGHZ96b] [CGHZ96a] and used by [VLW97]. Courtright outlines the design of RAIDframe, an application with a graphical interface that enables structured specification of RAID architectures, as well as detailed testing using built-in disk simulators or running the same code as real-world disk drivers. The system uses Directed Acyclic Graphs (DAG) to define RAID architectures in terms of primitive operations, together with the roll-away error recovery scheme, to enable quick development and testing. The system has shown exceptional code reuse, and has been utilized by other, independent researchers, e.g. [ABC97].

RAIDframe is therefore primarily 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.

The programming abstraction RAIDframe uses is based on directed acyclic graphs (DAGs). A designer wishing to introduce a new architecture or optimize an existing architecture will be able to achieve this goal by modifying the library of graphs and graph-invocation rules implemented in RAIDframe. While graphs and the binding of graphs to requests varies widely, the majority of the code in RAIDframe is found in the unchanging DAG interpretation-engine. In this way, designers are encouraged to experiment with and extend various RAID architectures because they can ignore the majority of the code, which is devoted to device-manipulation details. A typical DAG for RAID level 5 reconstruct-write is shown in Figure 16. Architecture features that cannot

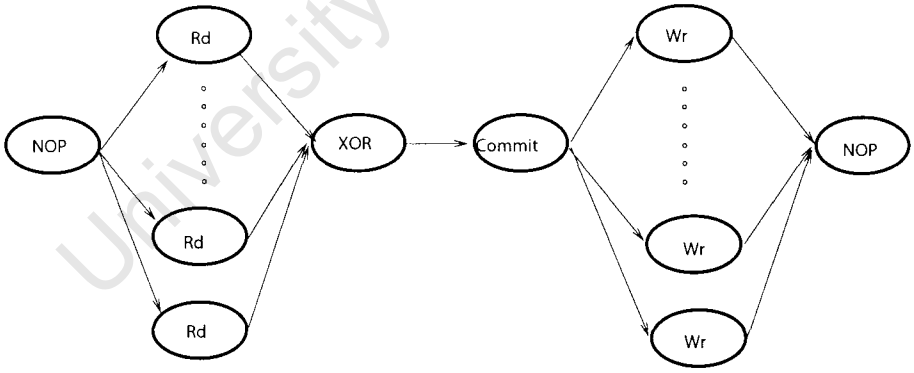


Figure 16: The write operation shown by the DAG in this figure writes both data and parity to disk. The cluster of *read* operations on the left side of the graph represent the read of old data and the single *read* operation at the bottom on the left represents the read of old parity. Once parity has been computed successfully, the new data and parity symbols are written to the array as shown

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.

RAIDframe moreover allows a single implementation to be evaluated in three distinct environments.

1. First, it provides a discrete event simulator with configurable array parameters and disk models. RAIDframe offers a synthetic and trace-driven load generator to exercise this simulator.

University of Cape Town

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).

The trace file contains actual I/O traces that have been collected from another application instead of synthetic traces that have been generated from a script. The trace file must contain a header and trace records. The header contains the number of independent processes in the trace, the number of traces for each process, and the file offsets for each trace.

2. Second, RAIDframe implementations and its load generator can be executed in a user process using the Unix device interface to access real disks instead of simulated disk models.

3. 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 a specific Unix operating system on specific workstations.

A particularly powerful feature of RAIDframe is that it separates error recovery from array architecture. The mechanism used to recover from failed primitive operations (such as a disk read) during the execution of an array operation is a part of RAIDframe's internal infrastructure. RAIDframe uses a two-phase approach to error recovery called *roll-away error recovery* to do this. RAIDframe's architecture-independent DAG interpreter handles errors by identifying those nodes in a DAG which commit data to disk and by specifying the direction of recovery based on when errors occur in relation to this commit point.

Specifically, if an error occurs before any data has been committed to disk, then the system rolls back, releasing resources, and retries the operation with a more appropriate graph. On the other hand, if an error occurs after data has been committed, the system rolls forward through the remainder of the graph, giving later requests the impression that this graph completed instantaneously before the error. In either case, this process is hidden from the user and performed without regard to array architecture. Graph commit points (see Figure 16) can be specified so that roll-back is inexpensive (that is, it does not induce additional device work in preparing for or executing roll-back) and so that roll-forward does not need to execute any device operation not already coded in the in-progress graph. By eliminating the need for architecture-specific code for handling errors, roll-away error recovery further simplifies the process of building new RAID architectures: there is no need to create or alter thousands of lines of error-recovery code.

Although aimed at RAID implementations in the first instance, RAIDframe has the facility to compare how different RAID architectures perform relative to one another when implemented in RAIDframe. It enables users to test throughput versus response time for various RAID architectures and configurations.

RAIDframe was developed in the Parallel Data Laboratory in the Computer Science Department of Carnegie Mellon University. It is said to be freely available to the research community, but seems to have fallen into disuse. The latest version of the comprehensive and informative RAIDframe manual [CGR+97] is dated June 1997.

University of Cape Town

4.2.2 DiskRaid

DiskRaid from MicroSoft is a command-line tool that enables configuration and management of RAID storage subsystems. No details could be found about this software.

4.2.3 RAIDtool

The RAIDtool[ZRM96][ZMR96] was developed by Jai Menon and Jeff Riegel at the IBM Almaden Research Center. Although still available the information[7] was last updated in April 1996 and it would therefore seem to have fallen into disuse. The stated objective in building *RAIDtool* was to develop a fast and easy to use software tool which allows the efficient evaluation of different RAID configuration alternatives [ZMR96].

RAIDtool accepts a description of a RAID array controller card as its input. The architecture on the card would however seem to be restricted to that shown in Figure 17 and although several drive array strings can be attached, the possible configurations are not many. The RAID array

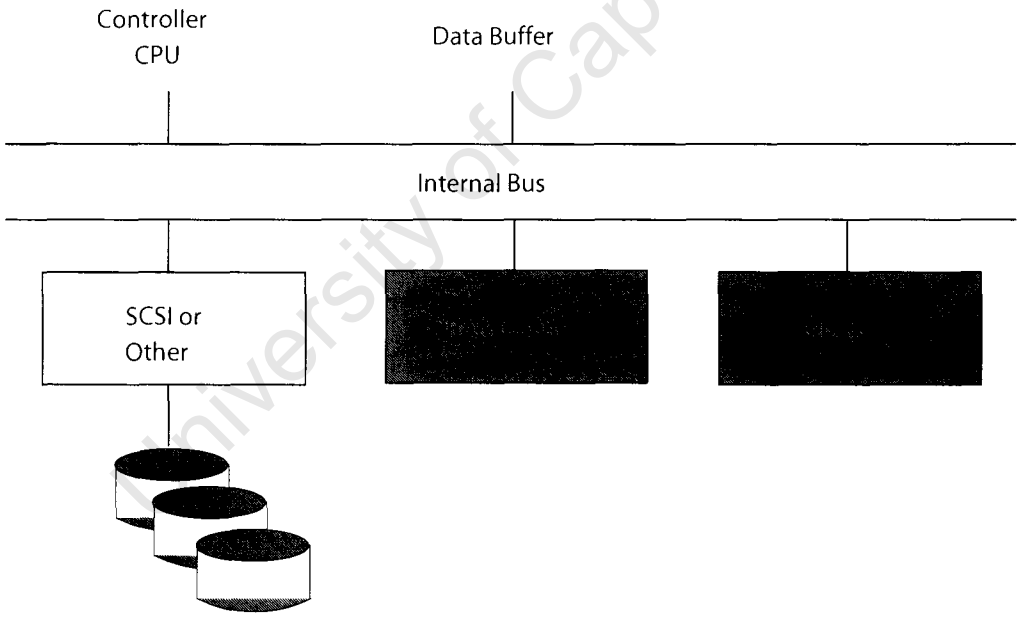


Figure 17: The standard RAIDtool architecture contains a data buffer, an optional DRAM for data caching, and an optional non-volatile RAM (NVRAM) for fast writes, all connected by an internal bus with a certain bandwidth. All the characteristics (speed, bandwidth, capacity, etc.) of the components and the bus are configurable

controller card is configurable for an unlimited number of RAID groups which may be configured for the same or different RAID levels and each group may have a different disk type. Disks themselves are configurable and support pre-fetch sector buffering.

The tool distinguishes between three different I/O-requests, each with a different priority. Listed in order from highest to lowest, these are:

[7] http://w3.almaden.ibm.com/ riegel/conftool.html

University of Cape Town

1. Regular I/O-requests which may pre-empt all others.
2. Destage requests from the NVRAM to disk.
3. Rebuild requests to reconstruct a failed disk.

Since RAIDtool allows for all RAID levels up to RAID 5, the scheduling of additional reads and writes for parity calculations during writes in RAID 5, for example, is simulated in detail using three different scheduling strategies ([ZMR96] page 27).

For more detail the reader is referred to the IBM Research Report [ZMR96] by P. Zabback, J. Riegel and J. Menon.

4.3 RAID performance modelling tools

There are several tools for modelling disk storage systems which do not necessarily have to be redundancy or parity protected. Although analytical models exist for simplified disk systems such as that by Peter Harrison[HZ04] this chapter concerns software tools only.

4.3.1 DiskSim

DiskSim [Bea03] is an efficient, accurate, highly-configurable disk system simulator developed at the University of Michigan and enhanced at Carnegie Mellon University to support research into various aspects of storage subsystem architecture. It is written in C and requires no special system software. DiskSim includes modules for four secondary storage components of interest, namely device drivers, buses, controllers and storage devices. Storage devices are the abstraction through which the various storage device models are interfaced with DiskSim. In the current release, there are 2 such models: conventional disks, and a simplified, fixed-access-time disk model which, *inter alia* allows no disk cache. There are two further components namely queues/schedulers and caches which serve as subcomponents of the above components.

The possible interconnections are independent of the components themselves except that an I/O-path from the host must begin with a single device driver and terminate with the storage devices. Exactly one or two array controllers must be between the device driver and each disk, with a bus connecting each such pair of components along the path from driver to disk. Each disk or controller can only be connected to one bus from the host side of the subsystem. A bus can have no more than 15 disks or controllers attached to it. A controller can have no more than 4 back-end buses. The system topology is specified to DiskSim via a topology specification. A topology specification consists of a device type, a device name and a list of devices which are children of that device. The named devices must be instantiated; the component instantiations should precede the topology specification in the parameter file. In the current implementation, no device may appear more than once in the topology specification. Future versions may provide multi-path support.

An example DiskSim topology is provided in Figure 18 below. The specification of the storage system (disk) addresses three aspects, namely:

University of Cape Town

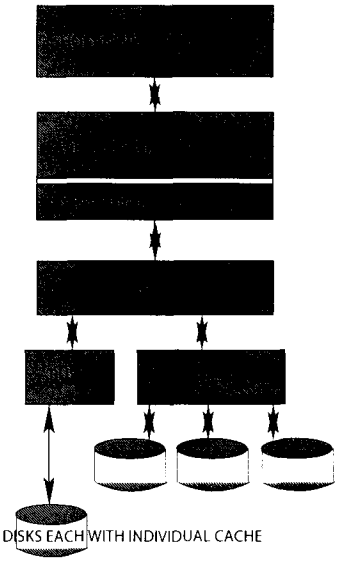


Figure 18: Typical DiskSim storage system configuration

1. The relationship with the bus, such as whether the bus is released during media access time or not.
2. The disk cache management with comprehensive options such as size, write pre-buffering, read-ahead options, fast write level, allowing cache hits on read and/or write, etc.
3. The disk actuator control such as allowing the actuator to begin relocation as soon as the access of the last sector of the current request has been completed, seek scheduler options (FCFS, SSTF, etc.) including whether detailed media mapping is available, and so on.

The (array) controller can be configured with its own cache. The cache management options available in that case are not as rich as that for the disk cache management, but include size, segment (block) replacement algorithms (FIFO, LRU or LIFO), fast write, read pre-fetch, size of individual I/O-transfers, etc.

DiskSim can simulate a variety of logical storage organizations, including striping and RAID Level 1 and 5 architectures. Although DiskSim is organized to allow such organizations both at the system-level (the front end of the device drivers) and at the controller-level, only system-level organizations are supported in the current release. It would not appear from the manual that DiskSim explicitly allows for disk rebuilds or reconstruction.

The DiskSim simulation can be driven by either a trace or a synthetic workload.

In the case of an I/O-trace, the default input format is a simple ASCII stream (or file), where each line contains values for five parameters which are:

- Request arrival time in milliseconds.
- Device number i.e., the storage component that the request accesses.
- LBA start address of the request.

University of Cape Town

- Request size.
- Operation, i.e., read or write, etc.

In the event that a synthetic workload is used to drive the simulation, then apart from specifying the length of the simulation, DiskSim generates values for fourteen parameters, where some are generated from a probability-distribution which may be one of the following:

- Uniform between a specified minimum and maximum value.
- Normal, requiring a mean and variance to be specified.
- Exponential, requiring a mean value to be specified.
- Poisson, requiring a mean value to be specified.
- A three parameter distribution, where a specified probability determines which of the two other specified values are returned.

Apart from the obvious parameters such as request size and inter-arrival time, the probability of sequential access along with the probability of local access, determine how a generated request's starting address is assigned. Each request's starting address is sequential, local or random. A random request is assigned a device and starting address from a uniform distribution spanning the entire available storage space. The reader is referred to the User Manual[Bea03] for more information.

DiskSim reports a comprehensive list of performance metrics at the overall system level as well as for each component in a configuration. The statistics can be turned on selectively when providing the corresponding configuration parameters in the input to DiskSim. Again, the reader is referred to the User Manual[Bea03] for more information. It makes no provision however for computing confidence intervals for any of the values reported and presumably leaves that to the user.

Finally, and very important, DiskSim explicitly includes hooks for inclusion into a larger scale system-level simulator and will be used to model individual disks in RÖSTI.

University of Cape Town

4.3.2 raidSim

The RAID simulator, raidSim[CP90] was developed at the University of California at Berkeley in 1990. It consists of a module for implementing a variety of RAID levels, a module for modelling disk behavior, and a module for generating synthetic I/O requests. The only resources modelled are disks.

Despite being no longer in use raidSim had interesting properties. It worked in three main steps. The first step was to collect information about the typical workload as I/O-traces. Rather than use these traces directly, the data was analyzed with a dual purpose in the second step, namely

- to generate a synthetic workload description with characteristics similar to the measured one, and

- to determine key workload characteristics which are important for the configuration of a RAID controller.

This approach allowed the simulator to be run several times using a scaled workload with, for instance, different arrival rates while doing processor intensive work, such as cache modelling, only once in the analyzer rather than every time in the simulator. In the third step different RAID array controller configurations were simulated.

The raidSim analyzer had two ingenious ways, in particular, of dealing with the real trace data. First of all it translated the location specific data in the trace to logical seek distances and then to seek times to be used in the simulator.

For the second clever idea, raidSim did not simulate cache management during simulation but in order to speed up the simulation, used the probability of a cache-hit calculated from the I/O-trace for each of four cache configurations:

1. read cache only (a hit in either the NVRAM or controller cache),
2. write hit in the RAM cache (i.e., the same block was written before, but not yet destaged),
3. write hit in the NVRAM cache (i.e., the block is different but has not yet destaged),
4. hits in the read and and both write caches.

In addition, the analyzer generated hit ratios for the disk buffers from the I/O-trace with a separate buffer hit ratio for each disk.

Other workload characteristics were request sizes which the workload synthesizer computed randomly from a distribution derived by the analyzer from the I/O-trace. Inter-arrival times were assumed to be exponentially distributed at high arrival rates and the simulator allowed for variations in the mean arrival rate derived by the analyzer over a window of a fixed number of I/Os. The controller processor time was accounted for in the simulation by specifying a MIPS rate and the instruction path length for a read hit/miss and a write hit as well as a write de-stage. SCSI or SSA interface overheads were fixed and taken from the corresponding standardization documents.

4.3.3 Pantheon

The Pantheon[Wil96] simulator was designed and built in the Storage Systems Program of the Hewlett-Packard Computer Systems Laboratory in Palo Alto to support the rapid exploration of design choices in storage systems and their components such as disks, tapes and array controllers. It has been functional in some form or another since summer 1992.

The Pantheon simulator proper is constructed from a set of primitive simulation components together with infrastructure to glue these together to configure and execute a simulation. A Pantheon simulation is primarily driven by recorded disk-level I/O traces but it has a less well-used workload generator facility. The package includes a set of analysis tools that can summarize the results of simulation runs.

Pantheon's simulation modules are written in C++, compiled and linked together to make a single Pantheon executable program. Using compiled building blocks of this form means that

University of Cape Town

simulations execute at full speed: the runtime cost of linking the components together is just a C++ virtual function call. In addition, new components can be added, or existing ones extended, to meet particular needs.

Considering that Pantheon was developed over a decade ago, it still uses an interpreted language, *Tcl*[Ous94], to control which simulation modules are to be instantiated, and how they should be connected and parameterized. Since *Tcl* is a full programming language, arbitrarily complicated configuration decisions are possible: for example, it is possible to calculate how many disk drives an array needs to accommodate its load as a function of its redundancy algorithms, rather than having to pre-compute this. The net result is that Pantheon achieves both great configuration flexibility and good execution-time performance.

Particularly instructive or interesting about Pantheon is the detailed way in which cache is modelled. Caches in Pantheon include support for a simple speed-matching pipeline (e.g., between the DMA engine and a drive) or FIFO buffer, caches that can hold contiguous data or multiple, separate items of data, multi-segment caches, and ones that have arbitrary replacement policies like LRU.

A *Cache* is simply a virtual class that specifies the operations that can be done on all the different kinds of cache: adding and removing address-ranges, searching for data that overlap a given range, and so on. In addition, there are producer-consumer semaphores, and flags indicating whether a specific cache-data range holds dirty data that has yet to be written out, or if it is fixed in memory and must not be selected for replacement.

The two basic ways in which Pantheon models the use of a cache are as:

1. speed-matching buffers between a producer and a consumer where the producer extends the range, space, valid, and ready areas while the consumer shrinks them. Extensions occur at the ends; i.e., the lengths increase, while shrinkage occurs at the beginning of the ranges, by advancing the start address and adjusting the remaining lengths accordingly, or

2. buffer caches, that have long-term state. Here the typical mode of use is that the range and space lengths are set equal when the cache-data range is created, and the valid and ready lengths advance in lock-step. The start addresses of these elements never change: instead, it is deleted when it is done with.

There are circumstances where a process thread needs to await some set of states across multiple caches. For example, that one cache contains no dirty data that overlaps a given range, while the other contains no fixed data ranges. Unfortunately, in the time required for the second set of conditions to become true, the first set might have been invalidated. The cache modelling functions address this by backtracking to re-establish earlier conditions if necessary.

Replacement and flush policies are modelled in Pantheon to decide what to do if space is required in a cache, but there is none available. In this case, a *replacement policy* function decides which existing cache data blocks to evict. If a replacement policy picks a dirty block to evict, it will usually first have to be written out to a lower-level device. The flush policy modelled allows additional blocks to be written out at the same and so on.

Further details are in the well-written summary[Wil96] about Pantheon.

University of Cape Town

Why use a tool?

In conclusion to this part it is probably wise to pause and ponder the role of modelling in the development of a new RAID system. Whereas it is accepted wisdom that one needs a model of some sort to verify certain aspects of what one is doing, a much more reliable, but less cost-efficient, approach is to build a prototype of the planned system. Even when simulating the implementation of a new redundancy coding technique, the detail which would have to be incorporated in a simulator would almost require as much coding as building a prototype.

For instance, consider a large write operation in an existing RAID level 5 array which overwrites data and parity with new information. The previous contents of the data and parity must be stored in the log, to guarantee that each of these write operations is undo-able. Instead of just overwriting each one, each disk operation must now read and write data and parity, doubling the total workload of the disks and decreasing the response time and throughput of the system. If a disk operation fails, then the saved state is restored; and, while the system restores state, processing stops. Whereas the performance impact of this can be modelled to some degree of confidence, one would have to build an implementation to be confident that one fully understands all aspects of the proposed new technique. The raidSim tool discussed above was developed for exactly this purpose, and RÖSTI is intended to provide similar functionality with greater ease-of-use and extensibility.

University of Cape Town

Part II

Implementation

University of Cape Town

Chapter 5

Building the Simulator

University of Cape Town

The **RAID Operations Simulator for Testing Implementations** (RÖSTI) forms the core of this work, and its origins are covered in Section 5.1. RÖSTI was designed to be an extensible, easy-to-use simulator that models the operation of a RAID Controller in an Enterprise Storage System (ESS). This chapter outlines the development of RÖSTI, from User Requirements Specification (Section 5.3) through the Architecture (Section 5.4), Design (Section 5.5) and Implementation (Section 5.6) phases. We also address problems encountered during implementation (Section 5.7).

5.1 RÖSTI and IBM

The RÖSTI project originated as part of a Joint Research Agreement between the University of Cape Town (UCT) and the IBM Research Lab in Zurich, Switzerland (ZRL). IBM Research was interested in developing new RAID technologies to address the need for larger capacities, which in turn require better protection methods. The Data Network Architecture (DNA) group in the Department of Computer Science at UCT proposed a collaboration with ZRL, leveraging its expertise in modelling and performance analysis to assist in this research. This collaboration was formalised with a Joint Research Agreement (JRA) that was signed by the two institutions, describing the areas of research and projects that would be jointly pursued.

RÖSTI was proposed as one of the initial projects that formed part of this JRA. The aim of RÖSTI was envisioned to be a simulator that would enable IBM to verify and validate the analytical models they were working with, as well as to compare and contrast with other simulation work they were conducting. RÖSTI would thus have to be: expressive, allowing many RAID architectures to be simulated; extensible; and easy to use[1].

Development of RÖSTI was approached as a normal software engineering task, involving the development of a simulator which is supported by configuration and analysis software modules. As part of this focus, software engineering best practices were used where appropriate, and a Release Schedule agreed upon that set clear milestones for delivery. This formal engineering approach guided the development of RÖSTI and simplified its implementation.

Given that RÖSTI is intended to be modular and extensible, a Responsibilities Driven design

[1] As the primary users would not be the developers

approach was adopted [WBW89]. This approach concentrates on segmenting a design into components with unique responsibilities. The contrasting Data Driven approach is inappropriate for object orientated design, as it encourages tightly integrated designs, with overlapping responsibilities.

5.2 UML

The Unified Modelling Language (UML) [RJB99, BRJ99] is an open industry standard maintained by the Object Management Group (OMG). UML is used to model application structure, behavior, and architecture. UML consists of a number of different diagram types, each with a different use. A good overview of the various diagrams may be found in [Mil03]. A brief synopsis of the article, for diagrams utilised in the design of RÖSTI, appears below.

1. **Use Case Diagrams** describe what a system does from the standpoint of an external observer. The emphasis is on what a system does rather than how. Use case diagrams were used to express the User Requirements obtained from ZRL (Section 5.3).
2. **Activity Diagrams** focus on the flow of activities involved in a single process. The activity diagram shows how those activities depend on one another. Activity Diagrams were used to describe the function and operation of the RAID Controller within RÖSTI (Section 5.3 (p. 53)).
3. **Class Diagrams** give an overview of a system by showing its classes and the relationships between them. Class diagrams are static - they display what interacts but not what happens when they do interact. Class diagrams were used to model the basic components of RÖSTI, and the relationship between them (Section 5.5 (p. 70)).
4. **Sequence Diagrams** detail how operations are carried out - what messages are sent and when. Sequence diagrams are organized according to time. The time progresses vertically, and the objects involved in the operation are listed horizontally according to when they take part in the message sequence. Sequence diagrams were used to model the message flows between components (Section 5.5 (p. 70)); a very important part of the design of RÖSTI.

ArgoUML[2] is a widely used, open-source UML creation tool we chose to create these diagrams. It is a simple tool to use, but still sufficiently powerful to create all the relevant UML diagrams we required.

5.3 User Requirements Specification

In any well-engineered software product, obtaining a complete list of user requirements is the first step [WBW89]. It ensures that the expectations of the client are clearly understood by the developers, and that these expectations are both realistic and feasible. Achieving these two aims helps prevent common development problems, such as: Feature Creep, where the list of requirements continually grows during development; changing requirements which extend development time;

[2]Open-source application, available at `http://argouml.tigris.org/`.

University of Cape Town

producing a product that does not meet user expectations; and having to continually refactor the system because of a lack of direction or focus.

In designing RÖSTI using Software Engineering best practices, we avoided the above pitfalls and ensured that the capabilities of RÖSTI were tightly constrained, thus ensuring that the software could be completed in the time available. The User Requirements of IBM ZRL as the client hence focused the rest of the design of RÖSTI.

5.3.1 Types of Requirements

When discussing requirements, it is important to differentiate between functional and non-functional requirements, as outlined by Mylopoulos et. al. [MCN92].

University of Cape Town

Functional Requirements:

Functional requirements capture the behavioural characteristics of a system. They define what the system should be capable of and are usually the first, and only, requirements considered during design. This is understandable, as they provide the most immediately useful information for programmers implementing the software. This is particularly true for the simulation core of RÖSTI, where correctness of operation is of primary importance.

The functional requirements for RÖSTI helped to define the system model that was used, by focusing the work on particular areas of interest within RAID systems in general. The requirements also helped refine this model, by focusing on specific aspects of the operation of a RAID system. This system model was then translated into a simulation implementation. Finally, the functional requirements were aligned with the capabilities of the simulator for the research areas in which it will be used.

Non-functional Requirements:

Non-Functional requirements generally relate to the software interface. They capture user expectations of modes of interaction and workflows. These are features of the system that are often overlooked, but are frequently most important – a system that is hard to use is frustrating.

Modes of interaction describe how the user expects to utilise the various functions offered by the system, as it is important that system functionality be easily accessible. Workflows describe how various functions are used together. For instance, it might be normal to open a simulation, alter a number of simulation parameters, and the execute the simulation. In such a case, it is important that the user interface support such a scenario, without introducing unnecessary complications.

For RÖSTI, the interface may be of little concern to specialist users, who are accustomed to interacting with software via configuration files and other complicated methods. However, the importance of a simple interface is underscored by our experience with DiskSim [3]. As we found with DiskSim, it is often necessary to use a simulator without being familiar with how it operates. Possible reasons for this are:

[3] See Section 5.7.3 (p. 81)

1. Assessing the suitability of a simulator to a given task or research area, such as the initial investigations conducted with DiskSim.
2. Determining the constraints and limitations of a particular simulator, as part of the above.
3. Delegating simulation tasks to an assistant, whose knowledge of a given simulator may not be as complete.
4. Automating simulation execution, through the use of batch runs. An interface that supports this natively, rather than require the use of external scripting facilities[4] greatly simplifies and expedites this process.

Our approach with RÖSTI was an attempt at balancing powerful functionality with ease-of-use. Thus, we have created an extensible backend simulator, using configuration files as stipulated by OMNet++[5], wrapped within a Graphical User Interface (GUI) that is both intuitive and easy-to-use.

University of Cape Town

5.3.2 Obtaining User Requirements

Our approach in obtaining user requirements for RÖSTI was based on that outlined by Wirfs-Brock and Wilkerson [WBW89]. We began with a series of informal discussions around the subject area of RAID Storage Systems, and how best to assist IBM's research goals. This process allowed us to gain an understanding of the work that was being undertaken, and determine that developing a simulator was the best course of action.

The next step was a series of formal discussions, eliciting areas of interest to the researchers and hence the functional requirements for the system. This was followed by formal feedback sessions, presenting our initial high-level designs via diagrams and verbal explanations. This was an iterative process, intended to refine requirements, and identify differences between user expectations and our understanding of their requirements.

An important part of this process was functionality walkthroughs, where particular parts of the system model were discussed in depth. For example, the operation of the RAID Controller was analysed by discussing each step required to satisfy any I/O Request. These walkthroughs were fairly technical in nature, as they formed the basis for the simulation model and approach adopted.

Discussions on interface requirements were also conducted to establish what was required. Restrictions imposed by OMNet++ initially limited the scope of these requirements to manual editing of configuration files. However, these requirements were supplemented with user feedback during testing of the simulator, at which point common usage patterns could be established. These patterns helped determine the focus of the GUI simulator interface, discussed in Chapter 6.

As part of the formal design process we adopted, it was necessary to transform the user requirements we obtained into a formal representation, from which lower level design documents could draw. As stated previously, UML was chosen as the formal language of choice. The following sections provide formal representations of both the Functional and Interface (Non-Functional) Requirements.

[4]Such as Bash, a common command line interface to the GNU/Linux operating system.
[5]See Section 3.5.3 (p. 37)

5.3.3 Non-Functional Requirements Specification

An overarching consideration for RÖSTI was that we foresaw it being used as a simulation environment by people who had not necessarily designed the simulation. Configuring a simulation, executing and examining the output should therefore not require detailed knowledge of the simulation being performed. Further, the design of RÖSTI supports common patterns of usage for simulation environments. Figure 19 illustrates the interactions between a user and RÖSTI, and the likely sequencing of these interactions. This dictated the design and relationship of the user interface components of RÖSTI, in order to more closely model the users' likely usage patterns.

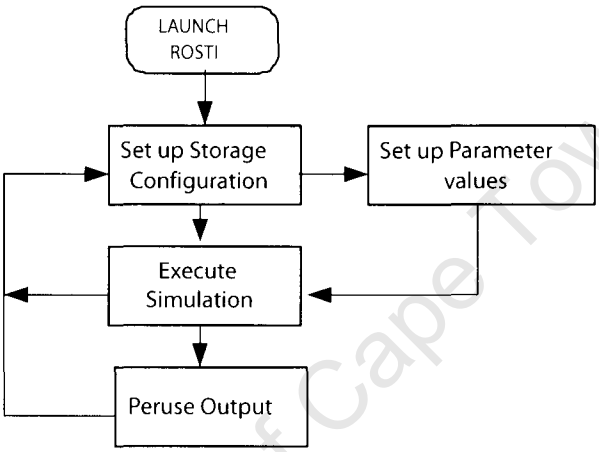


Figure 19: Collaboration Diagram illustrating typical usage patterns of the RÖSTI system.

The actual actions that can be performed in RÖSTI by a user are illustrated in Figure 20. These actions relate to interacting with a particular simulation model, and as such are differentiated into three categories, each relating to a particular aspect of the simulation model.

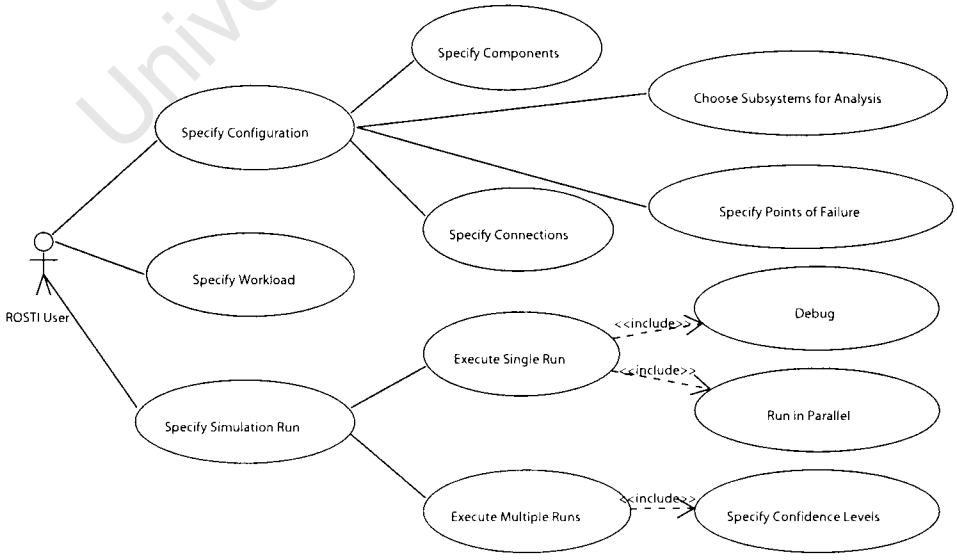


Figure 20: Use Case Diagram illustrating user requirements of simulator interface.

University of Cape Town

Specify Configuration

Configuring the simulation involves:

- Choosing the components used in the simulation model (disks, controllers, caches, etc.) and setting the parameters for each component.
- Specifying the connections between these components (IDE, SCSI, SATA, etc.) and setting applicable parameters for these connections.
- Choosing a subset of components to fail during simulation execution and configuring the timing and nature of these failures.
- Identifying and specifying particular sub-systems and components that will generate output for use in analysis.

A particular configuration identifies a unique instance of the simulation model, thus specifying a configuration equates to choosing a model to simulate. In RÖSTI this choice can be made from a set of pre-defined configurations (by loading a saved configuration), or by allowing the user to create a new configuration that is subsequently used. This flexibility allows RÖSTI to be used effectively by both novice and experienced users.

Specify Workload

Once the simulation model instance has been configured, it is necessary to specify the workload that will exercise the model. This workload specification will vary depending on the whether the workload is trace-driven or synthetic. In the former case, one need only specify an appropriate trace file, while the latter requires that a number of parameters governing the workload be appropriately set. Both actions were sufficiently similar to configuration of the simulation model that these two use cases could be integrated into a single user interface.

Specify Simulation Run

Having configured both the simulation model and the workload it is presented with, it is then necessary for the user to specify how the simulation model will be executed. If a single simulation run is required, the user must decide if the run should be in debug mode (with a graphics visualisation of the simulation operation) or possibly executed in parallel over multiple machines. If multiple simulation runs are required (for statistical error analysis), the user should be able to specify a required confidence level for which statistics will be calculated.

Analysing Results

Whereas configuration and execution of a simulation model occur before results are generated, analysis of these results can naturally only occur after results are available. Analysis of simulation results thus forms a separate set of use cases, as illustrated in Figure 21. User interaction with an analysis tool typically consists of selecting a set of sub-systems of interest, such as the set of disks

University of Cape Town

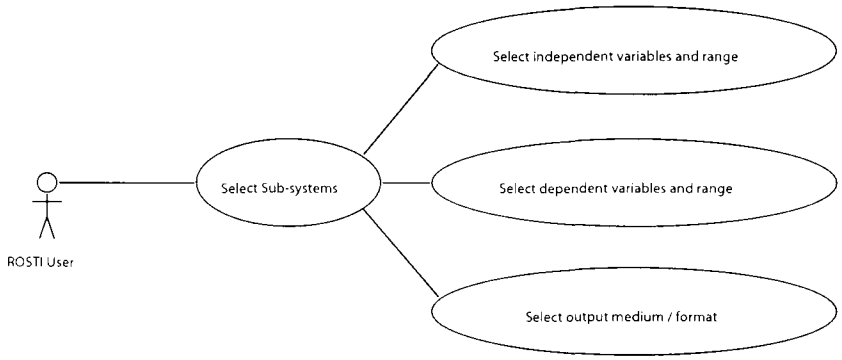


Figure 21: Use Case Diagram illustrating user requirements of analysis tool interface.

mapped to a given logical volume, and then performing some form of analysis on the outputs from this set. Typical tasks are: selecting a subset of independent parameters defining some aspect of the sub-system (such as the chosen RAID level); then selecting a subset of dependent statistical outputs (such as the response time for serving I/O requests); and then selecting an output medium and format for this comparison (such as plotting a graph of independent parameters versus dependent outputs).

5.3.4 Functionality Requirements Specification

The functional requirements for RÖSTI refer primarily to the manner in which RAID operations are performed. Specifically, since RÖSTI is a RAID Operations Simulator, it was necessary to identify the common RAID operations that can be performed. and formalise the steps involved in their execution. We consider here the general case of a RAID-like operation being performed on an array of disks, taking into account the effect of parity on both the operations and error recovery[6]. This generic approach allowed us to design the high-level structure of RÖSTI without being tied to the specifics related to a given RAID level.

Normal Mode Read

A Normal Mode Read operation refers to the case where the RAID Controller receives an I/O request to read data while the array is in Normal Mode (no disks have failed). The steps involved in fulfilling this request are illustrated in Figure 22. If the request maps to unprotected storage (such as RAID 0), the data is simply read from disk. However if the request maps to a RAID protected set of disks, it is first necessary to examine the RAID level cache to determine which portions of the requested data is available there. All data that does not exist in the cache must be read from disk and the cache appropriately updated with these. The final result is then reconstructed (to account for striping across disks) and returned to the requesting host.

Normal Mode Write

A Normal Mode Write operation refers to the case where the RAID Controller receives an I/O request to write data while the array is in Normal Mode (no disks have failed). The steps involved

[6]The specific steps involved for each of the supported RAID levels in RÖSTI can be found in Section 8.3 (p. 97)

University of Cape Town

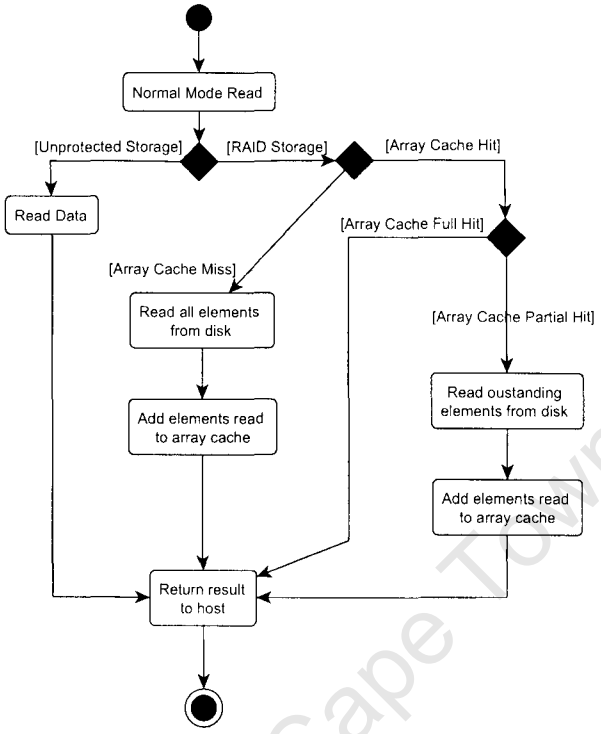


Figure 22: Activity Diagram illustrating actions to be performed during a Normal Mode Read.

in fulfilling this request are illustrated in Figure 23. If the request maps to unprotected storage (such as RAID 0), the data is simply written to disk. If the request maps to a RAID protected set of disks, the parity must be recalculated and the cache updated with the new data (dependent on the cache policy). For the parity calculation, if an entire stripe is being written then all necessary data is available to calculate parity so no reads from disk are required. If less than a full stripe is written, the missing data to recalculate parity must be read from disk. Once parity has been recalculated, the new data and parity can then be written to disk.

University of Cape Town

Degraded Mode Read

A Degraded Mode Read operation refers to an I/O request to read data while the array is in Degraded Mode (at least one disk has failed). The steps required to fulfill this request are illustrated in Figure 24. If the request maps to unprotected storage, the data is read and returned if all required strips are available. If any of the required strips have failed, the request returns an error. If the request maps to a RAID protected set of disks, the request can be treated as a Normal Mode Read if either only parity or unaffected data is missing. If data strips are unavailable, the RAID cache must first be checked for any of the requested data. Data not present in the cache must then be read from disk or reconstructed as appropriate[7]. Once all requested data has been retrieved, the result can be returned to the requesting host.

[7]For details see Section 8.3.8 (p. 106)

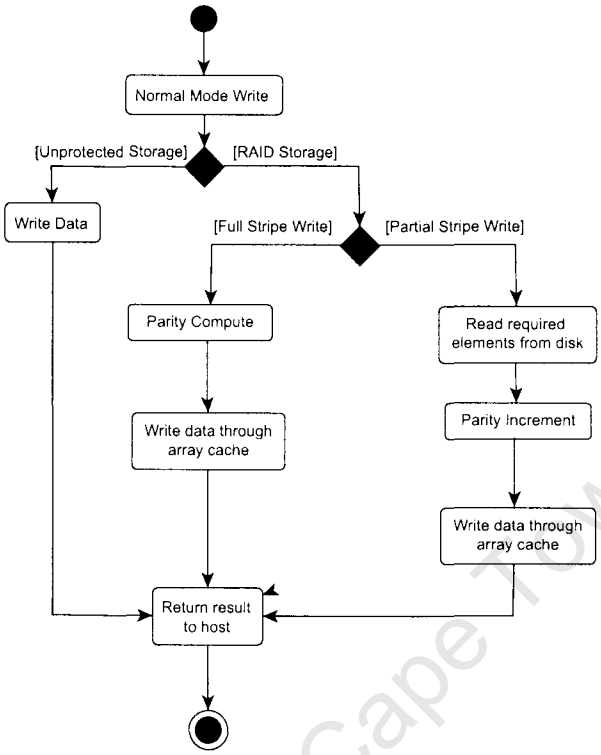


Figure 23: Activity Diagram illustrating actions to be performed during a Normal Mode Write.

University of Cape Town

Degraded Mode Write

A Degraded Mode Write operation refers to an I/O request to write data while the array is in Degraded Mode (at least one disk has failed). The steps required to fulfill this request are illustrated in Figure 25. If the request maps to unprotected storage, the request returns an error since it is not possible to write all the provided data. If the request maps to a RAID protected set of disks, the request can be treated as a Normal Mode Write if all the requested data and parity strips are available. If data strips are unavailable, the RAID cache must be updated with the provided data. If all data strips in a stripe are being written, the parity can be directly computed. If only part of a stripe is being written, the missing data strips must be reconstructed and used to recalculate parity together with the provided data from the I/O request[8]. Once parity has been recalculated, the new data and parity can be written to the surviving disks in the array.

5.4 Architecture

In defining the architecture of RÖSTI, we define a high-level model of the system, its components and their interactions. This model was based upon the User Requirements elicited in Section 5.3 and guided the development phase. The construction of the model and the ways in which it can be extended are covered in this section.

[8]For details see Section 8.3.9 (p. 108)

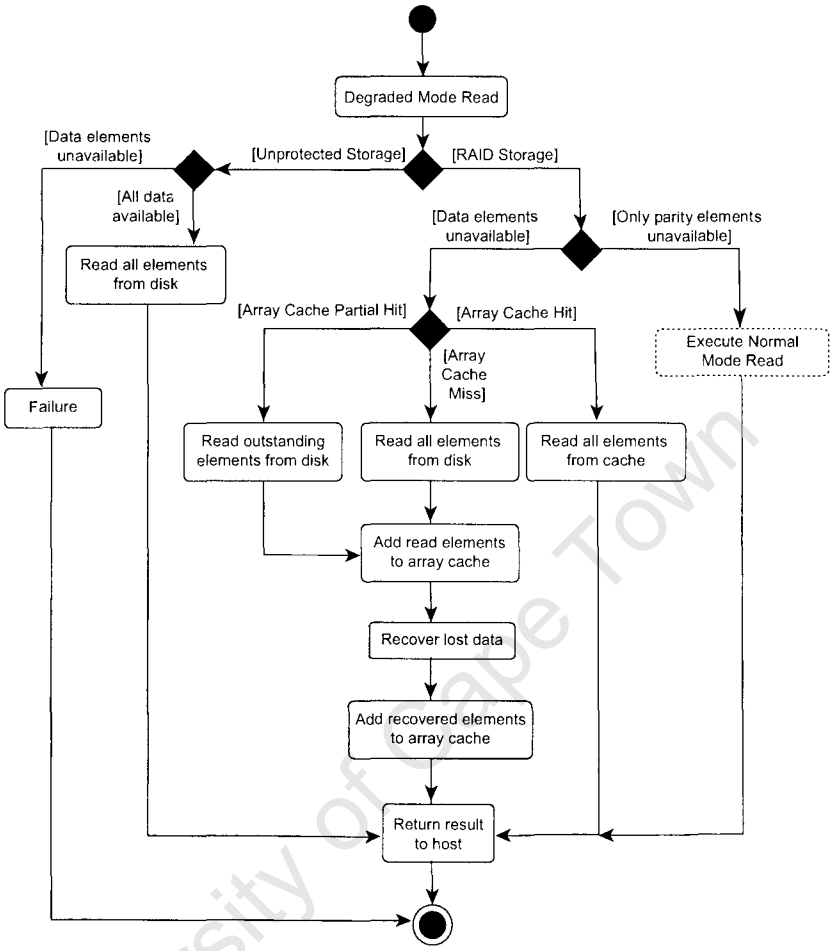


Figure 24: Activity Diagram illustrating actions to be performed during a Degraded Mode Read.

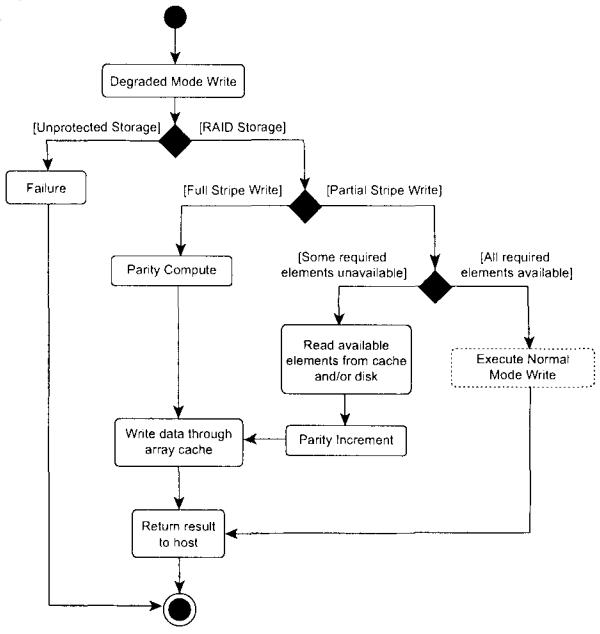


Figure 25: Activity Diagram illustrating actions to be performed during a Degraded Mode Write.

University of Cape Town

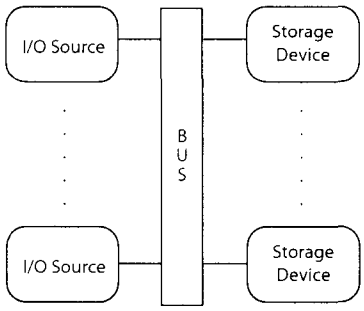


Figure 26: Model of an Enterprise Storage System, as used by RÖSTI

5.4.1 Defining the System Architecture

Our approach to creating the architectural system model focused on decomposing the simulator into a number of implementation independent components that function together to form the system. Each component was defined by the responsibilities it had, and was defined to be as generic as possible. This allows the user to create a desired configuration by composing a subset of the available components into an appropriate system architecture. This approach allows for the modular, flexible architecture that was one of the initial design goals. To illustrate the benefits of such an architecture, consider a user who chooses to simulate a configuration where a RAID 5 array with a large capacity is presented with a heavy, sequential workload. Alternately they may choose to simulate a RAID 6 array with limited capacity being presented with a light, random workload. In either case, the architecture we develop here should be able to support the required model.

We first developed a system model to support the above. This model maps well onto the functionally independent components referenced above. The high-level system model is constrained by our area of interest, namely Enterprise Storage Systems, and is presented in Section 5.4.2. Within this model, the functional requirements presented in 5.3.4 define areas of specific interest within the simulator. This information leads to an expansion of the RAID Controller sub-model, as presented in Section 5.4.3. Finally, we explore how the architecture we have developed can be applied to increasingly more complex systems in Section 5.4.4.

University of Cape Town

5.4.2 Storage System Model

The model presented in Figure 26 is illustrative of a typical Storage System arrangement, as initially presented by Patterson et. al. [PGK88] and later utilised by [Hol94, HGS93, UAM01, KS95, Pan95]. It is a relatively simple model, however it is capable of expressing fairly complex architectures[9]. Extending this model to represent Storage Area Networks (SAN) [GM00] or iSCSI [SSM+] is also possible by changing the model used for the bus.

The model consists of one or more I/O sources connected to a common bus. A number of Storage Device are also attached to this bus, each providing independent data storage. Each Storage Device is a self-contained entity, and may represent a physical disk drive or a RAID Storage device, as illustrated in Figure 27. The functions of each of these sub-models is covered in the following sections.

[9] See Section 5.4.4 (p. 68)

I/O source

An I/O source is an abstraction of a device which utilises the secondary storage provided by a Storage System. A common example of such a device would be a computer or workstation accessing a shared disk or, in our case, a Storage System (see below).

An I/O source is responsible for generating a stream of requests for submission to any of the available Storage Systems. The nature of these requests determine the workload submitted to the system. For example, a source could generate a stream of requests that accessed a sequential range of Logical Block Addresses (LBAs) to provide a sequential workload. Alternately, a source could replay a previously recorded trace of disk activity to simulate a particular real-world system of interest.

The I/O source is also responsible for accepting responses to these requests. These responses typically provide a response time for the requested operation, which can then be used to characterise the performance of the system.

Storage Device

The Storage Device is an abstraction of a physical device that provides secondary storage. A Storage Device can present multiple logical volumes to the connected I/O sources. Examples of Storage Devices are a simple disk drive or a RAID Array. In either case, a storage device must provide a range of Logical Block Addresses (LBAs) which can be used to access stored data. Additionally, it must accept I/O Requests for data, where each request is characterised by:

- LBA: The address of the data.
- Size: The amount of data requested.
- Operation: Determines whether the request is to read (fetch) or write (store) the specified data.

Given a request with the above information, any Storage Device must provide a response that contains the time taken to perform the requested operation. This time is determined by the internal simulation of processing overhead, which might encompass: the time required to locate the data; the time required to transfer the data; the time required to process the data and calculate protection information; and the time required to recover lost data.

RAID Storage Device

A RAID Storage Device is a specific type of Storage Device that provides the RAID functionality within our model. It comprises a RAID Controller connected to a common bus, to which is connected a number of Physical Storage Devices (see below). A RAID Storage Device has the same responsibilities as a generic Storage Device, and hence must accept the same requests, and provide the appropriate responses.

University of Cape Town

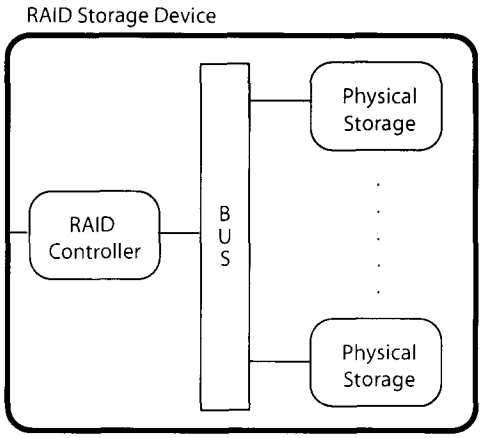


Figure 27: Model of RAID Storage, as used by RÖSTI

RAID Controller

The RAID Controller provides the RAID functions within a RAID Storage Device. It is responsible for intercepting and processing requests from I/O sources, issuing appropriate commands to the attached Storage Devices (which are usually Physical Storage Devices), and returning a response to the appropriate host. A more detailed model of this appears in Section 5.4.3 (p. 64).

Physical Storage Device

A Physical Storage Device is a specific type of Storage Device that represents a commodity hard disk drive. It is responsible for modelling such a device, using an approach similar to that outlined by Ruemmler et. al. [RW94].

Bus

The bus is an abstraction for the interconnection between devices. It may represent: a physical bus, such as SCSI; a device interface, such as IDE; or a communication channel, such as a network for iSCSI. In each case, this component is responsible for modelling the specifics of the exchange of data between two devices connected to this bus. These specifics may include delays due to transmission, contention, bandwidth limitations, data loss or other appropriate factors. Since the bus is modelled independently, this allows the accuracy of the model used to be changed without affecting other components. Thus, it is possible to use a bus with an underlying network model to simulate the operation of an iSCSI RAID array.

5.4.3 RAID Controller Model

Common implementations of RAID Controllers vary from software based schemes (Linux Software RAID [Bro05]) through hybrid schemes (Intel RAIDIOS Scheme [Poo02]) to pure hardware implementations (IBM Shark [IBM05]). It is necessary to develop a generic model of the operation of the RAID Controller which captures its core functionality to describe this range of

University of Cape Town

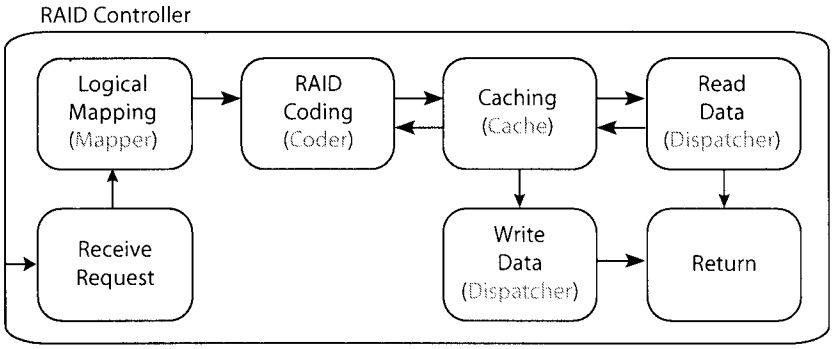


Figure 28: Model of the operation of a RAID Controller. Each step is labelled according to its function, and the name of the associated component appears in brackets.

schemes. The model we developed appears in Figure 28. It covers each of the necessary steps in satisfying an I/O Request to a RAID array[10], as well as illustrating the logical progression between steps. As such it is representative of the logical operation of any controller, rather than the physical implementation of a particular controller. This model was developed by synthesising the common features of the functional requirements presented in Figures 22 through 25. Each of these operations (Normal/Degraded Read/Write) thus map well onto the resultant model.

As outlined in Section 2.2.4 (p. 8), there are numerous ways of placing data on disk for a given RAID level. Given this, it is useful to separate the mapping of data to disk and the RAID functions performed on the data. This allows for various mapping schemes to be used with a given coding scheme – for example, using RAID 5 coding with a normal rotated parity layout, or with a distributed parity layout. This feature is evident in the model in the use of separate mapping and coding steps. An explanation of each of the individual steps follows.

University of Cape Town

Receive Request

When an I/O Request is received, the RAID Controller must interpret the request to establish what is required. It should also check that the request is valid, and that the requested data is available in the array.

Logical Mapping (Mapper)

Once a request has been received and interpreted, the first step in satisfying that request is locating the data on disk. The layout of data on disk is established by the user when the array is initially configured. As such, we assume that this layout is fixed for the duration of simulation. Possible layouts are determined by the chosen RAID level and the parity location scheme, such as Fixed Position Parity (RAID 4), Rotated Parity (RAID 5) or Distributed Parity (see 2.2.4).

The chosen layout determines mapping of the logical address (LBA) referenced by the source to the actual logical address of the requested data on disk. The logical mapping step uses the predetermined layout information to create a mapping between the stripes, strips and chunks layout that a RAID scheme refers to, and the physical location of these items on disk. Once

[10]These steps were established from the Function Requirements specification in Section 5.3.4 (p. 58).

this mapping has been established, operations and requests can be performed using only the abstract notions of stripes, strips and chunks. The corresponding physical locations need only be referenced when a chunk is read from or written to disk. This abstraction allows the Coding phase of operation to proceed independently of the physical arrangement of data on disk. As already mentioned, it also allows for various mapping schemes to be used with a given coding scheme.

Since the Mapper is aware of the physical layout of disks, it further follows that it should be responsible for maintaining a list of unavailable chunks on each disk. These chunks may be unavailable due to a complete disk failure or a disk strip failure[11]. During the Logical Mapping step, it is then possible to specify whether the chunks that the requested data maps to are actually available. This information is used in the RAID Coding step to determine an appropriate action to recover these lost chunks when the array is operating in degraded mode.

RAID Coding (Coder)

Once the logical mapping step is complete, the RAID Controller has the location of all requested data and the associated parity. Using this information, it can then proceed to perform the relevant RAID coding on the requested data. This coding always consists of reconstructing the requested data from the various stripes, and may also consist of:

- Calculating new parity information for a write request in normal mode.
- Reconstructing lost data for a read request in degraded mode.
- Reconstructing lost data and calculating new parity information for a write request in degraded mode.

In practice, the operations required are dependent on the particular coding scheme in use. For instance, with RAID 5 coding:

- Calculating New Parity involves computing the XOR of both old and new data.
- Reconstructing Lost Data involves computing the XOR of the parity information with the surviving data to recover the lost data.

It is also the responsibility of the RAID Coding step to indicate which data is required to fulfill the above requirements. Specifically, not all available data may be required, or it may be more efficient to use certain combinations of data and parity information. This decision determines what data needs to be fetched from disk, and can influence the performance of the system (see 8.3.3).

As evident in Figure 28, the RAID Coding step may be invoked more than once for a given request. This occurs when fulfilling a request requires two or more phases, such as a write request in a RAID 5 array which has an initial phase to read data, followed by a phase to write data. Thus, the RAID Controller must maintain state information for each request regarding the phase of execution it is in, and correctly process each phase of such requests.

[11] A sequence of adjacent blocks on disk have become unreadable.

University of Cape Town

Caching (Cache)

In real-world systems, the cache is often tightly integrated with the rest of the system, and may serve a dual-purpose as a processing buffer for parity computations. By contrast, software RAID implementations do not have a specific cache, relying instead on general system memory, which is allocated as necessary. Each of these choices are implementation specific, and as such are an impediment to generalisation. Restricting our cache model to one particular implementation restricts the possible flexibility of the simulator.

With regards to caching, we determined the impact of different caching schemes on the entire system. We preferred to be able to determine the best caching scheme for a given RAID level, rather than how best to implement caching for a given RAID Controller. Since accessing cache memory is three orders of magnitude faster than accessing disks[12], it is more reasonable to model the cache independently of other components and investigate other, more exact models where a particular investigation requires it.

As such, our choice decoupled the cache from the operation of the controller, and placed it as an intermediate step between processing and requesting data. This is conceptually how a cache operates, and also allows for caching within a given RAID Controller to be optional. That is, removing caching affects neither the operation nor the flow of execution within the controller. This choice also allowed for different caching schemes to be used with a given controller, since the cache implementation is independent of other components. For a complete explanation of how RÖSTI implements caching, see Section 8.4 (p. 111).

During operation, the Cache monitors communication between the Coder and Dispatcher. When the Cache detects a request to read a chunk of data, it checks whether the chunk exists in the cache. If it does exist, the chunk is marked as read, and is not processed by the Dispatcher. When the cache detects a request to write a chunk of data, it updates the value if it exists in the cache. If it does not exist, the cache management algorithm[13] is used to destage sufficient pages to create room for the new chunk, which is then added to the cache. These operations are transparent to the Coder and the Dispatcher.

Reading and Writing Data (Dispatcher)

This step encapsulates the interaction between the RAID Controller and attached Storage Devices, and is conceptually performed by a Dispatcher. The Dispatcher is responsible for dispatching I/O Requests to attached Storage Devices, and receiving and collating the associated responses.

When the RAID Controller needs to read or write data, it requests a block of chunks (Block) on various attached devices. The Dispatcher then constructs appropriate I/O Requests for each of these chunks. These requests are represented in exactly the same way as I/O Requests presented to the RAID Controller, and are then dispatched to the appropriate devices.

The Dispatcher maintains a list of outstanding requests, as well as the associated Block. Each such Block is associated with a particular I/O Request received by the RAID Controller. When the Dispatcher receives a response from an attached device, it matches this response against those that are outstanding. When all responses for a Block are received, the Dispatcher informs the

[12] RAM has average access speeds around 10ns whereas the figure for disk drives is 10ms.
[13] See Section 2.5.1 (p. 18)

University of Cape Town

Coder, which can then continue processing the data (in the case of multi-phase operations) or send an I/O Response to the appropriate I/O source.

5.4.4 Representing Complex Systems

We have thus far presented a relatively simple model of a RAID Storage System. While this is representative of a large number of commercial products, there is still the need to provide for more complex implementations. Obvious examples of this are RAID 1+0 and RAID 0+1. Both of these implementations use two levels of RAID, with the upper levels using the lower levels as single drives. Another notable example is the Shark Enterprise Storage Systems[14], which uses multiple underlying RAID arrays to provide a large pool of storage.

In attempting to represent the above architectures in RÖSTI, we wish to reuse as much of the available functionality as possible, so as to limit the amount of work required. One way of achieving this exploits our choice to use the same representation for any I/O Request within our model. As such, a request from a Source to a RAID Controller has exactly the same format as a request from a RAID Controller to a disk. It is thus possible for one RAID Controller to submit an I/O Request to a Storage Device, which happens to be another RAID Controller. This ability allows us to represent complex, multi-level architectures such as RAID 1+0 and RAID 0+1, which use the same communication paradigm. There are two equivalent, compatible methods we can use to model these complex systems: creating model Hierarchies, which are similar to tree structures; and using Nesting, which is analogous to Recursion. These two approaches are outlined below.

University of Cape Town

Hierarchies

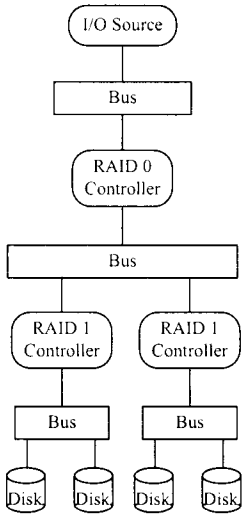


Figure 29: Example Hierarchical representation for a RAID 0+1 Storage System

As mentioned above, a Hierarchy is similar in concept to tree data structures. An example of a hierarchical representation of a RAID 0+1 configuration appears in Figure 29. A Hierarchy is a direct representation of the structure the system. It consists of a number of levels, with each level directly controlling the operation of the level below. In the diagram, this amounts to a

[14] Details can be found at `http://www-03.ibm.com/servers/storage/disk/ess/`

RAID 0 Controller issuing instructions to a number of RAID 1 Controllers, each of which issues instructions to attached disk drives. An implicit, but not obvious, feature of this approach is that each layer communicates only with the layers directly above and below it. This is similar to the interaction between different layers of the TCP/IP stack.

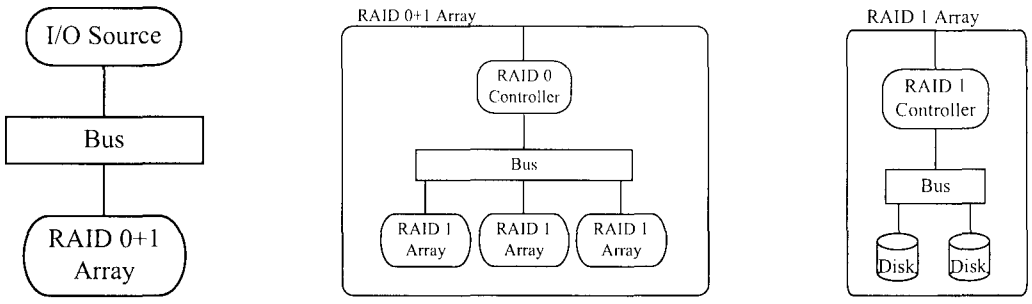
Using Hierarchies has the important benefit of showing the entire system model in a single diagram. Additionally, the connections between each module are explicitly shown. However, for complex hierarchies, the model quickly grows large and unwieldy. Additionally, because each connection must be explicitly created, it is not possible to exploit recurring patterns in the model. For example, each RAID 1 controller, together with attached disks, is a sub-system that occurs 3 times within the model in Figure 29. However, it is not possibly to reuse this model, since each such sub-system must be separately instantiated.

Nesting

An alternate approach to representing complex systems uses the idea of nesting. An example of a Nested representation of the same RAID 0+1 configuration appears in Figure 30. Nesting uses the same principles as recursion to reuse recurring portions of a model. The idea involves encapsulating the relevant components into a sub-model, and then using the sub-model in place of these components. This allows us to reuse the behaviour of the various components, without explicitly representing them.

As in the example figure, each level of complexity is represented by a separate sub-model. Each sub-model typically consists of a RAID Controller and attached Storage Devices. As per the System Model we have previously developed, each of these Storage Devices can either be a RAID Storage Device, which is a nested sub-model, or a Physical Storage Device, which represents a physical disk drive and terminates the recursion. We are thus able to reuse the concept of a RAID 1 Storage Device without having to explicitly represent all the relevant components.

This approach provides the extensibility that is a core design goal of RÖSTI. It also models real-world architectures, where layers of complexity are hidden behind standard interfaces. One notable disadvantage of this approach is that for complex systems, the system model is not immediately visible. However, every Nested system model can be converted to an equivalent Hierarchical system model, while the reverse is not possible. Thus a Nested model can be expanded to show the entire System Model at a glance.



(a) Top-level view of the system

(b) View inside the RAID 0+1 array

(c) View inside the RAID 1 array

Figure 30: Example Nested representation for a RAID 0+1 Storage System

University of Cape Town

5.5 Design and Implementation

In this section, we consider the implementation of our system architecture in OMNet++. We discuss how best to utilise the features that OMNet++ provides in Section 5.5.1. These include: encapsulating messages; visualising the execution of the simulation; promoting code re-use; and collecting statistical data. Given the modular nature of both the system model and OMNet++ programming environment, it is also essential to define a communication protocol. This protocol will govern communication between the various modules of the system and is covered in Section 5.5.2.

5.5.1 Exploiting OMNet++ Facilities

Besides being a comprehensive event-driven simulation library, OMNet++ offers a number of useful features that simplify the implementation of RÖSTI. The most interesting of these features are discussed below.

University of Cape Town

Encapsulating Messages

OMNet++ is a simulation environment that uses messages to communicate between various modules. Thus a RAID Controller would receive a message requesting a read operation. In processing this request, the modules within the Controller[15] will need to communicate via messages in order to satisfy the request. We would like to associate all such communication with the associated I/O request, so that the details of the request are always available without having to duplicate the information. OMNet++ provides for this by allowing message encapsulation, which amounts to attaching a secondary message to a primary message as an extra data element. It is thus possible to layer messages, with the inner message representing the original request, and the outer messages representing subsequent requests. This is illustrated in Figure 31.

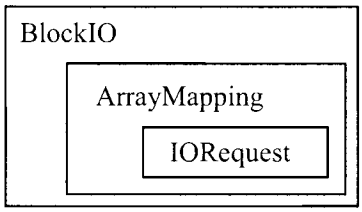


Figure 31: Encapsulation of messages, which maintains an association between related communication.

Visualising Simulation Execution

While able to run in a conventional command-line mode, OMNet++ also provides the ability to run a simulation with a Graphical User Interface (GUI). This interface displays the individual modules in the simulation, as well as the inter-connections between them. The user is able to control the execution of the simulation, as well as observe the operation of the simulation. The GUI is also useful for debugging and validating the simulation, as it shows the flow of messages

[15] See Section 5.4.3 (p. 64)

between modules, as well as allowing the user to inspect the internal state of the simulation. One can thus visually trace the execution of a simulation, and stop the simulation should an unexpected event occur. The state of the faulty module could then be inspected to determine the cause of the error, and make appropriate changes to the code. OMNet++ also allows the user to change the internal state of the simulation for testing purposes.

In order to accomplish the above, the simulator code must be slightly modified. OMNet++ requires that all messages derive from a base `cMessage` class to support visualisation of these messages. Rather than derive messages explicitly, however, OMNet++ provides a formal description language for messages that is fairly similar to C++. These descriptions are then translated into appropriate C++, which can be referenced and used in code. We have used this method for all the messages used in RÖSTI, allowing these messages to be animated and inspected within the GUI. The definitions of a subset of the messages used in RÖSTI appear in Appendix B; the C++ convention of using // to indicate comments is used in the examples.

It is necessary to use an OMNet++ specific data type, called `cPar`, to allow inspection of the simulation state. This structure can hold data of any type, and any variables within code that are declared to be of this type can be inspected and altered from the GUI. However, there are overhead issues associated with this type[16], so we have limited the number of these inspectable values in RÖSTI. However, they have still played an important role during the development of RÖSTI, specifically in debugging the simulator during development.

Collecting Statistical Data

Due to the fact that each module of RÖSTI is designed to be an independent entity, it is necessary to take a different approach to statistics collection. A centralised approach will not work, as there are a myriad of possible configurations that can be simulated. At the same time, we do want all statistics from a given simulation to be grouped together. Our approach places the responsibility of statistics calculation on the individual modules. This is reasonable, since the implementor of each module is best aware of the statistics of interest for that module. For instance, an I/O Source that generates a workload from a trace file may be interested in reporting the average inter-arrival time between requests, whereas that statistic is redundant for a source that generates a random workload with a specified inter-arrival time between requests.

In order to group the statistics, we use the facility of *Scalars* and *Vectors* provided by OMNet++. A Scalar is simply a statistic that is recorded at the end of a simulation execution, such as the average response time of the system. A Vector represents a series of values recorded over the duration of the simulation, such as the list of queueing times experienced by requests at a given disk. In both cases, OMNet++ provides function calls that any module can use to store either a Vector or Scalar value. These values from every module are then stored in common output files (one for Vectors and another for Scalars). Thus, our distributed statistical calculation approach still allows for centralised collection and analysis of this data.

[16] see Section 5.7.1 (p. 80)

University of Cape Town

Promoting Code Reuse

OMNet++ is an inherently Object Orientated (OO) software library, and as such it encourages the reuse of code. Particularly, modules are objects which can have any base class. We have used this fact to extract the common operations from sets of related modules, and encapsulated this within an abstract base class.

For instance, for any given I/O Source the steps involved in generating a request are the same, with the exception of how the individual fields are filled. So we have a base class that handles creating an **IORequest** message, dispatching that message, receiving the corresponding **IOResponse** message, and recording statistics related to that request. The task of determining the logical volume, LBA, size, and required operation is handled by subclasses. Adopting this approach allows modules to concentrate on their particular function, by abstracting the majority of the housekeeping that would otherwise be required.

5.5.2 Defining Communications Protocols

For our purposes, the communication we define consists of messages and interfaces. A message represents a simplex communication between two modules. It may represent a request for the recipient to perform some function, or a response to such a request. A message is defined by its type and its format. The message type defines its specific function, such as a request for an I/O operation. The message format contains the data necessary for the recipient module to perform the requested function. For an I/O operation, the message would contain the address and size of the requested data and the required operation.

Every module in our system must implement at least one interface. An interface specifies a set of request messages that the implementing module must accept and a corresponding set of messages that it should generate in response to requests. The advantage this offers is that any module that correctly implements a given interface can be used within the simulator without altering other components. For example, to add an additional disk simulation component, one need only develop a module that correctly implements the Storage Device interface. This module can then be used in a simulated RAID array without altering the code for either the RAID Controller or the I/O Source.

An additional benefit of using interfaces is that it supports the Nesting approach outlined in Section 5.4.4 (p. 69). By ensuring that both RAID Controller and Disk Drive modules implement the Storage Device interface, it is possible to seamlessly replace Disk Drives with a RAID Array. This allows more complicated architectures to be simulated, such as the RAID 0+1 array in Figure 30 (p. 69). The various interfaces used in RÖSTI appear below. Each of the messages referenced below were implemented as OMNet++ messages, the definitions of which appear in Appendix B (p. 135).

I/O Source Interface

The I/O Source interface governs generating I/O Requests. This interface is typically implemented by I/O Source modules. It is also implemented by RAID Controllers, in order to submit requests to attached Storage Devices. This interface stipulates that the module must generate **IORequest**

University of Cape Town

messages, and accept corresponding **IOResponse** messages.

Storage Device Interface

The Storage Device interface is the corollary to the I/O Source Interface. It governs receiving and satisfying I/O Requests. This interface is typically implemented by Physical Storage Devices, such as disk drives. It is also implemented by RAID Controllers, in order to receive requests from I/O Sources and other RAID Controllers, via the I/O Source Interface. This interface stipulates that the implementing module must accept **IORequest** messages and respond with corresponding **IOResponse** messages.

Mapper Interface

The Mapper Interface governs the operation of Mapper modules. It stipulates that implementing modules must accept **IORequest** messages and generate a corresponding **ArrayMapping** message. This **ArrayMapping** message specifies the layout of requested data on disk, as well as the availability of this data.

Coder Interface

The Coder Interface governs the operation of Coder Modules. It stipulates that implementing modules must accept **ArrayMapping** messages. In response, the modules must generate **BlockIO** messages, which indicate what operations to perform on marked blocks within the encapsulated **ArrayMapping** message. The **ArrayMapping** message is encapsulated in the **BlockIO** message, as illustrated in Figure 31.

The Coder must also accept **BlockIOResponse** messages, indicating which of the requested operations on marked chunks were successful. Using this information, the Coder can generate either a **IOResponse** message, indicating whether the requested operation completed successfully, or another **BlockIO** message. For a RAID 5 write, this **BlockIO** would represent the request to write new data and parity after having read old data and recomputed parity.

Cache Interface

The Cache Interface governs the operation of Cache Modules. It stipulates that modules using it must accept **BlockIO** messages. The cache is responsible for examining the list of requested chunks in the encapsulated **ArrayMapping**: if a read of the chunk is requested, the chunk should be marked as such; if a write of the chunk is requested, the cache should update its internal state appropriately. Once this processing is complete, the Cache should pass the modified message on to the Dispatcher.

Implementing modules must also accept **BlockIOResponse** messages. The Cache must examine the message and update its internal state whenever the message indicates that chunks have been successfully read. Once this processing is complete, the Cache should pass the modified message to the Coder.

University of Cape Town

5.6 Functionality

The System Model we have described for RÖSTI allows for a number of different implementations for each of the described components. This section outlines the range of components that have thus far been implemented, as well as the extent of their functionality. For the purposes of describing collected statistics in this section, the notation (MMMS) indicates that the Minimum, Maximum, Mean and Standard Deviation for the corresponding statistic have been calculated.

In addition, where the phrase "chosen distribution" appears, this indicates that any random distribution from those listed in Table 6 (p. 138) can be used to describe the associated parameter. The table also lists the required parameters for each distribution.

5.6.1 I/O Sources

As per our System Model, I/O Sources generate a workload that is presented to Storage Devices in the system. We have implemented a number of Sources, some producing synthetic workloads and others that use trace data from real systems. In each summary below, the nature of the generated workload is described, as well as the configurable parameters that determine the output of the source. All I/O Sources record the following statistics:

- Number of Requests generated.
- System Response Time for completing requests (MMMS).
- Inter-Arrival Time between requests (MMMS).

Random Source

The Random Source generates I/O Requests, with the parameters given in Table 1 (p. 74). Generated requests are distributed across all available Logical Volumes, and all distributions are synthetic. The Random Source generates a purely synthetic workload, distributed across all available logical volumes, described by the parameters listed in Table 1 (p. 74).

Volume	Logical Volume for which workload will be generated.
LBA	Chosen distribution ranging from 0 to the maximum addressable LBA for the above logical volume.
Size	The size of a given request, drawn from a chosen distribution.
ReadRatio	Read ratio, specifies what percentage of requests should be reads.
InterArrivalTime	Time between generation of requests, drawn from a chosen distribution.

Table 1: Configurable Parameters for the Random Source

University of Cape Town

Hotspot Source

This source is able to describe workloads similar to those in OLTP [17] systems, which consist of mixed reads and writes to random locations on disk, as well as multiple requests to a small number of locations that represent often used information. This source is similar to the Random Source and uses the same parameters listed in Table 1 (p. 74). In addition, a number of Hotspots can be chosen and the frequency with which requests will target one of these Hotspots can be specified. The location of the hotspots are drawn from a uniform random distribution ranging from 0 to the maximum addressable LBA. The additional configurable parameters are listed in Table 2 (p. 75).

NumHotspots	Specifies the number of hotspots to simulate.
HotspotFrequency	The percentage of requests that should target one of the specified hotspots.

Table 2: Configurable Parameters for the Hotspot Source

Sequential Source

The Sequential Source generates a purely synthetic workload covering one specified logical volume. The workload is sequential, in that it starts at a specified LBA, and proceeds to issue requests of varying sizes, with each subsequent request LBA starting at the end of the previous request. Each request generated is of a single specified type, thus representing scientific computing workloads which consist of large batches of reads or writes.

Volume	Logical Volume for which workload will be generated.
LBA	Starting LBA from which the workload will begin.
Size	The size of a given request, drawn from a chosen distribution.
Opcode	Decide whether workload will consist of reads or writes.
InterArrivalTime	Time between generation of requests, drawn from a chosen distribution.
NumRequests	Number of requests in sequence to generate.

Table 3: Configurable Parameters for the Sequential Source

Trace Driven Sources

Trace Driven Sources encompass a number of types that share one common attribute: they all generate workloads from files that record traces of I/O operations in real systems. Since their operation is so similar, they all share a common implementation and are configured by a single parameter, the name of the trace file. Each of these sources differ in the format of the trace files they can read. Currently only the SPC Trace Source is supported.

[17] See Section 3.3.3 (p. 27)

University of Cape Town

The SPC Trace Source reads input from a file in which the fields are comma separated, and are in the order and format specified by the SPC Benchmark, i.e.:

```
Volume, LBA, Size, Opcode, Timestamp
```

where

1. **Volume, LBA** and **Size** are *integer* values
2. **Opcode** is one of *('R', 'W')*
3. **Timestamp** is a *real* value.

5.6.2 Disks

The implemented disk models represent the Physical Storage components in our System Model. As will be mentioned in Section 5.7.2 (p. 80), these models have been adapted from freely available source code, rather than attempting a full implementation.

All disk modules have the capability to fail at any time during the execution of the simulation. When run in GUI mode, ROSTI is interactive and the user can stop the simulation, fail a disk[18], then continue from where it was stopped. It is also possible to specify a Mean Time to Failure (MTTF), and each disk will then automatically fail at a time drawn from an exponential distribution, with mean specified by the MTTF.

The list of integrated disks is given below, and all disks record the following statistics:

- Number of Requests receieved
- Request Queueing Time (MMMS)
- Request Service Time (MMMS)
- Inter-Arrival Time between requests (MMMS)

Simple Disk

This disk module simply serves incoming *IORequests* with an exponentially distributed response time. The mean of this response time is draw from a chosen distribution.

DiskSim_Disk

This module integrates the functionality of the DiskSim simulator into RÖSTI as a slave program. This module allows the simulation of a single *IBM18ES* disk drive. The specifications for this drive layout are separately specified, as per the DiskSim method, in a file called *ibm18es.parv*. Any of the available DiskSim disks can be used by appropriately modifying this file.

[18]This failure can be initiated by the user by changing the **Failed** parameter of the relevant disk module.

University of Cape Town

DiskSim_Array

This module extends the above by allowing the simulation of an array of 8, independent *IBM18ES* drives. The specifications for this layout are contained in the file *ibm18es_array.parv*.

IBMu146Z10 Disk

This module incorporates work by Dr. Xiao-Yu Hu from IBM Research Zürich. It specifies the behaviour of an IBMu146Z10 3.5 inch SCSI 10k RPM hard disk drive using a mathematical model of a disk drive adapted from the work of [RW94]. The three components of the drive response time are modeled as follows:

Seek Time The seek time s_t is given by:

$$s_t = \begin{cases} 0 & n = 0 \\ 500 + 22.6\sqrt{n} + 0.15431.n & \text{read} \\ 500 + 45.3\sqrt{n} + 0.063095.n & \text{write} \end{cases} \tag{7}$$

where n represents the distance, in number of cylinders, between the current head position and the target head position.

Rotational Delay The rotational delay r_t is given by:

$$r_t = s_t - (ss_r + cs_r + hs_r) \tag{8}$$

where r_t is the rotational delay, ss_r is the time to rotate to the start sector, cs_r is the time incurred due to a cylinder switch, hs_r is the time incurred due to a head switch.

Transfer Time The transfer time f_t is given by:

$$f_t = st_f + cs_f + hs_f \tag{9}$$

Notes regarding this implementation:

- The time to read sectors of the disk st_f is calculated as a fraction of the total time to read all sectors of the current track. Since sectors per track is zone dependent, the case where a request crosses a zone boundary is not considered.
- In contrast to the rotation delay above, each cylinder switch cs_f and head switch hs_f incur a fixed delay.
- The number of cylinder switches Δc is given by the end cylinder less the start cylinder.
- The number of head switches is given by the number of tracks Δt accessed. This is calculated in two ways:

$$\Delta t = \begin{cases} \text{end.track} - \text{start.track} - \Delta c & \text{All tracks in same zone} \\ \frac{\text{sectors}}{\text{sectors/track}_i} + 1 - \Delta c & \text{Tracks cross zones} \end{cases} \tag{10}$$

University of Cape Town

5.6.3 The RAIDController

As outlined in Section 5.4.3 (p. 64), the function of the RAID Controller is performed by a number of independent components. The combination of these components determines the functionality provided by the Controller. We therefore present here the implemented features by the associated components.

The Mapper and Coder

The combination of a particular Mapper with a compatible coder module determines the RAID functionality available. The functionality of the Mapper is determined by the types of codes it supports. Our current implementation provides two mapping modules for Distance 2[19] and Distance 3[20] codes. In addition, each Mapper version has support for SPIDRE, the RAID technology developed by IBM described in Section 2.3.6 (p. 13). There is also a Mapper provided for RAID 0 mapping, which simply stripes data across disks.

Together with these Mapper modules, we have implemented five Coding modules to perform the various RAID functions. These modules implement: RAID 5; RAID 5 + SPIDRE; RAID 6; RAID 6 + SPIDRE; RAID 0. Each module is able to operate in normal or degraded mode.

In order to use a particular RAID level, it is necessary to combine the correct pair of Mapper and Coder. The required combinations are illustrated in Table 4 (p. 78).

RAID Level	Mapper	Coder
RAID 4	Distance 2 without Parity Rotation	RAID 5
RAID 5	Distance 2	RAID 5
RAID 5 + SPIDRE	Distance 2	RAID 5 + SPIDRE
RAID 6	Distance 3	RAID 6
RAID 6 + SPIDRE	Distance 3	RAID 6 + SPIDRE

Table 4: The various RAID levels available in RÖSTI, together with the required Mapper/Coder combinations.

University of Cape Town

[19] Codes with 1 parity blocks per row, such as RAID 4, RAID 5.
[20] Codes with 2 parity blocks per row, such as RAID 6, EvenOdd

Cache

The cache intercepts messages to the **Dispatcher** and handles them according to the required operation.

- If it is a read operation, the cache contents are checked to determine if the required data can be found. If it is found, the data is returned, the cache page is updated with access information, and the corresponding disk request is cancelled.
- If it is a write operation, the data is written into the cache. If the data is already in cache, the value is updated. Access information for the cache page is updated and the corresponding disk request is passed on for processing.

If the cache becomes full, space needs to be freed. Three policies have been implemented to handle this[21]:

- **LRU**: Each cache page possesses a timestamp, which indicates the time of the last access to the corresponding data. When a page needs to be freed, the page with the oldest time (the *Least Recently Used* page) is freed.
- **LFU**: Each cache pages additionally possesses a frequency counter, indicating the number of accesses to each page in the cache. The page with the smallest frequency count (Least Frequently Used) is freed, since it has been used the least of all pages in cache.
- **ARC**: An advanced caching algorithm developed by IBM. A detailed description can be found in Section 2.5.1 (p. 20).

5.6.4 Interconnections

Interconnections are represented in RÖSTI using OMNet++ channels, which are queue-like message delivery mechanisms with the following properties:

- **Propagation delay**: The propagation delay models the latency involved in traversing a connection. For most local I/O connections (SCSI, IDE, SATA, etc.), this value is negligible given the proximity of the physical disk drive to the RAID Controller. For remote connections, such as iSCSI, this delay is noticeably longer as I/O requests must be routed across an external network. For the purposes of RÖSTI, which currently focuses on local connections, this delay is set to a minimal value.
- **Bit Error Rate**: The probability that a bit is corrupted during transmission. For modern error-correcting I/O connections, this value is assumed to be 0.
- **Data Rate**: The data rate models the time taken to transmit a message (or equivalently an I/O request) across a connection. It is specified in *bits/sec*, and is equivalent to the bandwidth of the corresponding connection. This is of importance when large requests are transmitted, which cause corresponding transmission delays. For RÖSTI, we have taken the maximum theoretical bandwidth of each modern I/O connection.

[21] For details see Section 2.5.1 (p. 18)

University of Cape Town

5.7 Implementation Issues

During the implementation of RÖSTI, we encountered a number of difficulties: some were due to limitations of the external software we used (Sections 5.7.1, 5.7.2 and 5.7.3); others to problems encountered in fulfilling the user requirements (Sections 5.7.4, 5.7.5 and 5.7.6). These difficulties and the corresponding solutions are discussed below.

5.7.1 Visualisation Inefficiencies

As mentioned in Section 5.5.1 (p. 70), OMNet++ provides a GUI that allows the user to inspect the state of an executing simulation, which can provide a useful debugging aid. While developing the cache simulation module, we attempted to use this ability to monitor the contents of the cache during simulation execution. This would allow us to visually track the movement of pages in the cache in order to test whether the caching scheme in use was working correctly.

In order to achieve this, each page of the cache was represented as an object deriving from the OMNet++ class `cObject`, and the cache itself served as a container for these objects. Doing this caused OMNet++ to generate TCL/TK[22] code that allowed the user to inspect the state of the cache during simulation execution. Moreover, this state was dynamically updated as changes to the cache occurred. As a debugging tool, this proved very useful, particularly in early stages of implementation.

The inefficiency alluded to above was encountered once full simulation runs were attempted. We noticed that for small cache sizes with approximately 10 000 pages, enabling caching had a minimal performance impact on the simulation execution time. However, as the cache size grew the execution time increased almost linearly, while an implementation using normal C++ arrays scaled significantly better[23]. Operating under the assumption that the poor performance was due to the TCL/TK code being generated for the GUI, we performed the same comparison test using the command-line version of the simulator. Surprisingly, the results of this comparison showed the same linear scaling for the visualised cache version.

Further investigation revealed that the TCL/TK generated code was executed irrespective of the mode (GUI or command-line) the simulator was run in. Given this and the poor performance of the cache with visualisation support, we were forced to utilise native C++ arrays and structs to represent the state of the cache. This meant that we were now unable to debug and test the cache within the OMNet++ GUI. Given the important role of caching within our system model, this presented a large problem. The solution we arrived at involved developing an external tool to both test and validate the operation of the various caches we implemented. The details of this tool are covered in Section 8.4 (p. 111).

5.7.2 Scarcity of Simulated Disk Implementations

RÖSTI is a simulation environment focused on RAID enabled storage systems. To that end, we had previously decided to focus development effort on those aspects directly related to RAID

[22] See Section 3.5.2 (p. 35)

[23] Although associative caches cannot be directly represented in software, we used hash-tables and free page lists to minimise the time searching the cache, hence a linear increase in execution time was unexpected.

University of Cape Town

(such as the controller and caching), and utilise external software models for the disk drives. Our first choice was DiskSim[24], which is a well-tested, comprehensive disk simulator with a number of available disk models. Unfortunately, we experienced a number of problems in trying to integrate DiskSim with RÖSTI[25]. In addition, the DIXTrac tool [SG00], which generates the physical parameters required to simulate a given disk drive to the DiskSim database, was unavailable. This meant that the available disk models were all outdated[26], and newer drives could not be added.

The problem of outdated disk drive models was something we encountered on a number of occasions. Most disk models that are available as part of a simulation environment [CGHZ96a, CP90, Wil96] are all integrated into other simulation environments, and our experience with DiskSim suggests that extracting these models would be equally difficult. The final problem we encountered was that the source code for some disk models referenced in published papers was unavailable due to commercial considerations [RW94, KTR94, CP90]. Thus, we were left with a small number of disk models that can be simulated efficiently[27]. This shortcoming will hopefully be addressed in future work with RÖSTI.

5.7.3 Encapsulating DiskSim

As mentioned previously, the DiskSim environment provides a number of comprehensive disk models that we wished to integrate with RÖSTI. The source code for DiskSim is available under an academic licence, allowing us to use it as a component of RÖSTI. However, given that DiskSim is a standalone simulation environment, we wished to extract just the disk models from the entire simulator.

Decomposing DiskSim

This is where we encountered out first problem. DiskSim is a monolithic system, written primarily in C, and as such is not well segmented into functional blocks. Moreover, the code is not well commented, and no documentation exists for it. As such, interpreting the purpose of various sections of code is difficult, and attempting to separate the system into independent components is even more so. We did manage to locate the code directly related to disk simulation, but this only made the tight integration of the system clearer, with explicit calls being made to core simulation functionality as well as other simulated components. The scale of the problem is evident in the comments left by the current maintainer of the code, who professes to "not understand" certain sections.

All the above meant that it was not possible to separate the disk model from DiskSim for use in RÖSTI. The alternate solution we eventually used was to run the full DiskSim simulation environment as a slave process under RÖSTI. This required that RÖSTI manually advance the DiskSim simulation time at a very fine interval. As one might expect, running a full simulation environment under another has a very heavy impact on performance. This expectation was realised when tests with RÖSTI and DiskSim working together were run. In these tests, RÖSTI was an order of magnitude slower than using a simple analytical disk model.

[24] Details on DiskSim appear in Section 4.3.1 (p. 45)

[25] See Section 5.7.3 (p. 81)

[26] The newest model represented was from 1999.

[27] While the DiskSim models can be used, the limitations their use imposes and the overhead involved makes it unsuitable for large simulation studies.

University of Cape Town

Our investigations revealed that the overhead of running DiskSim as a slave was not totally to blame, however. Equally problematic was that DiskSim required its internal time advanced in such small intervals that for a simple I/O Request, upwards of 50 calls to DiskSim were required for the request to complete. This in turn requires a similar number of calls to the OMNet++ backend to advance the RÖSTI simulation time at similar intervals. The net effect is an overall decrease in the performance of RÖSTI.

Porting DiskSim to Windows

Another issue we encountered was that DiskSim is not cross-platform, while RÖSTI is. Thus, although DiskSim uses standard C and compiles under Unix platforms, it produces errors when compiled under Windows. These compilation errors reference code deep within the simulation core, and as such we have been unable to make the necessary changes. The overall result of this is that while DiskSim can still be used under RÖSTI, its use is discouraged and limited to operation under Unix-based Systems.

5.7.4 Ensuring Optimality of Implementation

When processing a RAID request, there are a number of methods to fulfill that request. Although each method may be semantically correct, each may access a different number of disks. Consider the case of the small-write, as outlined by Courtright [Cou97]. This case occurs when a request is received to write a small number of blocks to disk, all of which are stored on a single disk in the array. The correct method to process this request is illustrated in Figure 32, where the corresponding old data from the disk is read together with the parity information. The new parity is then calculated using these two pieces of information, and new parity and data are written back to disk.

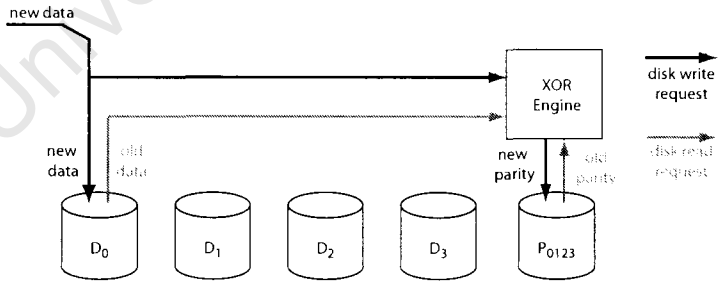


Figure 32: The small-write algorithm is used where a single disk in the array is written to.

An alternate method to process this request would read all old data ($D_1 \ldots D_3$) from the disks, together with old parity. New parity information would then be calculated using the new data (D_0), remaining old data ($D_1 \ldots D_3$) and the old parity. The new data, remaining old data and new parity would then all be written back to disk. Although this method is semantically correct, it is obviously inefficient in terms of number of disk accesses when compared to the first method.

A subtler example of this issue is illustrated in Figure 33. For this case, the optimal method is to read the remaining old data (D_3) and parity, compute the new parity and then write the new data ($D_0 \ldots D_2$) and parity to disk, as illustrated. However, it is also correct to read the corresponding old data ($D_0 \ldots D_2$) and parity. New parity would then be computed using the

University of Cape Town

old data, new data and old parity. The new data and parity would then be written to disk. The second approach is slightly more obvious, and though it produces the correct end result, it is again inefficient in terms of number of disk accesses compared to the first method.

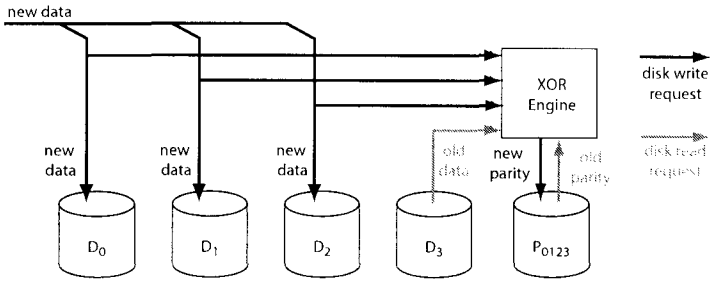


Figure 33: The reconstruct-write algorithm is used where more than half the disks in the array are being written to. The strips that are not being overwritten are read from disk.

These examples illustrate that ensuring that the RAID Controller is performing operations both correctly and efficiently can be problematic. This was a problem encountered during the implementation of RÖSTI, where the reconstruct-write algorithm was initially implemented using the second method above. The problem was only detected some time after the initial implementation, using the visualisation provided by the RÖSTI GUI interface. This initial problem prompted an investigation into the implementation of various other RAID operations, and more efficiency issues were found. In each case, although the implementation was correct in terms of the final state of the array, the number of disk accesses required was suboptimal.

A systematic search for these types of problems requires a knowledge of all the possible error cases. A request trace can be generated for each case that compares the actual disk accesses to the expected accesses. This was done for RÖSTI for each of the RAID 5 and RAID 6 Coder implementations, and the appropriate cases fixed. The exact cases used are covered in Section 8.3 (p. 97).

However, this process had to be tailored for each scheme, and there still existed the possibility of human error in ignoring certain cases to be tested for. This lack of a generic approach, and the possibility for error, led to the formulation of a scheme for formalising the operation of the RAID Controller. This scheme is outlined in Section A (p. 133), but has not been implemented due to time constraints. Should it prove successful, it would allow Controller implementations to be automatically verified for both correctness of operations and minimisation of required disk accesses.

5.7.5 Storing Parameters with Associated Results

OMNet++ provides a centralised method for collecting statistical output in common files, using the Scalars and Vectors facilities. Simulation configuration parameters are stored in a separate file, however, using a different format to the output files. Keeping these files linked is problematic, as is performing analyses on output statistics versus input parameters. The latter was an issue encountered during the implementation of the Analysis tool for RÖSTI[28].

Our solution stored all input parameters in the Scalars output file during initialisation. All

[28]See Chapter 7 (p. 91)

University of Cape Town

parameters were prefixed with the tag *parameter:*, to ensure they were not confused with actual statistical outputs. This approach ensured that related inputs and outputs were stored together, and it also simplified analysis of dependent output statistics versus independent input parameters in the Analysis tool.

5.7.6 Representing I/O Connections

Modern I/O buses such as IDE, SATA and SCSI support multiple devices connected to a single bus. Network connected I/O, such as iSCSI and Fibre Channel (FC) allow even more connected devices, and also support complex routing of requests that is transparent to the underlying hardware. Both types of interconnections support auto-detecting connected devices and indirect addressing of these devices by number or address. For RÖSTI, we would like to mirror this capability and also account for the fact that the RAID Controller operation is independent of the particulars of the underlying disk arrangement.

Our solution uses the idea of a bus manager which acts as a multiplexer/demultiplexer for requests between connected devices. This bus manager functions similarly to a DNS server and router on network. It allocates numeric identifiers to connected devices and routes I/O requests to the correct recipients. This allows the RAID Controller to address disks by index and disks to treat the RAID Controller as their sole host, which mirrors the way internal interconnects (IDE, SCSI) work. This scheme is also easily extensible to mimic the operation of network-based interconnects (iSCSI, FC), given that the concept of a routing and addressing mechanism is already present.

5.7.7 Dynamic Module Detection

Part of the modular design of RÖSTI creates the possibility that particular components of a given simulation can be seamlessly replaced by equivalent components that still adhere to the same interface[29]. This enables inter-operation between disks and RAID Controllers, neither requiring specific knowledge of the other, a principle similar to interface based software design.

Since all RAID Controllers and disks adhere to a common **Storage Device Interface**, it is possible to treat them in an identical manner. Hence when a **Storage Device** is connected to a bus, it is required to advertise itself and its capacity to the bus manager described above. When all devices have registered, the bus manager assigns each device a unique ID that is used for addressing devices on that bus.

The bus manager also informs all connected devices of the list of other connected devices on that bus, which allows any device on that bus to transparently send requests to any other device using only its ID. This also allows any RAID Controllers to dynamically determine the number of available storage devices it can use for striping, as well as their capacities.

[29] See Section 5.5.2 (p. 72)

University of Cape Town

Chapter 6

Configuring the simulation

University of Cape Town

In RÖSTI, as with most simulators, the largest part of user interaction with the simulator occurs during configuration: determining the component types required; describing the architecture to be simulated; and setting appropriate input parameter values for the various components. This chapter covers the steps we have taken to simplify these common tasks.

6.1 Motivation

As discussed in Section 5.3 (p. 53), one of the central requirements of RÖSTI was that it be easy to use. This is particularly important given our experiences with similar simulators[1], which have all proved to exhibit a steep learning curve.

RÖSTI offers a number of simulation options as well as a flexible system model, but to utilise them effectively, one needs to be well versed with the system implementation. As discussed in Section 5.3.1 (p. 54), there are a number of instances where using a simulator without such experience is necessary. We developed a configuration interface to RÖSTI (dubbed Configurator) to address this need. This interface enables simple configuration and execution of simulations, by:

- Allowing saving and loading of particular configurations, thus enabling sample configurations to be provided to illustrate how a particular architecture might be simulated.
- Providing meaningful descriptions of simulator parameters, thus ensuring the user is always aware of what values are required, what restrictions are imposed, and what changes will be made to the configuration.
- Constructing a GUI that reflects the way a simulation is configured, executed and then analysed[2], thus creating an interface that is intuitive in its operation.
- Hiding the complexities of the simulator implementation behind a GUI frontend.

There are also additional reasons for developing this configuration GUI, namely:

[1] See Chapter 4 (p. 39)
[2] Section 7 (p. 91)

6.1.1 Aiding Architecture Description

Describing storage architectures requires knowledge of both OMNet++ NED[3] and the available components in RÖSTI. Mistakes can be made whilst creating this textual description, and creating the required connections for large models can quickly become tedious. GNED is powerful tool that ships with OMNet++, allowing graphic manipulation of NED files. The tool is relatively simple to use, however it requires an understanding of what the various components in a simulation are and how they are intended to work together. It provides syntax checking of the generated NED descriptions, but is unable to perform semantic checking on the specified architectures. For complex architectures with many components, this semantic checking is a necessity. The configuration interface to RÖSTI provides this by using the system model we have defined and imposing a set of restrictions on what can be described.

6.1.2 Automating Multiple Executions

For statistically meaningful results, it is necessary to execute a given simulation multiple times, with varying random seeds. Doing this in OMNet++ requires multiple entries to be created in a configuration file, but doing this manually is time consuming and error prone. Moreover, it is not easily repeated for different configurations. By integrating this functionality into the configuration interface, we are able to provide a simple, reusable method to automate this process.

6.1.3 Simulating Over Parameter Ranges

When conducting simulation experiments, it is usual to investigate the behaviour of the system over a range of input parameter values. For instance, one might be interested in the behaviour of the system for request sizes in the range 4KB to 16KB, and cache sizes in the range 512KB to 2MB. With the configuration interface, we have provided a simple, extensible means of achieving this. This makes it possible to run a simulation experiment across multiple ranges of input parameters, with output from all executions stored in a common file. Performing such an experiment is thus simplified.

6.2 Approach

Our approach to developing the Configurator centred around three main aspects: the architectural restrictions on the possible RAID architectures supported by the Configurator that were necessary to make the task more manageable; the choice of OMNet++ environment and feedback to present to the user; and the provision of programming style variables to assist with conducting simulation experiments over ranges of input parameters. These considerations are discussed in more detail below.

[3] A language with syntax similar to C that describes the components in the system and the connections between them.

University of Cape Town

6.2.1 Setting Architectural Restrictions

Given the variety of architectures that can be specified using the system model we have defined, it was necessary to limit the possible architectures that the Configurator would allow. In particular, the recursive nature of some possible architectural descriptions proved problematic to cater for. We have thus limited the scope of systems that can be configured to those that fit the model illustrated in Figure 34, which is derived from our original model.

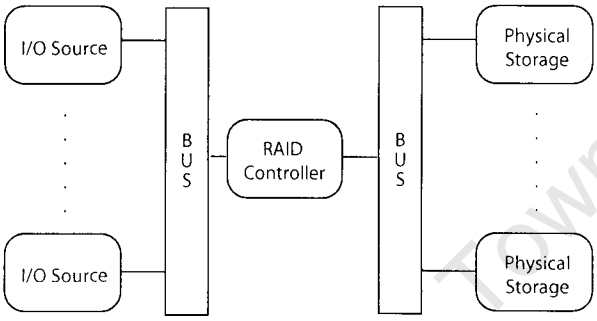


Figure 34: Restricted system model used by the Configurator.

This modified model is somewhat simpler than the original, but is still sufficiently expressive to allow many RAID configurations to be described. The obvious exception is complex systems with multiple layers of RAID Controllers[4], however such architectures can still be configured using the GNED tool.

6.2.2 Interacting with the Simulation

As covered in Section 3.5.3 (p. 37), OMNet++ offers two methods of monitoring the operation of a simulation. Command line mode is a non-interactive, text-based interface that allows the execution of batches of simulations from a single command. This mode also details the progress of a simulation through various output messages. Given that our aim is simplification of use, most of this feedback is unnecessary for general use, where the user is interested specifically in the outcome of the simulation. This mode is thus ideally suited to the case where multiple simulations must be run without user intervention. The feedback provided is filtered to provide a progress bar to the user, indicating the number of outstanding simulation executions. This is the default method used when a user chooses to execute a configuration using the Configurator.

6.2.3 Providing Parameter Variables

As stated above, simulating over parameter ranges is a common simulation activity. We have introduced the idea of variables to the Configurator to support such simulations. In our context, a variable consists of a finite set of typed values. Variables can be assigned to input parameters with the same type. For each execution of a simulation, one member of this set is selected and the associated input parameters are given this value. For example, we might define a variable whose values are $\{10, 20, 30, 40, 50\}$. If we assign this variable to the request size input parameter, the

[4]See Figure 30 (p. 69).

University of Cape Town

Configurator will generate 5 simulation executions. For each of these executions, the request size will be a different value from the variable set.

We have implemented 3 different variable types, but more can easily be added. These types are:

- **Integer Range**: Given minimum, maximum and step values, this variable covers all integers in the set $\{i \in \mathbb{Z} : (min \leq i \leq max), i = min + j \times step, j \in \mathbb{Z}\}$.
- **Double Range**: As above, but minimum, maximum and step are real values.
- **Ratio Range**: Same as Double Range, but minimum = 0 and maximum = 1.

6.3 Implementation

The Configurator described above was implemented in C# using a tab based interface, as illustrated in Figure 35.

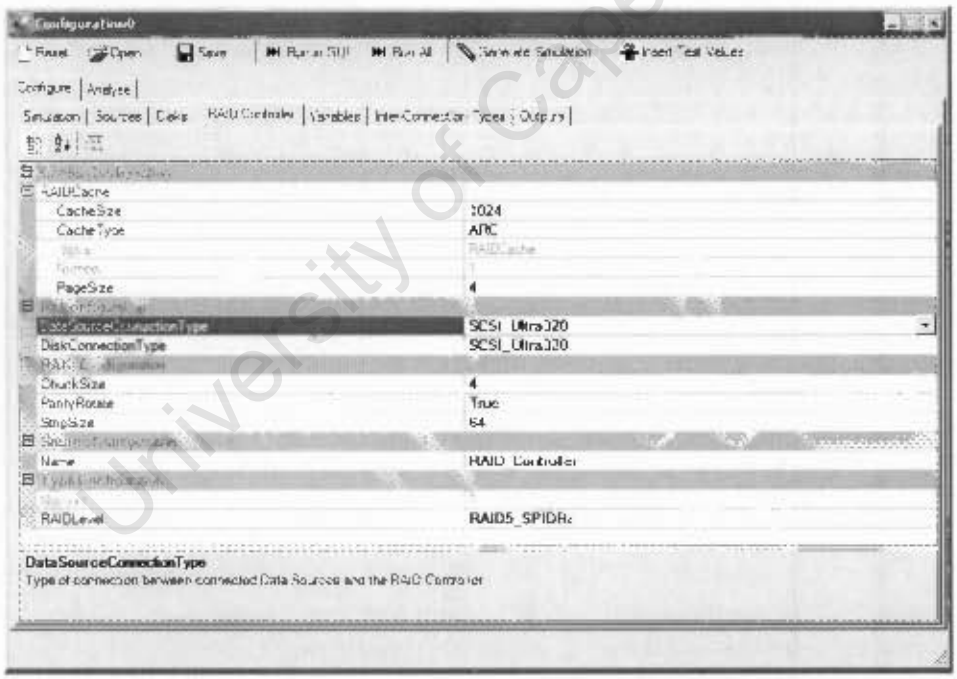


Figure 35: Screenshot of the Configurator utility for ROSTI

6.3.1 The Choice Of C#

C# is a Microsoft .NET language that is ideally suited to rapid GUI development. It encompasses an extensive visual control library, as well as complete API documentation. C# targets the Microsoft .NET Framework, and as such runs in a managed environment. In such an environment, memory allocation and disposal is automatically handled by a garbage collector, which greatly reduces the possibilities for memory related program errors. C# is natively supported by the Visual Studio 2003 IDE, an integrated GUI development environment that is better suited to

University of Cape Town

GUI Rapid Application Development (RAD) than Unix/C++ alternatives. All the above reasons make C# an ideal language for the implementation of the Configurator.

The configuration of OMNet++-based simulations is primarily accomplished through editing text files. A GUI configuration front-end is thus only necessary where the associated complexity of the text file editing is unacceptable for the target user. We have take the position that Unix/GNU Linux users are assumed to be technically knowledgeable, given that text-filed editing is the predominant means of configuration within this Operating System.

In contrast, Microsoft Windows users are accustomed to GUI interaction with and configuration of applications. As such, we have assumed the Configurator would be of primary use to this class of users. Restricting the development platform for the Configurator to Windows is thus a reasonable decision, especially given that the .NET Platform is in the process of being ported to Linux via the Mono project [dIJ02]. Once complete, this should allow the Configurator to be compiled and used seamlessly under Linux.

6.3.2 Allowing for Extensibility

Given that RÖSTI is extensible by means of additional modules, it is also necessary to provide some mechanism to extend the Configurator to accurately reflect these RÖSTI extensions. This is achieved by representing RÖSTI modules as C# classes with **Public Properties** that represent configurable aspects of the given module. These Public Properties are then examined at run-time by a **PropertyGrid** control, which displays all the Properties of a class and allows the user to alter them.

We have chosen to use the notion of C# **Attributes** to store descriptions of each Public Property in code alongside the Property, which ensures that the C# classes are self-documenting. We have also used C# base classes to determine in which section of the GUI RÖSTI components should appear. The Configurator is thus extensible by implementing a C# class that derives from the correct base class for each additional RÖSTI module.

Representing a RÖSTI module in code for use within the Configurator provides us with the important ability to customise how properties are edited in the PropertyGrid control. For instance, we can provide a drop-down list where selecting from a pre-defined list of options is required, or a slider-control where a numerical value in a defined range is necessary. Constraining user input in this manner is an essential part of minimising configuration errors.

6.3.3 Executing RÖSTI as an External Process

RÖSTI depends explicitly on the runtime and execution environment provided by OMNet++. There is currently no support for executing the OMNet++ runtime as a child process of another application, so hosting RÖSTI under the Configurator is not possible. It is thus necessary to run RÖSTI as a slave process of the Configurator, using the process management capabilities of the .NET Framework.

Configuration of RÖSTI from the Configurator is accomplished by writing the appropriate text files to disk, which are subsequently parsed by RÖSTI. A more important consideration is providing the user with a status indication of the progress of RÖSTI in executing the simulation.

University of Cape Town

This requires feedback from RÖSTI to be relayed back to the Configurator, which is achieved by redirecting and parsing the console output of RÖSTI. The Configurator then displays this progress status to the user.

The Configurator also provides the capability to run RÖSTI in Debug mode. This mode displays a GUI representation of the execution of a simulation, and allows the user to pause a simulation, edit simulation properties and alter the simulation execution speed. This is again accomplished by running the RÖSTI debug executable as a slave process.

6.3.4 Removing DiskSim Support

An unfortunate consequence of focusing development of the Configurator on the Windows Operating System platform was the exclusion of DiskSim support from the Configurator. As outlined in Section 5.7.3 (p. 81), the DiskSim support within RÖSTI is limited to execution in Unix-based environments such as GNU/Linux. It was thus not possible to support DiskSim from the Configurator.

6.3.5 Partitioning Results

The Analysis tool described in Chapter 7 depends on the partitioning of simulation results. This partitioning refers to separating the results from independent simulation runs, whilst still grouping related output results which are used to calculate output statistics over an entire simulation study. A simulation study consists of a series of runs based on a single base configuration.

For our purposes, a run refers to a single simulation configuration modified by either: changing the value of an input parameter that is represented by a variable[5]; or changing the seed for the random number generator. The former case allows a single simulation execution to span multiple simulation runs over a range of parameter values. The latter case allows a single configuration to be simulated multiple times for the calculation of confidence intervals. Both cases, and combinations thereof, are supported and correctly marked as such, thus allowing the Analysis tool to correctly parse this information.

We have chosen to separate the results into individual folders, which are assigned monotonically increasing numbers for each simulation study that is completed. Each folder contains the input configuration text-files generated based on values in the C# classes. The output scalar and vector files from the simulation study are also stored in this folder, thus creating a clear association between simulation configuration and results. As with all OMNet++-based simulations, each scalar and vector output file contains the results from a number of simulation runs.

[5]See Section 6.1.3 (p. 86)

University of Cape Town

Chapter 7

Analysing Simulation Results

University of Cape Town

7.1 Motivation

Analysis of simulation results forms part of the usage workflow for RÖSTI, as illustrated in Figure 19. An analysis tool that parses the output results of a simulation, and is able to graph relations between these results, is valuable in a number of scenarios:

- Determining whether simulation results are close to expectations, as a first order measure of the validity of a simulation model.
- Performing rapid fine-tuning of simulation parameters by integrating simple analysis with configuration and execution steps.
- Assisting novice users of RÖSTI when first running simulations. A simplified analysis tool would aid in understanding the initial simulation results and the effects of changing various parameters.

Given these considerations, we have chosen to develop a minimal data manipulation tool capable of graphing simulation input parameters against output results for initial analysis. This analysis tool is further integrated into the existing Configurator GUI interface, thus presenting a single interface to the user. This aids interface coherency, providing a single consistent window from which RÖSTI simulations can be configured, executed and analysed.

7.2 Utilising the Partitioned Results

In order to aid the processing of simulation output data, we have leveraged the partitioning of simulation output data performed by the Configurator. We have associated the notion of a result set with each partitioned data set, indicating that all the data from a number of runs[1] within a single simulation study[2] should be treated as related.

[1] A run is an OMNeT++ term indicating an execution of the simulation model with a particular configuration.

[2] A group of runs related by a common configuration to which slight modifications are made. See Section 6.3.5 (p. 90) for a more detailed description of how runs are grouped into a simulation study.

Using this concept of result sets, we are able to display the list of available result sets (and hence simulation studies) that are available. Selecting a set from this list allows various combinations of statistics of interest to be comparatively graphed and displayed within the GUI environment, as outlined below. The use of result sets importantly ensures that results from unrelated simulation studies, with possibly wildly different configurations, are not accidently compared.

7.3 Implementation

The Analysis tool has been implemented in C#, as with the Configurator. The interface itself is provided as a secondary tab within the Configurator interface, as illustrated in Figure 36. The interface is automatically activated at the end of a simulation study execution. This further accentuates the tool's role in the typical usage workflow of RÖSTI. The tool uses GnuPlot for producing graphs, and allows for plotting both scalar and vector results to various output formats.

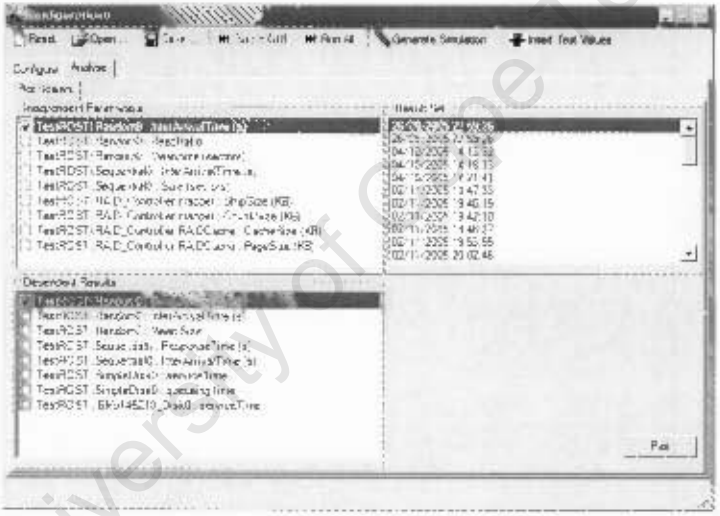


Figure 36: Screenshot of the interface to the analysis tool.

University of Cape Town

7.3.1 Using GnuPlot for Graphing

GnuPlot is an open-source, freely available[3], fully-featured graphing utility that is widely used within the scientific research community. It is capable of producing attractive graph outputs in numerous formats and supports both 2D and 3D graphs. GnuPlot is operated by means of text commands, and as such is not immediately applicable to our needs, which seek to reduce analysis complexity for the end user.

GnuPlot is capable of accepting batches of commands via text files, and it is this mode of interaction that we use to generate our graphs. By constructing a command file describing the desired graph and passing this file to GnuPlot as a command line parameter, we are able to seamlessly generate an output graph that can be displayed to the user. This processing by GnuPlot is accomplished by running it as a slave process under the Analysis GUI. When GnuPlot completes processing the batch file, the slave process exits and the GUI is able to continue operation. A sample output result from this process is illustrated in Figure 37.

[3]Downloadable at `http://www.gnuplot.info`.

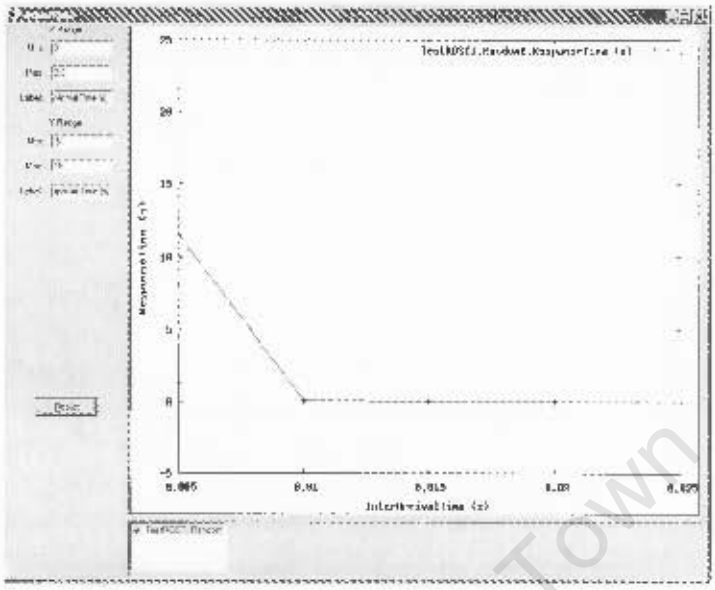


Figure 37: Screenshot of a sample output from the analysis tool.

The use of a batch command file allows us to specify the appearance of axis labels and ranges, as well as titles and colours for the graph legend. In addition, we provide the user with a set of controls to customise these features - when a control is changed, the batch command file is recreated and reprocessed by GnuPlot, and the resulting image displayed to the user. This provides the user with the illusion of interacting with an embedded graphing component, rather than an independent slave process.

The Analysis tool also allows saving the batch command file to disk. Since this file simply contains a series of GnuPlot commands, it can be run through GnuPlot on a separate machine (without RÖSTI installed) to produce an identical output to the source graph. More importantly, it allows for more precise customisation of graphs by inserting more advanced GnuPlot commands within the command file.

The advanced graphing capabilities of GnuPlot also allow for the possibility that the Analysis tool can be extended to handle 3D plots where 2 independent simulation input parameters are plotted against a single dependent output.

7.3.2 Plotting Output Scalars against Input Parameters

Scalar is OMNet++ terminology for single statistics measured over the course of a single simulation run. Since both input parameters and output scalars are written to the same file, it is necessary to differentiate between output scalars and input parameters by the use of a "parameter" prefix on the statistic name. Parsing the scalar output using this mechanism, a set of list boxes for independent inputs and dependent outputs is populated, allowing the user to select combinations of interest to be graphed.

For simplicity, we have restricted the user's choice to a single input parameter - cases with 2 or more independent variables are covered by the advanced graphing capabilities mentioned above. We allow any number of output scalars to be selected as dependent variables, however this may

University of Cape Town

cause problems where individual outputs cover widely different ranges. User discretion is required in this case, as it may require axis ranges to be specified, or for separate graphs to be created for each output scalar.

Once the user has selected the data to plotted, the batch command file is created with a legend labeled with static names for each dependent statistic obtained from the scalar output file. The axes are appropriately labeled using units drawn from the scalar output file, and the completed command file is passed to GnuPlot for processing. The resultant image is then displayed to the user.

7.3.3 Plotting Output Vectors

A vector is OMNet++ terminology for a series of individual data values captured throughout the execution of a simulation run. Each data value is associated with the simulated time at which it was measured, and output vectors thus have time as an implicit independent variable. Vectors are thus plotted identically to Scalars, with the vector data points as dependent values and the associated simulation time as independent values. The same graph customisation options are availabe for Vector plots, and multiple vectors can be plotted on the same graph.

University of Cape Town

Part III

Testing

University of Cape Town

Chapter 8

Validating the Simulator

University of Cape Town

RÖSTI consists of four main components that must each be validated: the Disk Models (Section 8.1); the Data Sources (Section 8.2); the RAID Controller (Section 8.3); and the RAID Cache (Section 8.4). Each section presents the issues relevant to validation, the approach adopted in validating the component, and any specific details relevant to validation.

8.1 Validating the Disk Models

For this release of RÖSTI we have chosen to focus on the operation and performance of the RAID Controller, and its effect on the system as a whole. The simulation of the disk drives in the system is not our main focus, and as such we have used other, specialised simulation modules that focus on this area. In particular, we have used disk simulations from DiskSim and the IBM Zürich research group. In each case, we have assumed that the respective disk simulation has been validated by its creators, as they have more experience with the intricacies of disk simulation than we do.

As with the rest of the simulator, the disk modules are designed to be replaceable, so any of the default disk models can be replaced with more accurate implementations as required.

8.2 Validating the Data Sources

RÖSTI currently supports two types of data sources: trace-driven, which use logs of I/O requests to replicate workloads from actual systems; and synthetic, which use mathematical models to generate workloads that are similar to real-world workloads. Given their origin, trace-driven data sources do not need validation as they represent a specific workload instance.

The synthetic data sources included with RÖSTI generate workloads that test various boundary condition behaviours of a RAID Controller. They are represented by simple mathematical relations, and frequently use random number generation to determine I/O field values. There are thus two areas that must be validated:

- The above workloads exhibit particular features, used in testing boundary conditions of

a RAID Controller, that are represented by a set of mathematical relations. Given these relations, it is necessary to validate that a given generated workload exhibits the expected features and is statistically consistent with the mathematical model. A tool such as ESSWA (Enterprise Storage System Workload Analyser [Sik06]) can be used to check this.

- The Random Number Generator (RNG) used by these data sources should be validated to ensure that the random number sequences produced by a given seed are always identical, and that these sequences are statistically equivalent to their respective distributions. The RNG used in RÖSTI is an implementation of the Mersenne Twister algorithm [MN98a] that ships as part of OMNet++ and has been extensively used in a number of simulation studies based on the OMNet++ framework. As such, we have assumed that this RNG is statistically correctly.

It is envisioned that future uses of RÖSTI will utilise synthetic workloads generated by ESSWA, a tool developed for analysing real-world trace files and generating representative synthetic workloads that exhibit the same characteristics. This will provide RÖSTI with a robust input-data generator for use in comprehensive studies of RAID Controller behaviours.

8.3 Validating the RAID Controller

The RAID Controller is the core of RÖSTI, but is also a simplified, synthesised model of the behaviour of complex real-world implementations. Validation therefore centres on the functional behaviour of the controller, and the extent to which this behaviour matches the behaviour of real-world systems. We specify a comprehensive set of test cases to test this, for each supported RAID level, that cover the full range of expected inputs. Each test case is then independently validated, and if all test cases are passed then the controller is deemed to be valid for that RAID level.

University of Cape Town

8.3.1 Approach

The work of Courtright [Cou97] presents a novel approach to verifying the correctness of RAID Controller operations. As part of this approach, he defines a modelling notation using Directed Acyclic Graphs (DAGs) that formalises the idea of a RAID Controller operation. Using this notation, he presents several orthogonal operation types that cover the spectrum of possible operations for a RAID Controller. Each operation type is constructed such that it minimises the number of disk requests issued. This condition is important as disk requests are the dominant time factor in fulfilling an I/O request.

Using these operation types, Courtright was able to construct a RAID simulator, RAIDframe [CGHZ96a], that performed automatic recovery and rollback from errors during operations, using the DAGs for each operation type. Since the correctness of the component DAGs are verifiable using formal methods, it follows that the above RAID simulator is also verifiable.

The operation types outlined by Courtright are categorised according to several attributes:

- The failure mode the array is in, e.g. Normal, 1-Disk failure, 2-Disk failure, etc.

- The type of the I/O request associated with Controller operation, i.e. Read or Write.
- The size of the I/O request relative to the number of disks in the array.

Each operation type can be illustrated by a diagram such as Figure 38 (p. 99). In this diagram, the cylinders represent strips in an array stripe, while the letters indicate the contents of that strip and the subscript serves as an index. Hence D_1 indicates a data strip with index 1, while P_{0123} indicates a parity strip protecting data strips 0 through 3. The arrows indicate the flow of data between the controller, the strips (and hence disks) and the XOR engine, which is responsible for calculating the parity over a set of data strips.

Each arrow terminating on a strip indicates a write request issued to the disk, while arrows originating from a strip indicate read requests to the associated disk. This convention then naturally leads to a validation strategy for RÖSTI that is agnostic of actual implementation details. Consider the RÖSTI RAID Controller to be a black box with I/O requests from data sources as inputs and I/O requests to attached disks as output. Each possible input can be classified as one of the allowed operation types, and the expected outputs determined from this. If the actual I/O requests issued to disk match the expected requests for that input, then the test is successful. Successfully repeating this procedure for a large number of inputs that all map to a single operation type, we can validate the operation of the controller for that operation type. This can then be applied to validate all operation types for a given RAID scheme.

University of Cape Town

8.3.2 Method

Although we have not adopted the DAG notion for RÖSTI, we have used the operation types outlined by Courtright to direct the validation of RÖSTI. Specifically, we generate representative workloads for each of the operation types, and then run a number of simulations using these workloads. For each run, we log all read and write requests submitted to each disk. By examining this log and comparing it with the expected set of disk requests for that operation type, we can determine if the RAID Controller correctly processed the associated request.We deem a request to be correctly processed if the following fields are identical between expected and actual disk requests:

- Request type (read/write)
- Request size (in bytes)
- Request LBA

In the following sections we present the various operation types for each supported RAID configuration. Each operation type description is accompanied by a diagram, as described above, which illustrates the expected outputs checked during validation.

8.3.3 RAID 5 - Normal Mode Operation Types

The following operation types all relate to operations submitted to a RAID-5 array running in Normal mode (no disk failures). For the purpose of illustration, operation type diagrams are

drawn for an array with 5 disks (4 data, 1 parity), but the rules defining the operation type can be applied to arrays of any size.

Read

Read requests are the simplest operation type, as illustrated in Figure 38. All disks are available in normal mode and no parity calculations are needed, thus the only consideration is the striping of data across the disks.

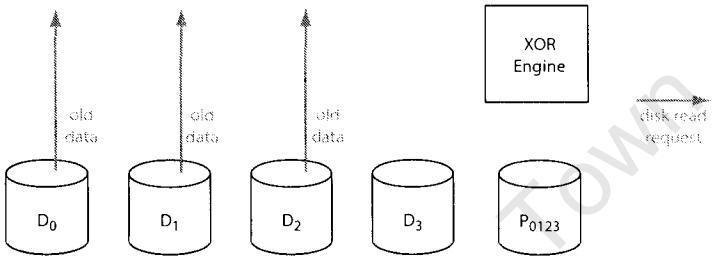


Figure 38: Disk operations for a RAID 5 Read request in Normal Mode

Small Write

Small writes refer I/O requests whose size is less than half the stripe capacity of an array. For example, the array in Figure 39 has 4 data strips in a stripe. Assuming a strip size of 16KB, the stripe capacity is 64KB and hence a small write would be any request with a size less than 32KB. In this situation, we are modifying less than half the disks in the array, so having to read the old data from all disks to recompute parity is less than ideal. We can avoid this by using the fact that:

$$P_{old} = D0_{old} \otimes D_1 \otimes D_2 \otimes D_3 \quad (11)$$

$$\Rightarrow P_{old} \otimes D0_{old} = D0_{old} \otimes D0_{old} \otimes D_1 \otimes D_2 \otimes D_3$$

$$= D_1 \otimes D_2 \otimes D_3 \quad (12)$$

where P_{old} is the old parity value and $D0_{old}$ the old value of D_0 before the write. This then provides the following formula to calculate the new parity value to be written to disk:

$$P_{new} = D0_{new} \otimes D_1 \otimes D_2 \otimes D_3$$

$$= D0_{new} \otimes P_{old} \otimes D0_{old} \quad (13)$$

Equation 13 then minimises the number of disk accesses required, as illustrated in Figure 39. Hence, we now need only read the data from strips that are being modified to recalculate parity.

Large Write

Large writes are the complementary case to small writes, and refer to I/O requests whose size is greater than half the stripe capacity but less than the full stripe capacity of an array. Referring to the example above, a large write would be any request between 32KB and 64KB. In this case,

University of Cape Town

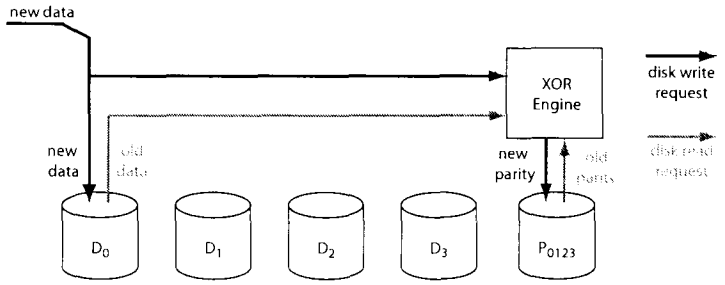


Figure 39: Disk operations for a RAID 5 Small Write request in Normal Mode

we have as input more than half the required data to calculate parity. Whereas with the small write it made sense to read only those data strips that were being modified, in this case we can read only those data strips that are unaffected by the write. Combined with the data that is to be written to the new disks, we can recalculate parity with far fewer disk accesses. This is illustrated in Figure 40 and follows from the equation:

$$P_{new} = D0_{new} \otimes D1_{new} \otimes D2_{new} \otimes D3_{old} \tag{14}$$

which effectively states that the new parity value is given by an exclusive-or of the newest values for each data strip in that stripe.

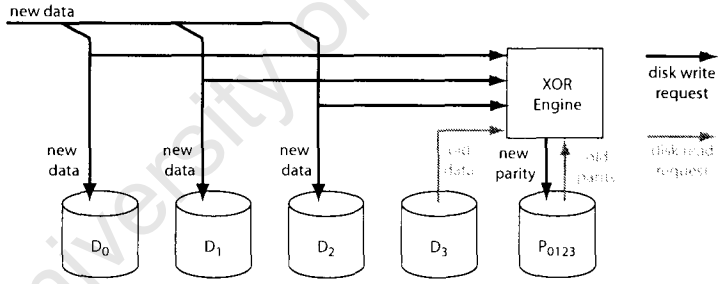


Figure 40: Disk operations for a RAID 5 Large Write request in Normal Mode

Full-Stripe Write

Full Stripe writes refer to I/O requests whose sizes are exactly equal to the stripe capacity of the array. In other words, every data strip in a single stripe is modified by the request. In this case it is unnecessary to read in any of the old data strips, as parity can be directly calculated from the new data, as illustrated in Figure 41.

Cross-Stripe Write

The previous three write operation types may be considered atomic, in the sense that there is no overlap between them. There are, however, various conditions in which an I/O request will stretch across stripe boundaries as illustrated in Figure 42. The Cross-Stripe Write is a composite operation type than encompasses the idea that a single I/O request may map to many operation types on adjacent stripes.

University of Cape Town

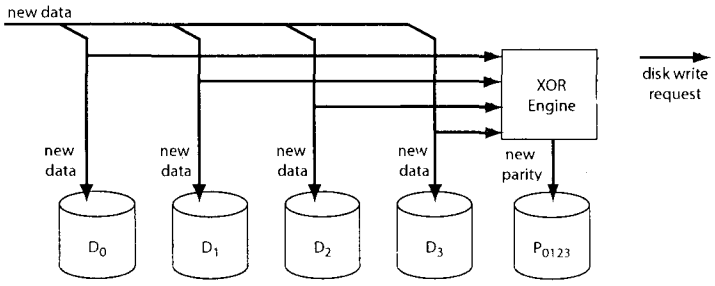


Figure 41: Disk operations for a RAID 5 Full-Stripe Write request in Normal Mode

Since a Cross-Stripe write is composed of several atomic operation types, it follows that validating each of the atomic operation types is equivalent to validating the composite operation type. A potential problem with this approach is that the RAID Controller combines several small requests to a given disk into one, large, continuous request. It is thus difficult to separate each request into its original component requests. The solution to this involves combining the expected disk requests for each operation type to create an expected composite disk request for each disk. This composite request is then compared to the actual disk request issued by the RAID Controller - if they match, then the Cross-Stripe write is deemed valid.

0	D	D	D	D	P
1	D	D	D	P	D
2	D	D	P	D	D
3	D	P	D	D	D
4	P	D	D	D	D

Figure 42: Cross-Stripe Write example. For this write request, accessing stripe 1 is a Large write, stripe 2 is a Full-Stripe write and stripe 3 is a Small Write.

University of Cape Town

8.3.4 RAID 5 - Degraded Mode Operation Types

Degraded mode operation occurs when exactly one disk in the array has failed. Depending on the location of the stripe containing the requested data in the array, the strip on the failed disk may contain either data or parity. Each case (missing data or missing parity) must be handled differently, and the number of required disk operations increases as more data is required to reconstruct that which is missing.

Read with Parity Strip Missing

In this case, a read of data strips is requested from a stripe where the parity strip is missing. Since all requested data is available, no data reconstruction is required. The operation can thus continue exactly as it would for the Normal Mode Read case, as illustrated in Figure 43.

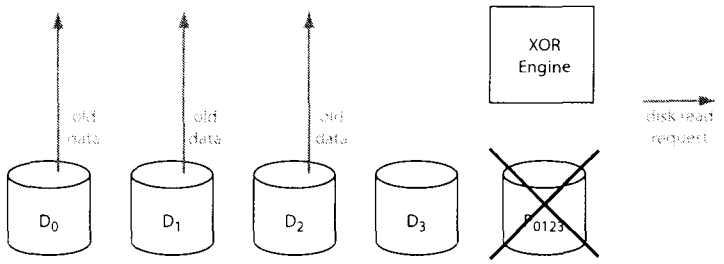


Figure 43: Disk operations for a RAID 5 Read request in Degraded Mode with a Parity strip missing.

Read with Unaffected Data Strip Missing

In this case, a read of data strips is requested from a stripe where a data strip is missing. The missing data strip has not been requested, however, and thus does not impact the operation. This case is almost identical to the previous one, as all required data is present and no data reconstruction is required.

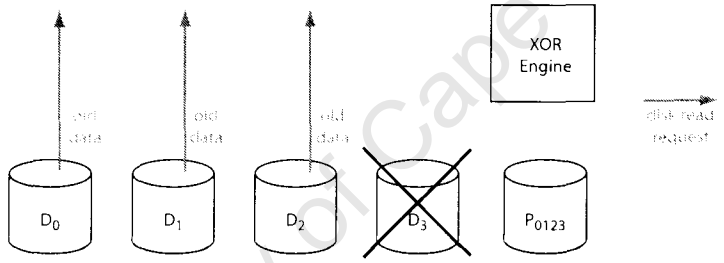


Figure 44: Disk operations for a RAID 5 Read request in Degraded Mode with an unaffected data strip missing.

Read with Required Data Strip Missing

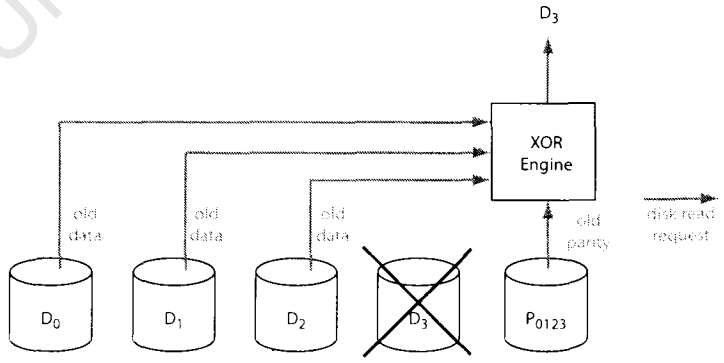


Figure 45: Disk operations for a RAID 5 Read request in Degraded Mode with a Parity strip missing.

In this case, a read of data strips is requested from a stripe where a required data strip is missing. Since the missing data is required, it is necessary to reconstruct it from the available data and parity. This involves reading all remaining data strips and the associated parity strip, as illustrated in Figure 45, and recalculating the missing data via the XOR engine using the relation:

$$D_3 = P \otimes D_0 \otimes D_1 \otimes D_2 \tag{15}$$

University of Cape Town

Write with Parity Missing

In this case, a write to a set of data strips is requested, while the associated parity strip is missing. Under normal mode, the parity mode would need to be updated as part of the write operation, but here it can safely be ignored since it cannot be used later on to reconstruct data in the stripe. The expected operations thus consist of a set of disk write requests for each of the modified data strips, as illustrated in Figure 46.

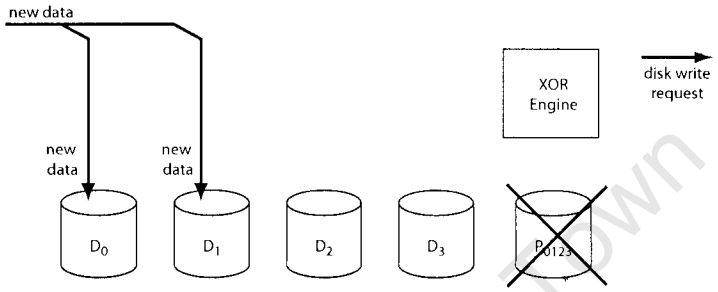


Figure 46: Disk operations for a RAID 5 Write request in Degraded Mode with a missing parity strip.

Write with Unaffected Data Strip Missing

In this case, a write to a set of data strips is requested, where an unaffected data strip is missing. Although the missing data strip is not being modified, it can still be reconstructed from the parity information available, and hence it is imperative that the parity be correctly updated with the new data being written. This update proceeds exactly like the Small Write case in Normal mode (Figure 39), and requires that all the old values of modified strips be read from disk and passed to the XOR engine together with the new data strips and the old parity value to calculate the new parity value. This new parity is then written back to disk along with the new data strips, as illustrated in Figure 47.

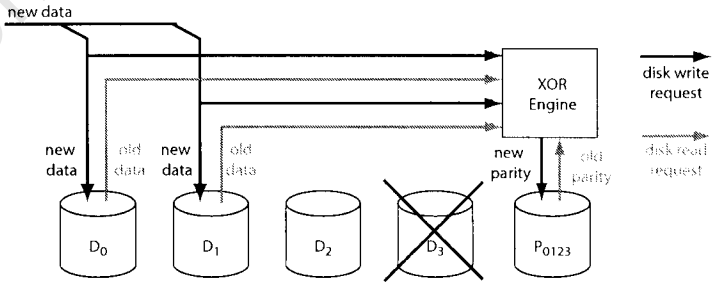


Figure 47: Disk operations for a RAID 5 Write request in Degraded Mode with a missing unaffected data strip.

Write with Required Data Strip Missing

In this case, a write to a set of data strips is requested, where one of the strips in the set is unavailable. The missing data strip can be reconstructed from available parity information, hence the write can be accomplished by updating the parity with *all* new data available. This

University of Cape Town

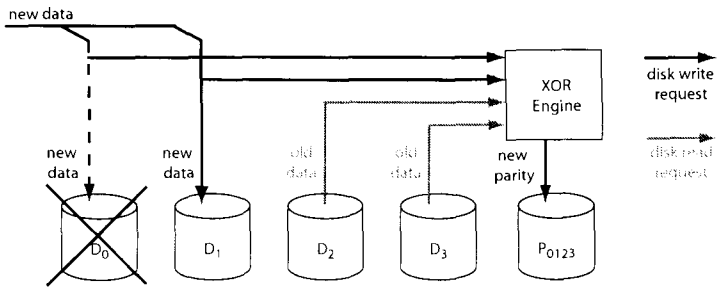


Figure 48: Disk operations for a RAID 5 Write request in Degraded Mode to a missing data strip.

updated parity is then written to disk along with the other modified data strips, as illustrated in Figure 48.

8.3.5 RAID 6 Preamble

As outlined in Section 2.3.5 (p. 12), RAID 6 is a dual-failure tolerant scheme that uses both XOR parity (like RAID 5) and Reed-Solomon (RS) parity. Reed-Solomon parity is independent from XOR parity and hence the combination of the two can be used to reconstruct any two data strip failures within a single stripe. Further, one of the mathematical properties of Reed-Solomon coding is that changes in RS parity can be calculated using only the changes in dependent data strips, rather than requiring it be recalculated using all data strips in the stripe. This follows from the matrix equation:

$$c'_i = c_i + \sum_{j=1}^{n} f_{ij}(d'_j - d_j) \tag{16}$$

where f_{ij} are the elements of an $n \times m$ Vandermonde matrix: $f_{ij} = j^{i-1}$, m is the number of data disks in the array, and n is the number of parity disks. c'_i and c_i are, respectively, the new and old value of the RS parity, and d'_j and d_j respectively are the old and new values of the changed data strips in the array. The above arithmetic is performed over a Galois Field. This equation is effectively equivalent to the parity recalculation equation for RAID 5 arrays with XOR parity (Equation 13).

The important point is that since parity updates can be done using only changed data, we can simply modify the RAID 5 operation types for use in RAID 6, whilst still minimising the number of issued disk requests per I/O request. Further, these operation types can be composed using Composite Stripe Writes in exactly the same manner as the RAID 5 operation types.

8.3.6 RAID 6 - Normal Mode Operation Types

The following operation types all apply to a RAID 6 array operating in Normal Mode, with all disks available. We have presented the operation types using a canonical array representation with 4 data disks, 1 XOR parity (P) disk and 1 RS parity (Q) disk, thus these operation types can be equally well applied to larger arrays, or arrays with parity rotation enabled.

University of Cape Town

Read

In this case, a read of a set of data strips is required from a single stripe. Since no parity updates are required, it is only necessary to read the relevant data strips, as illustrated in Figure 49. The individual data from each strip is then combined by the RAID Controller to reverse the effect of data striping across the disks.

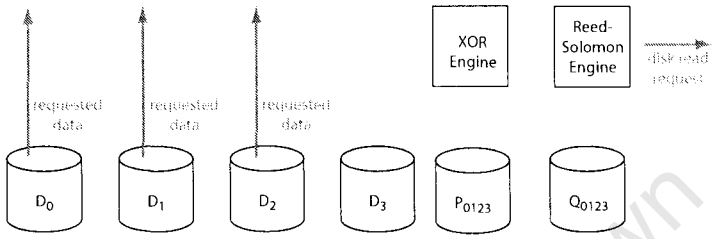


Figure 49: Disk operations for a RAID 6 Read request in Normal mode.

Small Write

This case applies when a request is received to write less than half the array stripe capacity, and is almost identical to the Small Write for RAID 5, with the addition that the old and new data values are also used to update the Reed-Solomon parity strip. As illustrated in Figure 50, both P and Q parities are updated using the changes in data strips, rather than the entire stripe contents, using Equations 13 and 16.

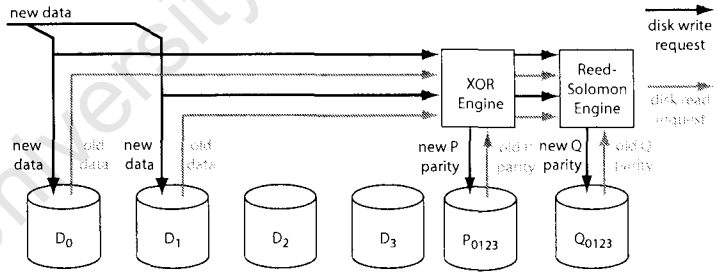


Figure 50: Disk operations for a RAID 6 Small Write request in Normal mode.

Large Write

This case applies when a request is received to write more than half the array stripe capacity, like the RAID 5 Large Write case. Both P and Q parities are recalculated using the newest values for each strip in the stripe, as illustrated in Figure 51.

Full-Stripe Write

This case applies when a request is received to write the entire array stripe capacity. Both P and Q parities can be recalculated from the new data without requiring any reads from disk, as illustrated in Figure 52.

University of Cape Town

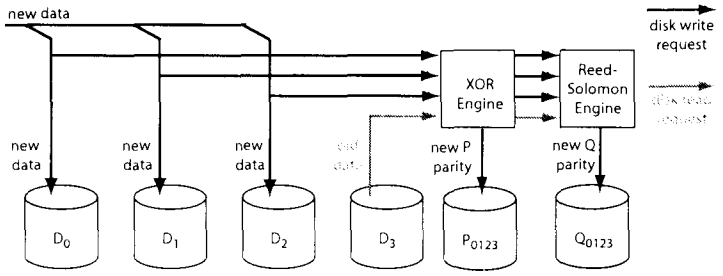


Figure 51: Disk operations for a RAID 6 Large Write request in Normal mode.

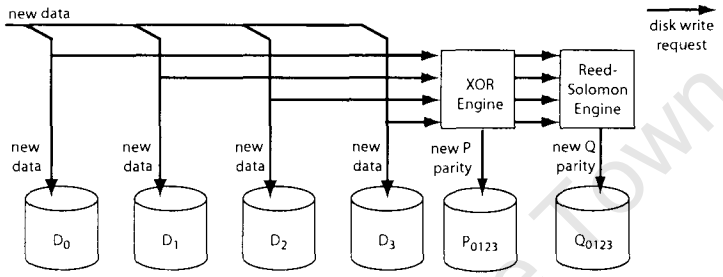


Figure 52: Disk operations for a RAID 6 Full-Stripe Write request in Normal mode.

8.3.7 RAID 6 - Degraded and Critical Mode Operation Types

A RAID 6 can be in either of two failure states: degraded mode where a single disk has failed, in which case the array behaves almost exactly like a RAID 5 array in degraded mode; or critical mode where two disks have failed, putting the array in a critical state from which no further failures can be tolerated. As is the case with RAID 5 arrays, the strip on the failed disk may contain either data or parity depending on the location of the stripe containing the requested data in the array. Due to the similarity of RAID 6 degraded mode to RAID 5 degraded mode, the operation types presented focus on critical mode with reference made to degraded mode where the expected behaviour is not clear.

University of Cape Town

8.3.8 RAID 6 - Failure Mode Read Operation Types

For RAID 6 operations in a failed mode, a distinction is made between read and write operation types. This section covers the read operation types encountered when in either degraded or critical failure mode.

Read with No Reconstruction

This case applies when a request is received to read a set of strips that are all available in the related stripe. Since no reconstruction is required, the strips can simply be read and returned. This operation type is applicable to both degraded and critical failure modes, and is illustrated in Figure 53.

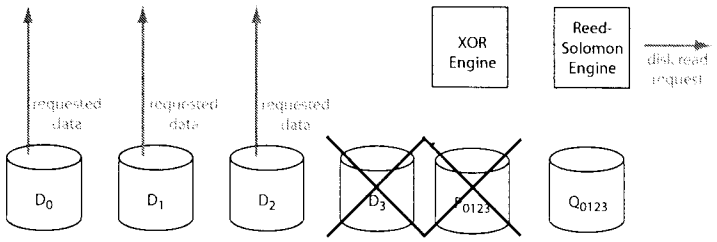


Figure 53: Disk operations for a RAID 6 Read request in Degraded mode where no data reconstruction is necessary.

Read with Single Reconstruction

This case applies when a request is received to read a set of strips where exactly one requested data strip is unavailable in the related stripe. This missing data strip must be reconstructed by reading the remaining data and parity strips, and using the appropriate recovery technique depending on which parity is available, as either of the parity strips may be unavailable. It may also be the case that both parity strips are available (and hence only a single data strip is missing), in which case either parity may be used in reconstruction. This case is applicable to both degraded and critical failure modes, and is illustrated in Figure 54.

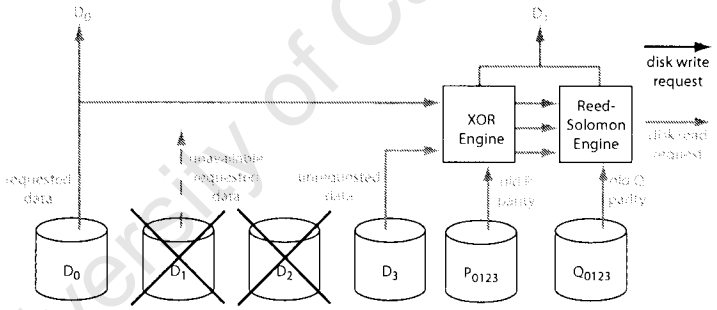


Figure 54: Disk operations for a RAID 6 Read request in Degraded mode where a single data reconstruction is necessary.

Read with Double Reconstruction

This case applies when a request is received to read a set of strips where exactly two requested data strips are unavailable in the related stripe. This implies that the array is in critical mode and that both parity strips are available. The missing data must be reconstructed by reading the remaining data strips and both parity strips in the stripe. The relationships between the surviving data strips and each of the parity strips are linearly independent, hence the missing strips can be reconstructed by solving a simple set of simultaneous equations or using Gaussian Elimination on an appropriately constructed matrix, as presented by Plank [Pla97]. The required disk requests to enable this are illustrated in Figure 55.

University of Cape Town

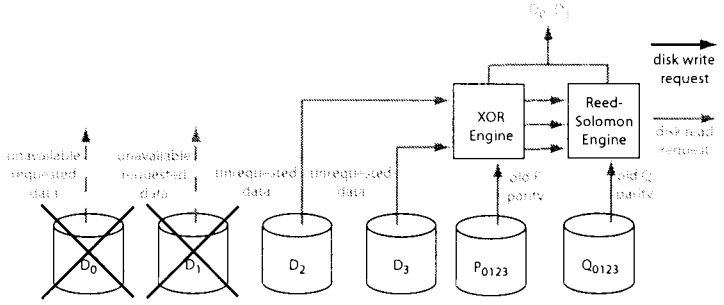


Figure 55: Disk operations for a RAID 6 read request in Degraded mode where a double reconstruction is necessary.

8.3.9 RAID 6 - Failure Mode Write Operation Types

This section covers all write operation types in both degraded and failed modes. There are three types of strip failures that can be experienced: data failures, where a modified data strip is unavailable; parity failures where the P or Q parity for a given stripe is unavailable; or unaffected failures, where an unmodified data strip is unavailable. In degraded mode, the operation types are very similar to RAID 5, however there is significant overlap in how the various combinations of these failure are handled in critical mode.

Write Operation Type	Missing Strips
Write With Only Missing Parity Strips	1 parity 2 parity
Write With Missing Unaffected Strips	1 unaffected 2 unaffected 1 unaffected, 1 parity
Write With Missing Data Strips	1 data 1 data, 1 parity 1 data, 1 unaffected 2 data

Table 5: The relation between failed strip types and the degraded write operation types.

University of Cape Town

In particular, each strip failure type adds an additional level of complexity to how a write operation is handled. We have therefore separated a failure mode write into three operation types, with each type adding recovery support for one more strip failure types. Table 5 (p. 108) lists these three operation types that we have derived together with the associated failure conditions. Each operation type covers the expected requests for each associated failure condition.

Write with only Missing Parity Strips

This operation type deals only with write requests where one (degraded) or both (critical) parity strips are unavailable. If one parity strip is available, it can be processed identically to a RAID 5 Normal Mode write[1]. If both parity strips are unavailable, the write can be successfully completed

[1]A RAID 6 stripe with one failed parity strip is functionally equivalent to a RAID 5 stripe

by ignoring the failed parity strip(s) and writing the appropriate data strips, as illustrated in Figure 56.

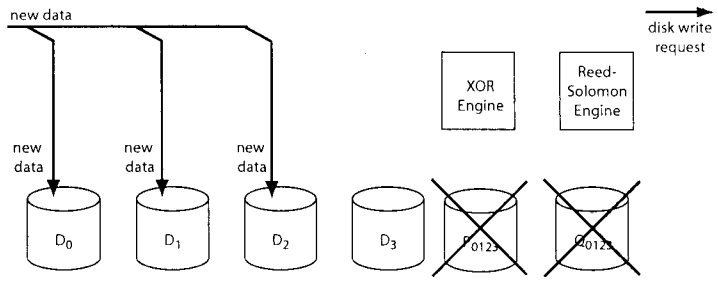


Figure 56: Disk operations for a RAID 6 Write request in Critical mode with only parity strips missing.

Write with Missing Unaffected Strips

This operation type deals with write requests where unaffected and parity strips may be unavailable. The common factor here is that at least 1 unaffected strip is missing. This presents problems, since parity strips can not be recalculated using all available strip data (see RAID 5 Large Write). This implies that any parity strips must be updated using the same procedure as for a RAID 6 Small Write, even though this doubles the number of disk requests required. Thus the old value of all modified strips must be read and combined with the new data from the write request and the current parity values to update the parity, which is then written back to disk along with the new data.

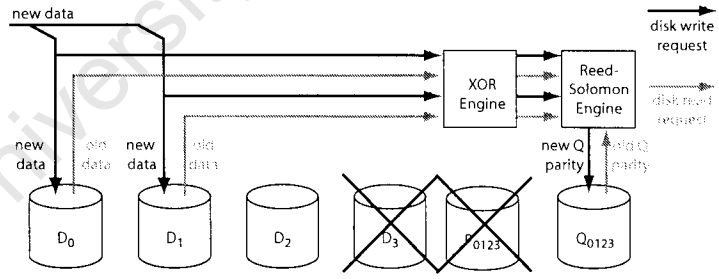


Figure 57: Disk operations for a RAID 6 Write request in Critical mode where one unaffected and one parity strip are missing.

Figure 57 illustrates this parity update for the case where one unaffected and one parity strip are missing. The remaining Q parity strip is updated using the method above. Figure 58 illustrates the expected requests for the case with two missing unaffected strips. Here both P and Q parities are updated using the Small Write algorithm. The case where one unaffected strip is missing is handled identically.

Write with Missing Data Strips

This operation type allows for any combination of strip failure types, with the restriction that at least one data strip has failed. In this case, it is not possible to read the old value corresponding

University of Cape Town

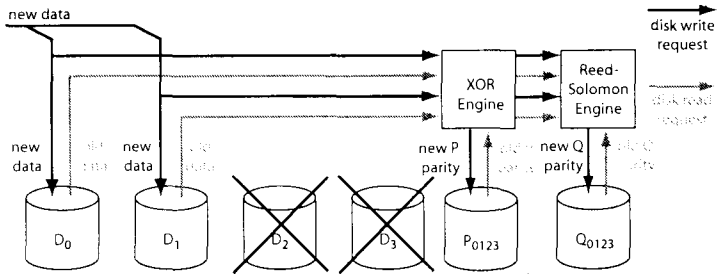


Figure 58: Disk operations for a RAID 6 Write request in Degraded mode where two unaffected strips are missing.

to at least one new data strip, so updating parity using the Small Write method is not possible. This implies that the new parity values must be reconstructed using all data strips in the array.

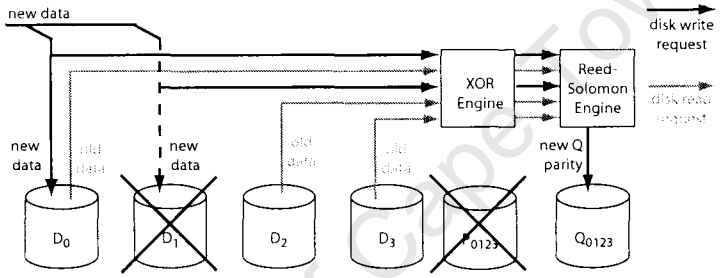


Figure 59: Disk operations for a RAID 6 Write request in Critical mode where one data and one parity strip are missing.

If no unaffected strips are missing, this is simple and amounts to reading all unaffected strips and performing the parity calculations on these together with the new data. The new data and parity strips are then written back to disk. This is illustrated in Figure 59 (failed data and failed parity) and Figure 60 (two failed data).

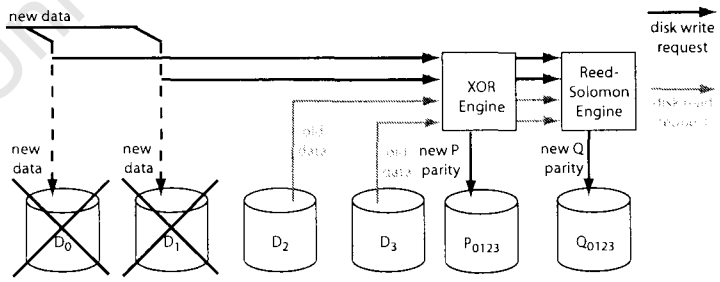


Figure 60: Disk operations for a RAID 6 Write request in Degraded mode where two data strips are missing.

A problem arises if both a data and an unaffected strip are missing. In this case, as previously, all old data cannot be read, hence parity cannot be recalculated. This requires that the missing unaffected data be reconstructed using one of the remaining parity strips, as is the case for a RAID 6 Degraded Read. Once the missing unaffected data is reconstructed, the parity strips can be recalculated using all the data strips in the stripe, since they are now all available. The new data and parity strips are then written back to disk.

University of Cape Town

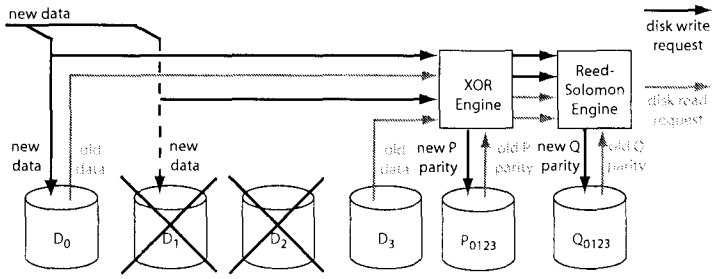


Figure 61: Disk operations for a RAID 6 Write request in Degraded mode where a single reconstruction is necessary.

8.4 Validating Cache Operation

Caching in RÖSTI models the effect of a cache on the overall performance of the system, rather than the actual physical implementation of such a cache. Thus the primary focus of validation of RÖSTI caching schemes is on the functional behaviour of such a cache, rather than the validity of the design. In other words, although a RÖSTI cache implementation may differ significantly from real-world equivalent implementations, the effect of the cache on the operation of the system will be the same. It is the correctness of this effect that we wish to validate.

University of Cape Town

8.4.1 Approach

Our validation of the cache implementation focuses on analysing functional behaviour of the cache, rather than low-level implementation details. Changes in the state of the cache occur only when an operation is executed, and hence a particular operation can be characterised in terms of the expected changes in the cache state. As such, we can concentrate on examining the state of the cache over the course of simulation and verifying that a given operation produces the expected change in the cache state.

There are two approaches to ensuring this correlation between cache operations and cache state changes. The first is to implement some form of automated testing system which examines each requested operation, determines the expected cache state change, and verifies that this change occurs. The second is to visualise the state of the cache in a meaningful manner and using this visualisation to manually check state changes.

The automated testing system option works well in cases where there are large numbers of implementations to test, or if a particular implementation has numerous test cases. Such a system would require some form of formal description technique to describe the expected state change for a given operation. Validating a single operation would require different steps for each particular cache policy being tested, due to the interdependency between the cache contents and the associated status information. Finally, depending on the complexity of this testing system, it may be necessary to validate the correctness of the testing tool itself. This complexity makes an automated testing solution a poor fit for our testing requirements.

The latter option is well suited to small numbers of test cases, and is the approach we have adopted. Visualising the cache state requires the cache implementation to be slightly modified so as to output the cache state after each operation into a trace file. This output is then used as

input to a separate visualisation tool that was created, which shows the cache state before and after each operation. This scheme is also extensible to incorporate automation, by feeding the output cache state data to an automated validation tool. This output data can then be used to manually determine the correctness of the cache operation for each of the boundary condition test cases.

Generating these test cases is accomplished by specifying workloads which exercise particular boundary conditions. For each such workload, a list of expected state changes is described and then manually compared to the visualised cache state. If all expected state changes are encountered, without side-effects, the associated test case is deemed to have been validated.

The normal operation of the cache is first validated by conducting a number of simulation runs, and testing the constraints outlined in Section 8.4.2 (p. 112). Once this is accomplished, boundary conditions for each cache policy can be validated. The boundary conditions vary dependent on the cache policy in use, and are described in Section 8.4.3 (p. 114). Each boundary condition tests a specific aspect of the cache operation where errors are most likely, since they are rarely encountered.

The validation process is initially conducted on small cache sizes, as these are significantly more manageable in terms of testing. Once the cache has been validated for these sizes, a small number of larger cache sizes are validated to ensure correct operation. This approach is valid since the caching policy and operation are independent of cache size, and as such an inductive approach to validation works well.

Each cache implementation in RÖSTI was subjected to the following validation procedures in an iterative manner, as errors were discovered and subsequently corrected. The end result of this process is a series of cache policy implementations that are functionally valid and behave as expected.

University of Cape Town

8.4.2 Common Validation Scenarios

There are three specific scenarios within the operation of the cache that require individual validation. Each scenario is associated with a specific subset of operations[2], and is common across all caching schemes. In addition, each cache algorithm has a series of constraints that must also be checked at varying points during execution. These details are listed in the individual cache validation descriptions in Section 8.4.3 (p. 114).

Initialisation

This refers to the period between starting a simulation, when the cache is empty, and the cache reaching a full state. During this period, any queries to the cache are automatically added (or updated if they are already present), but no destaging occurs. The following conditions must be checked in order to validate operation over this period:

- Each page in the cache is allocated as entries are added, such that there are no unused pages remaining once the cache is listed as full.

[2]These operations and their relation to the function of the RAID Controller are referenced in Figures 22 to 25 on Pages 59 to 61

- Added entries should not overwrite or modify other entries present in the cache.
- The number of pages allocated is never greater than the maximum permitted by the configured cache size.

Cache Hit

A cache hit occurs when one or more of the requested blocks in a read request currently exist in the cache. In this scenario, the following tasks must be performed, and hence tested for correctness:

- The associated IORequest should be modified so as to reflect that the associated disk blocks are present in the cache (and hence need not be fetched from disk). Unrelated fields in the IORequest should not be altered.
- The status information for the appropriate page should be correctly updated. This could refer to the recency (LRU) or frequency (LFU) status of the related page, or it could require moving the page between various tables (ARC).

Cache Update

A cache update occurs if one or more of the requested blocks in a read request does not exist in the cache (a Cache Miss) or if a write request is intercepted. In either case it will be necessary to create space in the cache to accommodate the new data blocks. This entails freeing, and possibly destaging, pages from the cache and adding the new blocks to the cache. In this scenario it is necessary to validate the following steps:

- The page that is freed should match the conditions imposed by the caching scheme, e.g. the least-recently used page should be freed under LRU.
- If the dirty bit is set, indicating the cache page has been modified by a write operation, the contents of the freed page should be correctly destaged to disk.
- If the cache update is triggered by a write request:
  - If write-through caching is enabled, the new blocks should simultaneously be written to disk.
  - If write-back caching is enabled, the dirty flag on the associated cache page should be set.
- The status information for the freed page should be correctly reset. This prevents errors where new page data is treated as old.
- The new data blocks should occupy the same cache location as the recently freed page. The referenced cache location should be valid and within the size of the cache. This enforces a locality constraint on the cache update operation.
- After each Cache Update operation, the cache should be full. This checks for lost or missing pages and prevents the cache equivalent of memory leaks.

University of Cape Town

8.4.3 Cache Policy Dependent Validation

In addition to the general cache validation outlined above, it is necessary to perform cache policy specific validation to account for the differences in operation between the policies. The steps involved in this validation test various unique constraints and conditions associated with each policy, as well as checking for certain expected behaviours at boundary conditions. The approach taken for each implemented cache policy is outlined below.

LRU

The Least Recently Used (LRU) cache policy specifies that the cache page with the oldest timestamp should be destaged. It therefore follows that the primary validation condition should check that the pages freed satisfy this criteria. This is easily checked by examining the cache trace output before and after each update to determine that the cache page freed was also the least recently used.

The cache behaviour induced by this policy is best validated by examining three boundary conditions related to the provided workload. For each of these boundary conditions, a series of test I/O workloads were prepared that provided the expected cache workload, and the actual cache behaviour was compared to the expected one. The nature of these workloads, and the corresponding expected behaviour is detailed below:

- When provided with a sequential workload[3], we expect to observe cache cycling as no data is repeatedly used. Specifically, we expect the position of the freed cache page to mimic the workload: the index of the freed page increases sequentially with each new data block request, cycling back to 0 once the end of the cache is reached.
- A repeating sequential workload is a sequential workload with a finite size that is repeated several times. We consider specifically a repeating sequential workload that requests exactly as many disk blocks as there are pages in the cache. In this case, it is expected that after the initialisation phase the cache contents will remain static, as no additional data is added and hence no pages need be freed. This is the best-case scenario for the LRU scheme.
- A refinement of the previous boundary condition uses a repeating sequential workload with length = cache size + 1. This is the worst-case scenario for the LRU scheme, in which the cache is expected to continuously free pages as the frequently used set is just larger than the cache itself.

LFU

The Least Frequently Used (LFU) cache policy specifies that the cache page with the fewest number of accesses should be destaged. As with LRU above, this constraint can be checked by examining the cache trace output. In addition, the following boundary conditions and the associated expected behaviours were tested:

[3] See *Sequential Source* in Section 5.6.1 (p. 75)

University of Cape Town

- When presented with a repeating sequential workload with length greater than the cache size, an LFU cache is expected to exhibit LRU-like behaviour, as the least-recently used block is also the least-frequently used.
- When presented with a hotspot[4] workload where the number of frequently used blocks is less than half the cache size, the cache should contain a set of static pages (representing the frequently used data) as well as a set of frequently freed pages (representing the randomly accessed data). The persistence of the static pages in the cache is the validation criteria for this boundary condition, which is the best-case scenario for LFU.
- The final boundary condition is characterised by an initial repeating sequential workload consisting of cache size - 1 blocks. After several repetitions of this sequence, the workload changes to a random one. This is the worst-case scenario for LFU, as the cache is initially polluted by the frequently used blocks in the sequential workload, which persist in the cache long after they have actually been used. In this case we expect to observe thrashing in the cache, as the random workload has only 1 effective free page in which it can be cached.

ARC

The Adaptive Replacement Cache (ARC) policy attempts to adapt the caching scheme dynamically in order to make best use of both recency and frequency information. ARC has a number of parameters that are tuned by the algorithm during execution. This added complexity implies that the simple validation performed for the above two policies is insufficient in this case. A brief overview of the ARC scheme is presented in 2.5.1.

We have developed a GUI validation tool (see Figure 62) to account for this that is able to parse the cache trace output and visualise the state of the cache. It also monitors the ARC constraints across cache state changes, and raises an alert when one is violated. This tool greatly simplified the validation of the ARC scheme.

Given a cache size, C, the ARC constraints that must hold at all times are:

- $T1 + T2 = C$
- $T1 + T2 + B1 + B2 = 2 \times C$
- $T1 \leq C$, $T2 \leq C$, $B1 \leq C$, $B2 \leq C$
- $T1$ should converge to the target value $targetT1$

where $T1$, $T2$, $B1$, $B2$, and $targetT1$ are the parameters dynamically tuned by the ARC scheme[5].

In addition to the above constraints, the following boundary conditions were tested:

- When presented with a purely sequential workload, we expected T1 (the recency portion of the cache) to expand to the entire cache size while T2 (the frequency portion) remains empty. Further, the behaviour of T1 is expected to be identical to the behaviour of an LRU cache subjected to the same workload.

[4] See *Hotspot Source* in Section 5.6.1 (p. 75)
[5] See Section 2.5.1 (p. 18) for details on the ARC scheme.

University of Cape Town

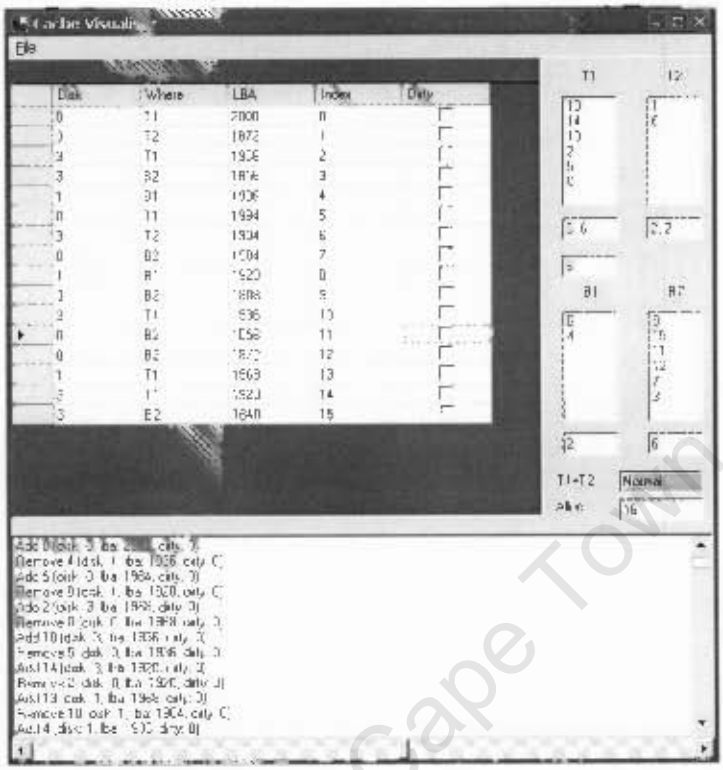


Figure 62: Screenshot of the ARC GUI tool developed to assist in validation of ROSTI's ARC implementation.

- When presented with repeating sequential workload with length equal to the cache size, we expect T2 to fill up and expand to the entire size of the cache, while T1 remains empty. This follows as ARC attempts to maximise the amount of cache frequency information used.
- When presented with the same hotspot workload as LFU above, we expect T2 to expand to contain exactly those blocks corresponding to hotspots, while T1 cycles through the random blocks that are requested.
- When presented with a random workload, we expect T1 and T2 to fluctuate as the workload changes, which is in keeping with the adaptive nature of ARC.
- Finally, a RAID workload consisting of small sequential writes produces an interesting behaviour. Given that the writes are small, we expect a single parity block on disk to be accessed for each write to that stripe. At the same time, the data blocks are accessed comparatively infrequently. Thus, we expect T2 to fill with the parity blocks being updated, while T1 fills with the data blocks that are being written.

University of Cape Town

Chapter 9

Test Case

University of Cape Town

An integral part of the validation of RÖSTI involved specifying and executing simulations of simplified RAID systems. These experiments replicated the use of RÖSTI in actual simulation studies, and tested that RÖSTI adequately fulfills the requirements of its intended use. The results presented below are from one such experiment.

9.1 RAID Simulation Model

The system we chose to simulate consisted of a simple RAID setup with a single data source, one RAID Controller, and several disk drives. The configured RÖSTI simulation appears in Figure 63.

The RAID Controller was configured to use a Strip size of 64KB, with 16 Chunks per Strip, resulting in a Chunk size of 4KB. The RAID mode being simulated was an independent parameter of the simulation that was alternated between a classic RAID 5 scheme and RAID 5 with SPIDRE. No caching was enabled for this experiment, and the RAID system was only simulated in Normal mode operation (ie. no disk failures were simulated).

9.2 Workload

The workload for this simulation was provided by a single, synthetic, Random Source[1], configured to generate 100 000 I/O requests, uniformly distributed across the available addressable disk space, with a mean size for each request of 4KB. The I/O requests were alternated between only reads and only writes for each simulation run. The time between requests (the Inter-Arrival Time) was drawn from a Normal distribution with a mean value that was a second independent parameter of the simulation.

[1] See 5.6.1 (p. 74)

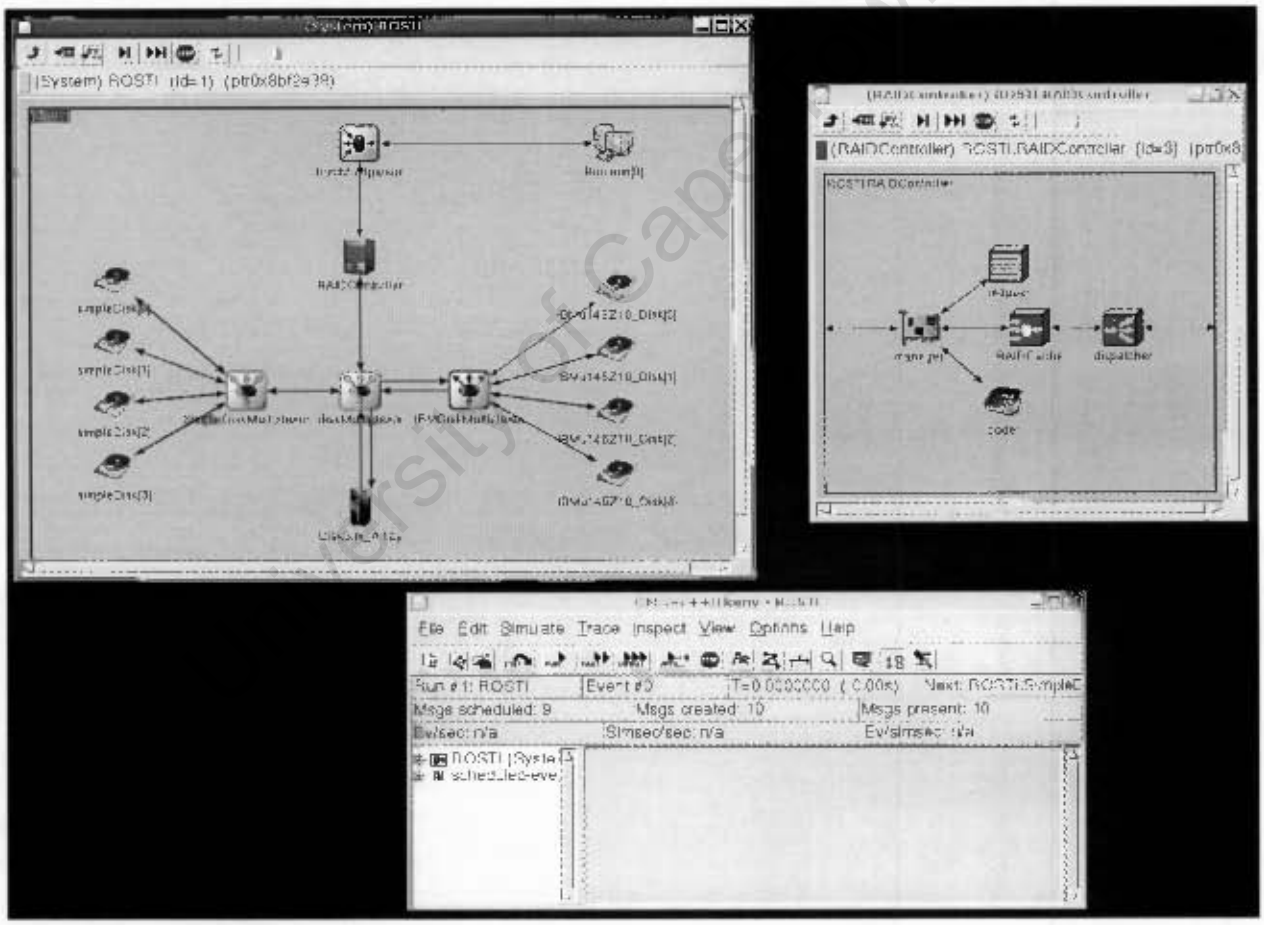


Figure 63: Screenshot of the configured RÖSTI test model prior to simulation. The window on the top left represents the top-level RAID system model. The window on the top right shows the view inside the RAID Controller component. The window at the bottom is the RÖSTI control window, which displays status information and provides control over the simulation.

9.3 Disk Drives

Three different drive simulators were used in the experiments, the details of which are described in 5.6.2 (p. 76), and the choice of drive used was a third independent parameter of the simulation. The drives used were: an array of 4 Simple Disks, with a mean response time of 4ms; an array of 4 IBMu146Z10 drives; and a DiskSim array of 4 IBM18ES drives.

9.4 Definitions

- The **Inter-Arrival Time** (IAT) between requests is the time between successive requests arriving at the RAID Controller. In this simulation, it is determined by drawing values from a Normal distribution with a specified Mean value (mIAT).
- The **Response Time** of the system is the average elapsed time between a request arriving at the RAID Controller and the Controller issuing a reply to the data source indicating completion of the request.
- The **Minimum Response Time** is the lowest possible value of the system Response Time, and is determined by the ability of the simulated system to process a request in the absence of resource contention or queueing delays. In this instance, this time is primarily determined by the physical limitations of the disk drives in the system [RW94], as the effect of command overhead and processing is assumed to be minimal.

9.5 Simulation Expectations

For large values of mIAT, the system is effectively processing isolated requests, and thus the measured Response Time of the system should be equal to the Minimum Response Time. This value is known for the Simple Disk array, as we have specified a 4ms mean Response Time as a parameter to this disk model. Parity between this specified parameter and the measured Response Time from the simulation is the first validation criteria we checked.

As the specified mIAT approaches the Minimum Response Time of the system (the saturation point), we expect the measured Response Time to grow exponentially. This follows since requests are arriving at the RAID Controller faster than they can be processed, and hence queueing effects start to dominate the behaviour of the system. This provides the second validation criteria for the system: the value of mIAT at which the measured Response Time begins to grow exponentially should be comparable to the value of the measured Response Time at high values of the mIAT.

Another differentiating feature is expected between simulations conducted with a workload of purely read requests (Figure 64) and those with a workload of purely write requests (Figure 65). For the read workload, both the RAID 5 and RAID5–SPIDRF configurations are expected to perform similarly, since neither needs to access parity information.

For the write workload, both configurations need to update parity information on a different disk, which results in overhead in processing this request, as well as generally requiring a read-modify-write cycle[2]. This is expected to manifest as a higher Minimum Response Time for the write

[2]See 40 (p. 100)

University of Cape Town

workload than for the comparable read workload, with a minimum increase of 200% (owing to twice as many required operations).

An additional discrepancy should be evident between the RAID5 and RAID5+SPIDRE configurations, due to the additional parity update requirements of SPIDRE. This overhead should be measurable as an increased Minimum Response Time for the RAID5+SPIDRE configuration versus the default RAID5 configuration.

9.6 Simulation Results

The results presented here represent the output of various RÖSTI simulation runs executed with the above defined configuration. The results are first partitioned by workload: simulation runs conducted with read-only workloads are illustrated in Figure 64 and write-only workloads are illustrated in Figure 65.

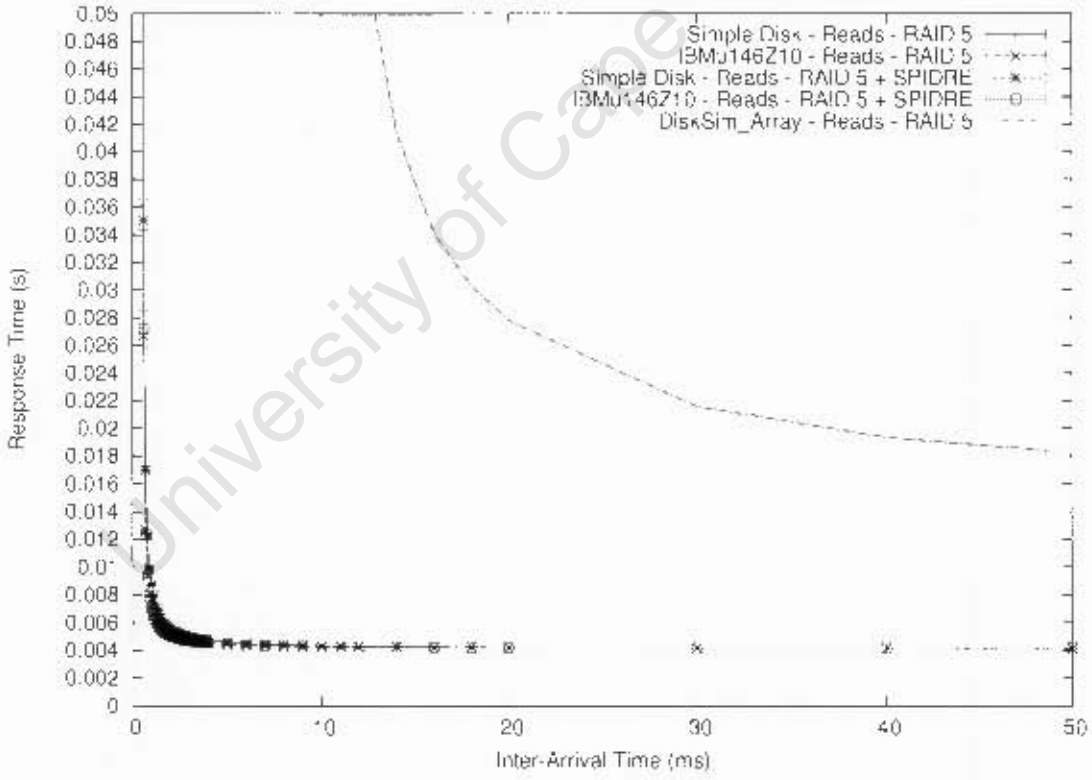


Figure 64: Mean Response Times for each configuration when driven by a *read-only* workload.

Within these partitions, the results are further differentiated by the type of disk array used and the RAID scheme implemented. Each combination of these two parameters is represented by a separate line in the graphed results.

The horizontal axis of both graphs represents the mean IAT, specified as a simulation parameter. The vertical axis represents the mean Response Time of the system, an average of the measured response time for all requests for that mIAT.

University of Cape Town

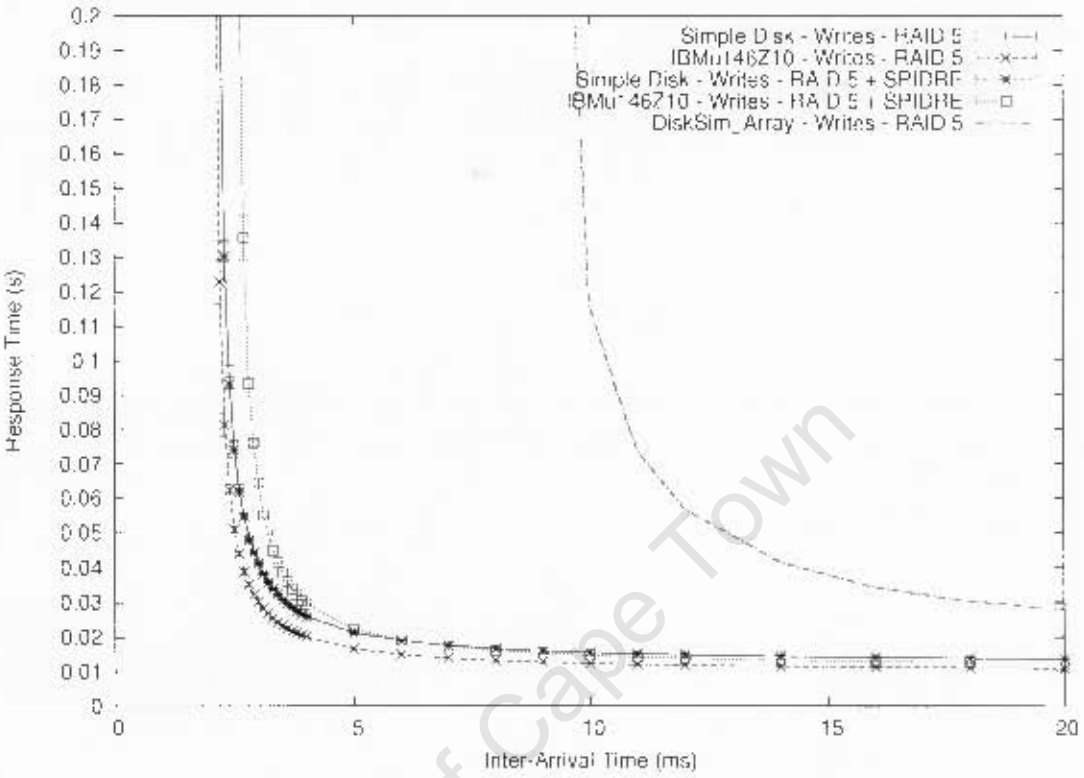


Figure 65: Mean Response Times for each configuration when driven by a *write-only* workload.

University of Cape Town

9.7 Simulation Analysis

The first curves in Figure 64 show that apart from the case for the DiskSim array drives, all other drive simulators yield the same results. Note that the response time is almost flat at 4 milliseconds which, in the case of the Simple_Disk model is the mean response time and at low inter-arrival rates is expected. The fact that the DiskSim graph is so much different is due to the greater detail in the simulation and that the performance of this rather old drive is relatively very slow.

For read requests, the saturation IAT is almost 8 times lower than the ResponseTime at low IAT. This result makes sense, in that the request sizes are small (fitting within 1 strip), thus all 8 disks can be servicing read requests simultaneously. This suggests that the capacity of the system (in terms of IAT) should be 8 times smaller than the Response Time at large IAT, which is observed.

The second set of curves in Figure 65 are more interesting in that the overhead that SPIDRE coding incurs on the performance is more clearly evident. As expected, the mean Response Time of both RAID configurations is measurably higher for the write-only workload. Additionally, there is the expected measurable discrepancy between the configurations with and without SPIDRE, which is particularly noticeable in the IBMu146Z10 results (the more accurate disk model). Finally, the relationship between the Minimum Response Time (as predicted at high values of mIAT) and the saturation point is found to be as expected.

Chapter 10

Summary

University of Cape Town

In this work, we have focused on the modelling and simulation of Redundant Array of Independent Disks (RAID) storage systems. RAID systems allow organisations to leverage large storage capacities and parallel data accessibility using low-cost, commodity hard drives. Given the value of such systems, a simulation environment tailored to RAID systems would allow prototypes to be quickly evaluated without the overhead of physically implementing them. With this in mind, the RÖSTI project was begun with the stated goal of developing an extensible simulation environment that could be used to implement and test the performance of new RAID schemes. A secondary goal was to increase the usability of the system for a novice end user, both in terms of simulation configuration and analysis.

10.1 Overview

Part I of this thesis presents the relevant background to the topic under consideration. Chapter 2 presents extensive coverage of the principles and operation of RAID storage systems, a taxonomy of current RAID schemes, and a brief summary of some of the issues facing RAID systems. Chapter 3 provides an overview of the field of simulation, and the considerations involved in developing a custom simulation environment. The decision to use OMNet++ as the simulation library foundation on which RÖSTI is built is discussed, in comparison to CSIM as an alternative option. An examination of available simulation environments in Chapter 4 helped identify useful functionality where RÖSTI could provide improvement, particularly in regards to extensibility and ease-of-use of the simulation environment.

Part II represents the main body of work that is the subject of this thesis. Chapter 5 details the development process of RÖSTI as a simulation environment. It covers the use of Unified Modelling Language (UML) to obtain user requirements using industry best practices. These UML requirements were then transformed into high-level system architectures, guided by the principles of modularity and reusability. The system architectures were translated into OMNet++ specific designs, which were subsequently implemented. RÖSTI's current support for various RAID schemes and capabilities is discussed, together with the problems encountered during implementation, and their solutions.

Having created RÖSTI, the secondary goal of usability was addressed. Chapter 6 describes the

design and development of a Graphical User Interface (GUI) to RÖSTI for configuring and executing RAID simulations. While relatively restricted in terms of capabilities, this GUI represents an important first step in improving usability of the system, particularly for casual users. The next logical step was to provide rudimentary data analysis functionality for the results produced by a typical RÖSTI simulation run. The development of this tool, and its current capabilities are covered in Chapter 7.

Part III of this thesis covers the testing performed on RÖSTI to ensure the validity of the system and component models that have been implemented. Chapter 8 presents the rationale behind omitting validation of the disk models used in RÖSTI, and the limited validation performed on the data source modules. Given the central importance of the RAID Controller, an extensive validation of each of the implemented RAID schemes was performed, and is documented in Section 8.3 (p. 97). The other important component of the system is the caching model, and the testing process for each of the three implemented caching schemes (LFU, LRU, ARC) is described in Section 8.4 (p. 111). Chapter 9 presents an exemplar simulation experiment of RÖSTI being used to investigate a RAID system. The simulation results are found to match expectations. further validating the implementation of RÖSTI.

University of Cape Town

10.2 Outcomes

Having researched the fields of simulation and RAID storage, we were able to develop the RAID Operations Simulator for Testing Implementations (RÖSTI). RÖSTI is a RAID-oriented simulation environment, built in C++ on the OMNet++ library and designed to be modular and extensible. This environment serves as the foundation on which implementations of three RAID protection schemes (RAID 5, RAID 6, SPIDRE) were implemented. Simulation input is accepted via trace files or one of three implemented synthetic workload generators. Several disk implementations are provided, most notably through the DiskSim package. Users can interact with RÖSTI using either a command-line interface, or a Graphical User Interface (GUI) provided by OMNet++. The architecture of RÖSTI allows for any combination of RAID schemes, workloads and disks to be simulated on either Linux (TM) or Microsoft Windows (TM).

The use of RÖSTI is simplified through a secondary GUI interface, implemented in C#, which hides the initial complexity of RÖSTI and OMNet++. This interface allows novice users to configure and execute simulations without requiring knowledge of OMNet++, as well as automating execution of multiple simulation runs. The capabilities of RÖSTI were further extended by the development of a results-analysis module that integrates with the above GUI. This allows users to generate 2-dimensional graphs plotting independent simulation parameters against dependent simulation results. This analysis is useful as initial data analysis tools to fine tune simulation parameters, or establish the approximate accuracy of a simulated model.

These capabilities fulfill RÖSTI's initial primary goal of providing a modular, extensible RAID simulation environment, and the secondary goal of improving the usability of the software.

In addition we also developed a validation strategy for RAID implementations focused on high-level behaviour rather than low level implementation[1]. This strategy greatly simplified the implementation and testing of RÖSTI. It allowed specific aspects of controller functionality to be tested independently, and the sum of all such related tests ensured the validity of implementation

[1] See Section 8.3 (p. 97)

of each RAID scheme. By analogy, this system fulfilled the same role that unit tests currently fill in commercial-grade software projects.

Finally, we have outlined a possible avenue of research into a formalisation strategy for RAID Controller operations[2]. This strategy would allow the individual disk operations required for a given array operation to be automatically inferred from a high-level description of the RAID scheme (RAID 5, RAID 6, etc.) in use. If successful, this would provide a less error-prone and more efficient means of implementing theoretical RAID protection schemes, either in software or hardware.

10.3 Future Work

In completing the first version of RÖSTI, several opportunities for future improvements to the system were identified. The first stems from RÖSTI's focus on RAID, which concentrated the majority of the development effort on the implementation of several RAID Controller modules. This focus has meant that both the data sources and disk models provided are lacking in certain capabilities.

The synthetic data sources, which serve as input to drive a RÖSTI simulation, are limited in terms of the complexity of workloads they can generate. They currently only support generation of random workloads that can be expressed as one of the statistical distributions listed in Appendix C. More realistic input workloads can be achieved by implementing and integrating a synthetic workload generator that uses data from real-world trace files. ESSWA[3] is just such a tool that was recently developed in the DNA group at the University of Cape Town as an MSc thesis by Paul Sikalinda [Sik06]. Integration with ESSWA would greatly extend the capability of RÖSTI to simulate more realistic workloads.

The disk drive models could also be updated to allow simulation of recent commodity drives. Models of modern disk drives can be added to DiskSim using the DIXTrac utility, but this has not been released by its creators. An alternate option is to use the work of Lee et. al. [LK93] on disk drive modelling to alter the parameters of the included disk models, or implement an entirely new disk model.

RÖSTI currently lacks support for simulating the rebuilding of a RAID array after data loss. Although this occurs infrequently compared to other actions in a RAID array (notably operation in Normal or Degraded mode), it is still of importance in systems where the length of downtime is important. Hence extending RÖSTI to provide these capabilities would add value to the system as a whole.

Another possible area for improvement is the Graphical User Interface to RÖSTI. At present, it places restrictions (which do not exist in RÖSTI) on the types of RAID architecture that can be configured. This was necessary to simplify its implementation. It would be useful to provide a structured, visual design environment to allow any possible architecture to configured and simulated. It is envisioned that this environment would be similar to that provided by OMNet++, but customised with regards to the specific requirements of RAID architecture.

[2] See Appendix A (p. 133)

[3] Enterprise Storage System Workload Analyser

University of Cape Town

The results analyser provided with RÖSTI is currently limited. This is a particularly important aspect of a simulation study, and it may thus be useful to extend this tool to allow for more complex analysis to be performed on output from RÖSTI. This could extend to more elaborate graphing support, statistical analysis of the output or even comparison with analytical models.

A final possible improvement relates to the technical implementation of RÖSTI's cache model. As it presently stands, multi-level RAID hierarchies are representable in RÖSTI[4], but each RAID level has its own independent cache. Thus although a multi-level caching system is present, there is no explicit support for cache coherency across these caches. This is particularly problematic where write-through caching is not used. Implementing support for the notion of cache-coherency in RÖSTI would greatly benefit the development of these hierarchical models.

University of Cape Town

[4]See Section 5.4.4 (p. 68)

Bibliography

[AAD97] M. Aboutabl, A. Agrawala, and J. Decotignie. Temporally determinate disk access: An experimental approach. Technical Report CS-TR-3752, Dept. of Computer Science, University of Maryland, College Park, 1997.

[ABC97] Guillermo A. Alvarez, Walter A. Burkhard, and Flaviu Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 62–72. IEEE Computer Society Press, 1997.

[Anv07] H. Peter Anvin. The mathematics of RAID 6. `http://www.cs.utk.edu/~plank/plank/papers/CS-07-602.pdf`, 2007.

[BBBM94] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd: An optimal scheme for tolerating double disk failures in raid architectures. In *Proceedings of the 21st Symp. on Computer Architecture*, pages 245–254, 1994.

[Bea03] J. S. Bucy and G. R. Ganger et al. The DISKSim simulation environment. Technical Report CMU-CS-03-102, Parallel Data Laboratory, Carnegie Mellon University, January 2003.

[BG88] Dina Britton and Jim Gray. Disk shadowing. HP Labs Technical Reports, June 1988.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[Bro05] Neil Brown. mdadm. `http://cgi.cse.unsw.edu.au/~neilb/mdadm`, 2005.

[CG94] William V. Courtright II and Garth A. Gibson. Backward error recovery in redundant disk arrays. In *Proceedings of the 1994 Computer Measurement Group Conference (CMG)*, volume 1, pages 63–74, December 1994.

[CGHZ96a] William V. Courtright II, Garth Gibson, Mark Holland, and Jim Zelenka. Raidframe: rapid prototyping for disk arrays. In *Proceedings of the 1996 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, volume 24, pages 268–269, May 1996.

[CGHZ96b] William V. Courtright II, Garth Gibson, Mark Holland, and Jim Zelenka. A structured approach to redundant disk array implementation. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS)*, 1996.

University of Cape Town

[CGR+97] William V. Courtright II, Garth Gibson, LeAnn Neal Reilly, Mark Holland, and Jim Zelenka. *RAIDframe: rapid prototyping for RAID systems.* Parallel Data laboratory, Carnegie Mellon University, CMU-CS-97-142 edition, June 1997.

[Cou97] William V. Courtright II. A transactional approach to redundant disk array implementation. Master's thesis, Carnegie Mellon University, 1997.

[CP90] P. Chen and D. Patterson. Maximizing performance in a striped disk array. In *Proceedings of ACM SIGARCH Conference*, pages 322–331. ACM, 1990.

[dIJ02] M. de Icaza and B. Jepson. Mono and the. Net framework. *Dr. Dobbs Journal of Software Tools*, 27(1):21–24, 2002.

[EPM99] G. Ewing, K. Pawlikowski, and D. McNickle. Akaroa2: Exploiting network computing by distributing stochastic simulation, 1999.

[GM00] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, 2000.

[HG92] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 23–35, October 1992.

[HGK+94] Lisa Hellerstein, Garth A. Gibson, Richard M. Karp, Randy H. Katz, and David A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2/3):182–208, 1994.

[HGS93] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing*, 1993.

[HGS94] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Architectures and algorithms for on-line failure recovery in redundant disk arrays. *Journal of Distributed and Parallel Databases*, 2(3):295–335, July 1994.

[HLM+97] M. Harrison, D. Libes, M. McLennan, J. Ousterhout, T. Poindexter, M. Roseman, L.A. Rowe, B. Smith, M. Ulferts, A. Brighton, et al. *Tcl/Tk tools.* O'Reilly & Associates, Inc. Sebastopol, CA, USA, 1997.

[Hol94] Mark Calvin Holland. On-line data reconstruction in redundant disk arrays. Master's thesis, Carnegie Mellon University, 1994.

[HZ04] P. Harrison and S. Zertal. Calibration of a queueing model of RAID systems. In *Proceedings of Practical Application of Stochastic Models*, page to appear, Imperial College, London, September 2004.

[IBM05] IBM. Ess product brochure. `http://www-03.ibm.com/servers/storage/disk/ess/pdf/ess-brochure.pdf`, 2005.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS.* ACM, November 2000.

University of Cape Town

[KS95] A. Kuratti and W. H. Sanders. Performance analysis of the RAID5 disk array. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 236–245, Erlangen, Germany, 24–26 1995.

[KTR94] David Kotz, Song B Toh, and Sriram Radhakrishnan. A detailed simulation model of the hp 97560 disk drive. Technical report, Dartmouth College, Hanover, NH, USA, 1994.

[Leh51] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery, Cambridge, MA, 1949*, pages 141–146, Cambridge, MA, 1951. Harvard University Press.

[LK82] Averill M. Law and David W. Kelton. *Simulation Modelling and Analysis.* McGraw-Hill, Inc., 1982.

[LK93] Edward K. Lee and Randy H. Katz. An analytic performance model of disk arrays. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 98–109. ACM Press, 1993.

[LKB87] Miron Livny, Setrag Khoshafian, and Haran Boral. Multi-disk management algorithms. In *Proceedings of the 1987 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 69–77. ACM Press, 1987.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: a process-oriented approach. *IEEE Trans. Softw. Eng.*, 18(6):488–497, 1992.

[Mil03] Randy Miller. Practical uml: A hands-on introduction for developers. Borland Developer Network, `http://bdn.borland.com/article/0,1410,31863,00.html`, December 2003.

[MM92] J. Menon and D. Mattson. Performance of disk arrays in transaction processing environments. In *12th International Conference on Distributed Computing Systems*, pages 302–309, 1992.

[MM03a] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *USENIX File & Storage Technologies Conference (FAST)*, 2003.

[MM03b] Nimrod Megiddo and Dharmendra S. Modha. A simple adaptive cache algorithm outperforms lru. IBM Research Report RJ 10284, IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, February 2003.

[MN98a] M. MATSUMOTO and T. NISHIMURA. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[MN98b] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[Ous94] J. K. Ousterhout. *Tcl and the Tk toolkit.* Professional Computing series. Addison-Wesley, April 1994.

University of Cape Town

[Pan95] O.A. Panfilov. Performance analysis of raid-5 disk arrays. In *28th Hawaii International Conference on System Sciences (HICSS'95)*, 1995.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

[Pla97] James S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software: Practice & Experience*, 27(9):995–1012, 1997.

[PLK02] E. J. Chen P. L'Ecuyer, R. Simard and W. D. Kelton. An objected-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

[Pon93] György Pongor. Omnet: Objective modular network testbed. In *MASCOTS '93: Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 323–326. Society for Computer Simulation, 1993.

[Poo02] Frank W. Poole. A look at modular raid on motherboards. `http://www.intel.com/design/network/papers/25157501.pdf`, 2002.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1991. ACM Press.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[SCO90] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990. USENIX Association.

[SG99] J. Schindler and G.R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, School of Computer Science, Carnegie Mellon University, December 1999.

[SG00] J. Schindler and G.R. Ganger. Automated disk drive characterization (poster session). *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 112–113, 2000.

[SGG02] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, 2002.

[SGH93] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 64–75. ACM Press, 1993.

University of Cape Town

[SHG93] Daniel Stodolsky, Mark Holland, and Garth A. Gibson. A redundant disk array architecture for efficient small writes. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[Sik06] Paul Sikalinda. Esswa: Enterprise storage system workload analyser. Master's thesis, University of Cape Town, 2006.

[SO91] Jon A. Solworth and Cyril U. Orji. Distorted mirrors. In *Proceedings of the first international conference on Parallel and distributed information systems*, pages 10–17. IEEE Computer Society Press, 1991.

[SSM+] Julian Satran, Daniel Smith, Kalman Meth, Ofer Biran, Jim Hafner, Costa Sapuntzakis, Mark Bakke, and et al. iscsi.

[ST70] J. W. Schmidt and R. E. Taylor. *Simulation and Analysis of Industrial Systems.* Irwin, Homewood, Illinois, 1970.

[UAM01] M. Uysal, G. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. of the 9th Intl. Symp. on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS)*, pages 183 – 192, August 2001.

[VK03] A. Veitch and K. Keeton. The Rubicon workload characterization tool. SSP technical report HPL-SSP-2003-13, HP Laboratories, March 2003.

[VLW97] Mandana Vaziri, Nancy A. Lynch, and Jeannette M. Wing. Proving correctness of a controller algorithm for the RAID level 5 system. In *Symposium on Fault-Tolerant Computing*, pages 16–25, 1997.

[WBW89] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: a responsibility-driven approach. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 71–75, New York, NY, USA, 1989. ACM Press.

[WGP95] B. L. Worthington, G. R. Ganger, and Y. N. Patt. On-line extraction of SCSI disk drive parameters. In *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 146–156, 1995.

[Wil96] J. Wilkes. The Pantheon storage-system simulator, version 1. SSP technical report HPLSSP9514 revision 1, Storage Systems Program, HP Laboratories, Hewlett-Packard Laboratories, Palo Alto, CA, May 1996.

[YG99] Haruo Yokota and Masanori Goto. Fbd: A fault-tolerant buffering disk system for improving write performance of raid5 systems. In *PRDC '99: Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, page 95, Washington, DC, USA, 1999. IEEE Computer Society.

[Yok00] H. Yokota. Performance and reliability of secondary storage systems. In *World Multiconference on Systemics, Cybernetics and Informatics*, pages 668–673, July 2000. invited paper.

[ZMR96] P. Zabback, J. Menon, and J. Riegel. The RAID configuration tool. In *Proceedings of Third International Conference on High-Performance Computing*, HiPC, pages 55–61, Trivandrum, INDIA, December 1996.

University of Cape Town

[ZRM96] P. Zabback, J. Riegel, and J. Menon. The RAID configuration tool. Research Report RJ 10055, IBM Research Division, Almaden, November 1996.

University of Cape Town

Part IV

Appendices

University of Cape Town

Appendix A

Formalising Controller Operation

University of Cape Town

This chapter outlines a proposal for future work that would formalise the characteristics of a given RAID level using dependency relationships. Using this formalism, it would be possible to implement the controller logic for such a defined RAID level automatically, by determining the affected chunks and applying a standard set of operations to these chunks.

A.1 Raid Layout Specification

The layout of a RAID Array refers to the number and arrangement of Stripes, Strips and Chunks in the array[1]. In general, this amounts to defining the type of each chunk as being either data (the chunk contains user data) or parity (data protection information). One approach is to hard code an algorithm based on the scheme in use.

Our proposed approach is to differentiate between three types of parity:

- Horizontal: All parity chunks lie in a single row for each stripe.
- Vertical: All parity chunks lie in a single strip, for each stripe (eg. RAID 4).
- Diagonal: The position of parity chunks lie within a single strip, but the location of this strip rotates in a round-robin fashion (RAID 5 and RAID 6).

Using these parity types, it is possible to specify a number of types of parities to apply to a layout, together with an order in which to apply them. This automatically produces the correct layouts for any combination of Stripe, Strip and Chunk size settings. A proof of concept of this technique has been implemented, and was tested with combinations of all 3 types of parity. In all cases, the layout produced was correct.

[1] See Section 2.1 (p. 5).

A.2 Specifying Protection Groups

The usefulness of the above formalism becomes more apparent when we consider that a given type of parity protects data chunks in a consistent manner. Specifically, Horizontal parity chunks protects all data chunks in its Strip, while Vertical and Diagonal parity chunks protect all data chunks in their Row. This web of protection can easily be represented using a dependency graph, linking data chunks to the parity chunks that protect them. This graph can be built as the above layout process occurs, based on a simple set of rules. The use of a dependency graph also allows for more complicated dependency mappings, but using the same three parity types. Thus, we could conceive of a Horizontal parity that protects all data chunks that lie in the same diagonal.

A.3 Deriving Array Operations

The dependency graph created above can now be used to automate the operation of the RAID Controller. Specifically, if data needs to be reconstructed for a read operation due to a failed disk, the dependency graph can be used to determine the list of all parity chunks protecting the data chunks in question. Using this information, it is possible to determine the minimum number of disk accesses required to read in appropriate parity and recover the data, thus addressing the optimality issue raised earlier.

For a write, parity chunks need to be updated to reflect changes in data. Using the dependency graph, we can ensure that all parity chunks related to changed data chunks are correctly updated. We can even minimise the number of disk I/Os issued for certain operations (eg. Large Stripe Write).

Since the operations are extracted from the dependency graph, rather than coded by hand, it is possible to guarantee that operations are executed correctly and efficiently, under the assumption that the dependency graph is correctly constructed. Additionally, a trace of the operation of a controller using this scheme can be used to verify whether another, hand-coded controller is operating correctly.

University of Cape Town

A.4 Correctness of Operation

Although a RAID Controller may operate correctly during normal operation, it may still not handle error scenarios correctly. The most critical time during which errors can occur is if a disk failure happens while an I/O Request is being processed. The array may be left in an inconsistent state, and hand-coding of recovery schemes for all possible cases is a time-consuming and error-prone task. The work of Courtright [Cou97] presents a formal method to prevent this, using Directed Acyclic Graphs (DAGs). By using the dependency graph described above to automatically generate these DAGs, where possible, one can create an end-to-end formalism governing the operation of the RAID Controller.

Appendix B

RÖSTI Messages

B.1 IORequest

```
message IORequest
// Message mapping a host IORequest to physical array storage. Uses an array
// of partial stripes to represent this mapping, which in turn consists of
// arrays of strips which each contain 1 or more elements. This general
// representation allows any coding scheme to be both represented and simulated.
// Additional fields with the messages data members are used during execution of
// a request to signal any required operations on strips and the state of
// various elements.
{
    fields:
        int Volume;       // Logical Volume to which this request applies
        llong LBA;        // Starting address of request
        long Size;        // Size of request, in 512B blocks
        long ActualSize; // Actual number of bytes requested
        int Opcode enum(OpcodeEnum); // Operation requested (see OpcodeEnum)
        int Identifier;  // Unique identifier, used by host to keep track
                         // of outstanding requests
        int HostIdentifier; // Unique identifier, used to enumerate and
                            // differentiate hosts
        int Flags;        // For any specific bit type flags that
                         // may be required
};
```

University of Cape Town

B.2 IOResponse

```
message IOResponse
//Represents a response to a particular IORequest. Specifies the result of
// the request. Encapsulates the IORequest it references

//*Encapsulates IORequest
{
    fields:
        int Response enum(ResponseTypeEnum); // Result of requested operation
        int Identifier;     // Corresponding IORequest identifier
        int HostIdentifier; // Identifier for host that issued corresponding
                            // IORequest. Used in routing responses.
};
```

B.3 ArrayMapping

```
message ArrayMapping extends IntraArray
// Message mapping a host IORequest to physical array storage. Uses an array
// of partial stripes to represent this mapping, which in turn consists of
// arrays of strips which each contain 1 or more elements. This general
// representation allows any coding scheme to be both represented and simulated.
// Additional fields with the messages data members are used during execution
// of a request to signal any required operations on strips and the state
// of various elements.

//*Encapsulates IORequest
{
    fields:
        int DataLength;             // How many sectors (512B) is the request
        PartialStripe Stripes[];    // Array of partial stripes,
                                    //used to effect host-to-array mapping.
}


class PartialStripe
//Represents part (or a whole) stripe in a logical volume.
{
    fields:
        int StripeNumber; // Logical number of stripe (starting from 0)
        int StripSize; // Size of a strip in this stripe
        int ChunkSize; // Size of an element in any strip in this stripe
        PartialStrip Strips[]; // Array of strips in this stripe
}
```

University of Cape Town

B.4 BlockIO

```
message IntraArray
// Base class that provides fields common to all intra-array messages
// (messages that are only used within the simulation of the array-controller).
{
    fields:
        int ArrayMode enum(ArrayModeEnum);  // The mode the array is in
                                            // for the current operation
        int ArrayOp enum(OpcodeEnum);       // The overall operation being
                                            // performed by the array
}

message BlockIO extends IntraArray
//Indicates an internal request by the array controller to schedule
// multiple disk requests to either read or write data.

//*Encapsulates ArrayMapping
{
    fields:
        int Stage;  // Stage of completion for this request.
                    // (eg. RAID 5 read has 1 stage, RAID 5 write has 2 stages.
        int Identifier; // Used by array controller to enumerate all requests
}
```

B.5 BlockIOResponse

```
message BlockIOResponse extends IntraArray
//Indicates a response to a BlockIO request. Indicates to the coding module
// (which controls the stage of array operations, and subsequent actions)
// what the results, from each disk to which a request was issued, were.

//*Encapsulates BlockIO
{
    fields:
        int NumResults; // Number of results returned
                        // (and hence num IORequests to disks issued
        int Results[];  // Array of results for disks
        int Stage;      // Copied from BlockIO
        int Identifier; // Copied from BlockIO
}
```

University of Cape Town

Appendix C

Available Distributions

University of Cape Town

C.1 Discrete Distributions

Function	Description
intuniform(a, b, rng=0)	uniform integer from a..b
bernoulli(p, rng=0)	result of a Bernoulli trial with probability 0<=p<=1 (1 with probability p and 0 with probability (1-p))
binomial(n, p, rng=0)	binomial distribution with parameters n>=0 and 0<=p<=1
geometric(p, rng=0)	geometric distribution with parameter 0<=p<=1
negbinomial(n, p, rng=0)	binomial distribution with parameters n>0 and 0<=p<=1
poisson(lambda, rng=0)	Poisson distribution with parameter lambda

Table 6: Available Discrete Random Number Distributions for specifying workload parameters. Listed Distributions are provided by OMNet++.

C.2 Continuous Distributions

Function	Description
uniform(a, b, rng=0)	uniform distribution in the range [a,b)
exponential(mean, rng=0)	exponential distribution with the given mean
normal(mean, stddev, rng=0)	normal distribution with the given mean and standard deviation
truncnormal(mean, stddev, rng=0)	normal distribution truncated to nonnegative values
gamma_d(alpha, beta, rng=0)	gamma distribution with parameters alpha>0, beta>0
beta(alpha1, alpha2, rng=0)	beta distribution with parameters alpha1>0, alpha2>0
erlang_k(k, mean, rng=0)	Erlang distribution with k>0 phases
chi_square(k, rng=0)	chi-square distribution with k>0 degrees of freedom
student_t(i, rng=0)	student-t distribution with i>0 degrees of freedom
cauchy(a, b, rng=0)	Cauchy distribution with parameters a,b where b>0
triang(a, b, c, rng=0)	triangular distribution with parameters a<=b<=c, a!=c
lognormal(m, s, rng=0)	lognormal distribution
weibull(a, b, rng=0)	Weibull distribution
pareto_shifted(a, b, c, rng=0)	generalized Pareto distribution

Table 7: Available Continuous Random Number Distributions for specifying workload parameters. Listed Distributions are provided by OMNet++.

University of Cape Town