Proving Correctness of a Controller Algorithm for the RAID Level 5 System

by

Mandana Vaziri-Farahani

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1996

Author ....
Department of Electrical Engineering and Computer Science
August 9, 1996

Certified by ....
Nancy A. Lynch
Cecil H. Green Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Proving Correctness of a Controller Algorithm for the RAID Level 5 System

by

Mandana Vaziri-Farahani

Submitted to the Department of Electrical Engineering and Computer Science
on August 9, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

A RAID system is composed of two components: a disk array and a disk array controller. The disk array is a collection of magnetic disks that can be accessed in parallel. The controller's function is to receive an operation from the user of the disk array, and to carry out that operation by performing a set of actions on specific disks. The user has no knowledge about the existence of a disk array and sees it as one large, logical disk with high performance.

RAID systems have two main advantages over traditional secondary storage systems. First, data on the disks can be accessed in parallel, which improves the I/O performance. Secondly, disk arrays contain some form of redundancy which allows fault tolerance.

Many algorithms have been devised for the controller. These algorithms allow fine interleavings between actions on the disks. As a consequence, their implementations are difficult to test. This is the reason why it is useful to apply formal methods to validate these algorithms.

An algorithm for the RAID Level 5 controller is considered. The algorithm and its specification are described using I/O automata and the simulation proof technique is used to show that the algorithm implements its specification. The proof is written in such a way that its main structure and invariants can be reused for the proof of correctness of controller algorithms for other RAID architectures.

Thesis Supervisor: Nancy A. Lynch

Title: Cecil H. Green Professor of Computer Science and Engineering

Acknowledgments

First of all, I would like to thank my advisor Nancy Lynch for her wonderful supervision of this thesis. I started working on the verification of RAID systems in the summer before coming to MIT, and Nancy gave me the freedom to continue working on this research project. She indicated a direction for this research that has been very enriching to me. Her suggestions, comments and careful reading of previous drafts have made this thesis possible. I would also like to thank my former advisor, Jeannette Wing, for all her help and encouragement, and for initiating the project on verifying RAID systems. I wish to thank Bill Courtright who was always very patient with answering my numerous questions about RAID.

Then, I would like to thank everyone in the Theory of Distributed Systems group for being so supportive and creating such a warm environment in which to work. Special thanks go to Roberto De Prisco, Victor Luchangco, and Giovanni Della-Libera.

But above all, I would like to thank my parents Chahnaz and Faramarz Vaziri, and my brother Shahram Vaziri, for all their love and support during all these years, without which this thesis would have never been possible.

Contents

List of Figures

Chapter 1

Introduction

Improvements in semiconductor technology make possible faster microprocessors and larger primary memory systems, making secondary storage systems the bottleneck of overall system performance. As microprocessors get faster, the overall system improvement will not be significant unless there is also an improvement in secondary storage systems.

The emergence of new applications such as video, hypertext and multimedia has also increased the need for larger secondary storage systems with higher performance. RAID or Redundant Arrays of Inexpensive Disks were developed in the 1980's to address this need. They were first described at the beginning of the decade [Lawlor81, Park86], and popularized by the work of a group at UC Berkeley [Patterson88, Patterson89].

Abstractly, we can think of a RAID system as being composed of two components:

- A disk array, and
- A disk array controller.

The disk array is a collection of magnetic disks that can be accessed in parallel. The controller's function is to receive an *operation* from the user of the disk array, and to carry out that operation by performing a set of *actions* on specific disks. The user has no knowledge about the existence of a disk array and sees it as one large, logical disk with high performance.

RAID systems have two main advantages over traditional secondary storage systems. First, the data on the disks can be accessed in parallel which improves the I/O performance. Each file that is stored in the array is decomposed into blocks and placed on several disks. This scheme improves the response time when the user accesses that file [Kim86, Reddy89, Salem86]. The controller can also carry out several operations at the same time if the set of disks involved in these operations are non-conflicting. This scheme improves the throughput.

Secondly, when the number of disks increases in a disk array, the availability of data and the reliability of the disk array, may decrease dramatically [Gibson93]. To overcome this problem, RAID systems are designed to be fault-tolerant by storing redundant data [Gibson90]. RAID systems are usually 1 or 2 fault tolerant. The redundancy can be an identical copy of each data unit, also known as disk mirroring [Bitton88, Gray90]. In this case if the disk containing one copy fails, the controller can use the other copy which is on a separate disk. Having two copies of each data unit also has the advantage that if the disk containing one copy is busy with a different operation, the other disk can be used instead, improving throughput. In this form of redundancy, lost or damaged data can be recovered by using the backup copy.

Another form of redundancy is having a parity block for every group of n blocks, independently stored [Patterson88]. The parity block is computed by performing an exclusive or operation on the blocks it covers. Given any set of $n-1$ blocks, the nth block can be recovered by performing an exclusive or operation on the $n-1$ blocks.

There are several RAID architectures that are classified as five "levels" [Patterson88]. RAID Level 1 employs disk mirroring and thus uses twice as many disks as a non-redundant disk array for the same amount of data. RAID Level 2 provides redundancy by using Hamming codes. Levels 3, 4 and 5 all use parity. RAID Level 3 is bit-interleaved meaning that data is interleaved bit-wise over the data disks. RAID Level 4 is block-interleaved. RAID Level 5 is also block-interleaved, but distributes parity among all the disks in the array. All the architectures mentioned above tolerate a single fault. Recently, two other levels have been introduced. The first one is RAID Level 6 which is a two fault-tolerant architecture. It employs two parities, one of which is computed using Reed-Solomon codes. The second one is just a non-redundant disk array, RAID Level 0, which is not fault-tolerant.

When the disk array controller receives an operation from the user, it chooses a local algorithm to carry out that operation given the state of the disk array. If a disk is needed during an operation but that disk has failed, then the controller is responsible for recovering the data needed, transparently to the user. If a disk fails during the execution of an operation, then the controller must complete the operation by operating in a degraded mode.

Traditional controller algorithms employ forward error-recovery, which consists of transitioning from an erroneous state[1] directly to completion. This method requires knowing about the context in which an error occurred and thus involves enumerating a large number of erroneous states. Courtright and Gibson propose a form of backward error-recovery method [Courtright94] to allow context-free recovery. Traditional backward error-recovery methods consist of undoing operations and returning the disk array to an error-free state. The disadvantage of these methods is that they are expensive. However, Courtright and Gibson's method is based on retry. When an error is encountered, the state of the system is modified to note which disk has failed, and the operation is retried based on the new state.

In this approach, operations are represented as Directed Acyclic Graphs (DAGs), which is an expansion on the representation used in TickerTAIP [Cao93], a distributed implementation of RAID Level 5. Each node in a DAG is an action to be performed on a disk or an action that computes data. DAGs provide a visualization of operations which simplifies reasoning about the ordering of actions.

Courtright and Gibson's method to error-recovery [Courtright94] has two requirements. First, actions must be idempotent. When a DAG fails, some actions may have been executed and some may have not. The controller then retries the operation with a similar DAG and actions that have already been performed will be performed again. So idempotency ensures that an action that is executed several times has the same effect as if it is executed only once. Secondly, the execution of each DAG must not result in the modification of data residing on disks that are not to be written. If a DAG changes the value of a disk and fails, then a following DAG will not be able to restore that value, because DAGs are selected

[1] An erroneous state is one in which a disk failure occurred in the middle of the execution of an operation.

independently from the context in which an error occurs. This requirement seems trivial to satisfy: DAGs must not directly write to disks that are not to be written. However, the matter gets more complicated in the presence of failures. When a disk has failed, its value is inferred by the disk array. This inferred value can be changed by a DAG that does not update the redundant data correctly. In this case, the second requirement implies that the inferred value of a disk not to be written, must not be changed with the execution of each DAG. A DAG satisfying the second requirement is said to *preserve consistency*.

Although many RAID controller algorithms perform actions that are idempotent, there exists some algorithms for which this is not the case [Courtright96]. Also for some architectures, it is impossible to build DAGs that preserves consistency for all operations. In order to provide a context-free error recovery method that would be general enough for all algorithms, Gibson et al. have devised roll-away recovery, which is a hybrid between Courtright and Gibson's method described above and more traditional backward error-recovery approaches [Courtright96].

In this thesis, we are concerned with controller algorithms that use Courtright and Gibson's error-recovery method [Courtright94] described previously[2]. Although these algorithms employ context-free[3] error recovery, the logics used to select a new DAG are nonetheless complicated. Also these algorithms allow fine interleavings between actions on the disks. As a consequence, these algorithms are difficult to test and to reason about. This is why formal methods are useful for proving the correctness of these algorithms.

The topic of this thesis is to prove the correctness of a controller algorithm for the RAID Level 5 System [Gibson95], that uses Courtright and Gibson's error recovery method, with the objective of formalizing the general notion of consistency.

We describe the algorithm and its specification using the I/O Automaton model [Lynch89]. This model is suitable for specifying components of asynchronous concurrent systems. Although the algorithm we consider is essentially sequential, the concurrency due to disk failures and actions on the disks, make the I/O Automaton model a suitable model to use. We prove that the algorithm is correct by showing that it implements its specification, using

[2]This method is different from the roll-away error recovery method described in the previous paragraph.
[3]Context-free means independent from the context in which an error occurs.

the proof by simulation technique [Lamport83, Lynch87, Lynch95, Lynch96].

In the course of this proof, we formalize the general notion of consistency, in a way that it can be applied to DAGs of other similar RAID controller algorithms. We used our consistency property to find an error in a DAG for the RAID Level 6 architecture, which appears in [Gibson95].

The outline of the thesis is the following. Chapter 2 presents the RAID Level 5 algorithm and its specification. Chapter 3 presents the proof of correctness. We show extensions of the algorithm in Chapter 4. Finally, Chapter 5 presents a summary of our conclusions.

Chapter 2

Algorithm

2.1 Informal Description

We now describe a controller algorithm for the RAID level 5 system [Gibson95], that uses Courtright and Gibson's error recovery method [Courtright96]. When the controller receives an operation, it chooses a local algorithm based on the state of the disk array and starts executing it. Local algorithms are represented as antecedence graphs. Each node in a graph is a *Read* or a *Write* action to a particular disk or an *Xor* action. If action A precedes action B in a graph, then the controller performs A before B. When a failure occurs in the middle of the execution of a graph, the graph stops executing and the controller then changes the state of the disk array and chooses a new graph to complete the initial operation. Figure 2-1 is a high-level representation of the algorithm.

Antecedence Graphs When the controller first receives an operation from the user, it determines which disks are directly involved in the operation. These are the disks that contain the data to be read, or the disks that must be written. We call the set of such disks *UsedDisks*. Note that the parity disk is not included in the *UsedDisks* set. The antecedence graphs are represented in Figure 2-2. We assume that the RAID system contains a single file that can be read and written by the user. The unit of data storage is a block. We also assume that the RAID system is composed of $n+1$ disks, indexed from 0 to n where the nth disk is the parity disk and each disk holds one block of data. The latter assumptions seems to be an over-simplification at first sight, since the RAID Level 5 has distributed parity that cannot be modeled with only one block per disk. However, we prove the correctness of a sub-controller that deals with only one parity group. Any number of these sub-controllers can be composed together and run concurrently. Since sub-controllers do not share any data, proving that their composition is correct does not require any special consideration.

Blocks are numbered from 0 to $n-1$. The initial value stored at disk D_i is $Block0_i$. We also have the condition that:

$$Block0_0 \oplus \ldots \oplus Block0_n = 0.$$

In the figures the notation "RD_i" means read from disk i and other node labels have similar meanings. The notation "ud" refers to a disk that belongs to *UsedDisks* and "nud" refers to one that does not.

- **Fault Free Read** The Fault-Free Read graph is used when there is no failure among the disks in *UsedDisks*. It consists of reading each disk directly.

- **Degraded Read** The Degraded Read is used when one of the disks to be read has failed. It consists of reading the entire array and reconstructing the missing data using the parity block.

- **Small Write** The Small Write operation is used in the absence of failures, when less than half of the array is to be written. In the presence of a failure in a disk that is not in *UsedDisks*, the Small Write is also used regardless of the number of disks to be written. It consists of reading the old data on the disks to be written and the parity, computing the new parity and writing the new parity and the new data.

- **Large Write** The Large Write is used when all the disks are to be written, in the absence of failures. In this case, the controller computes the new parity directly and writes to all the disks.

- **Reconstruct Write** The Reconstruct Write graph is used in the absence of failures, when more than half of the array is to be written. In the presence of a failure of a

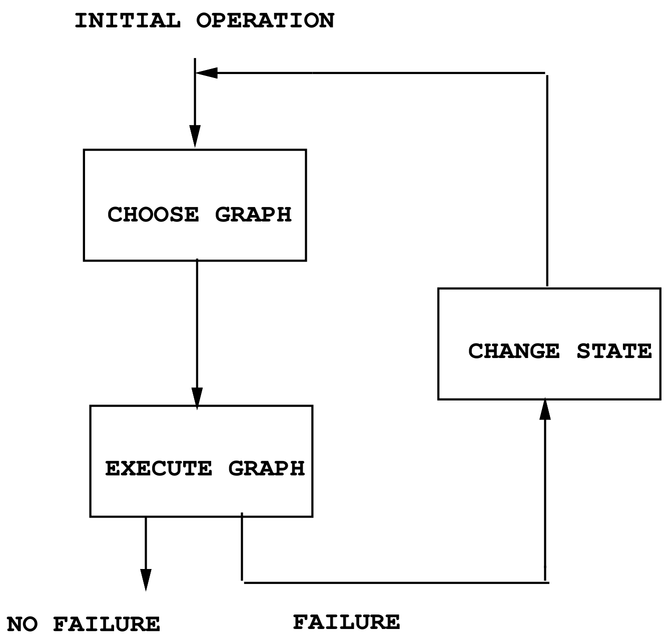

Figure 2-1: Algorithm

disk that is in *UsedDisks*, the Reconstruct Write is used regardless of the number of disks being written. It consists of reading the data from the disks that are not to be written, computing the parity from the data read and the data to be written and writing the new parity and data.

- **Parity Failed** The Parity Failed graph is used when the parity disk has failed. It consists of writing the disks directly without updating the parity disk.

2.2 Specification and Algorithm Automata

In this section, we describe the algorithm and its specification formally using I/O automata.

2.2.1 Conventions

In the following sections, we follow the conventions shown below:

- b, b_1 and b_2 are block numbers that are in $\{ 0, \ldots , n-1 \}$.

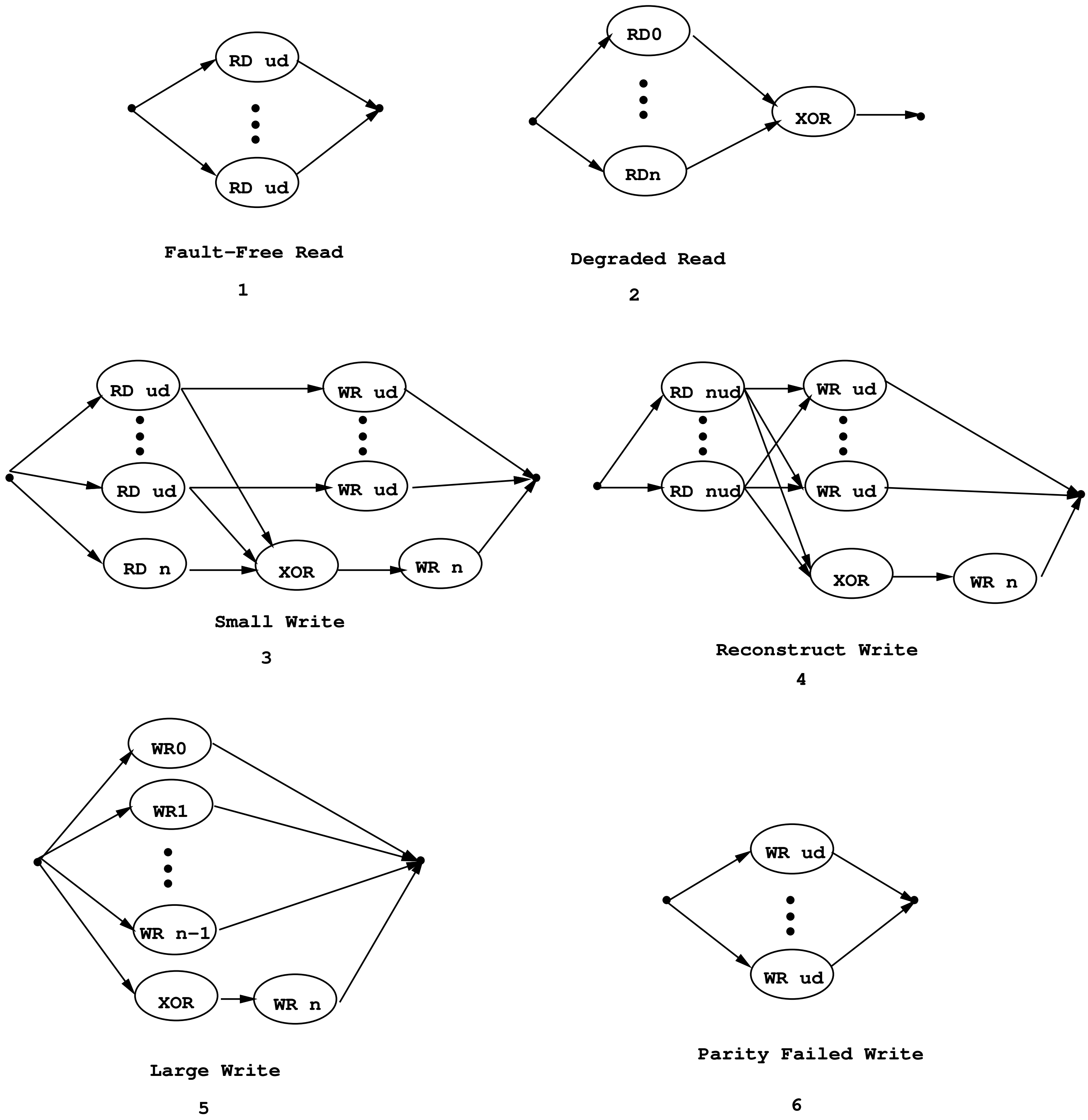

Figure 2-2: Antecedence Graphs

- $Value$ is an array of data blocks of variable length, indexed starting at 0.
- V is an array of data blocks of length $n+1$, indexed from 0 to n.
- v is one block of data.
- D is an array of n booleans, indexed from 0 to $n-1$.
- g is a graph index. It is an integer from 1 through 6. The corresponding graphs are shown in Figure 2-2.
- i is a disk index in $\{ 0, \dots , n \}$.
- F is a set of disk indeces.

We also use the following functions:

- $Size(Value)$, where $Value$ is an array of data blocks, gives the length of $Value$ in blocks.
- $Concat(UD, Data)$, where UD is an array of booleans of size n, indexed from 0 to $n-1$, and $Data$ is an array of data blocks of size $n+1$, indexed from 0 to n, returns an array of data blocks of variable size that is the concatenation of all the blocks in $Data$ with index i, such that $UD[i]$ is $True$.
- $Number(UD)$, where UD is the same as above, returns the number of indeces i for which $UD[i]$ is $True$.

2.2.2 Specification

Figure 2-3 shows the overall system architecture. The user interacts with the RAID system with the following set of actions:

- *Read(* b_1, b_2*)* : Read portion of file between block number b_1 and b_2, both endpoints included.

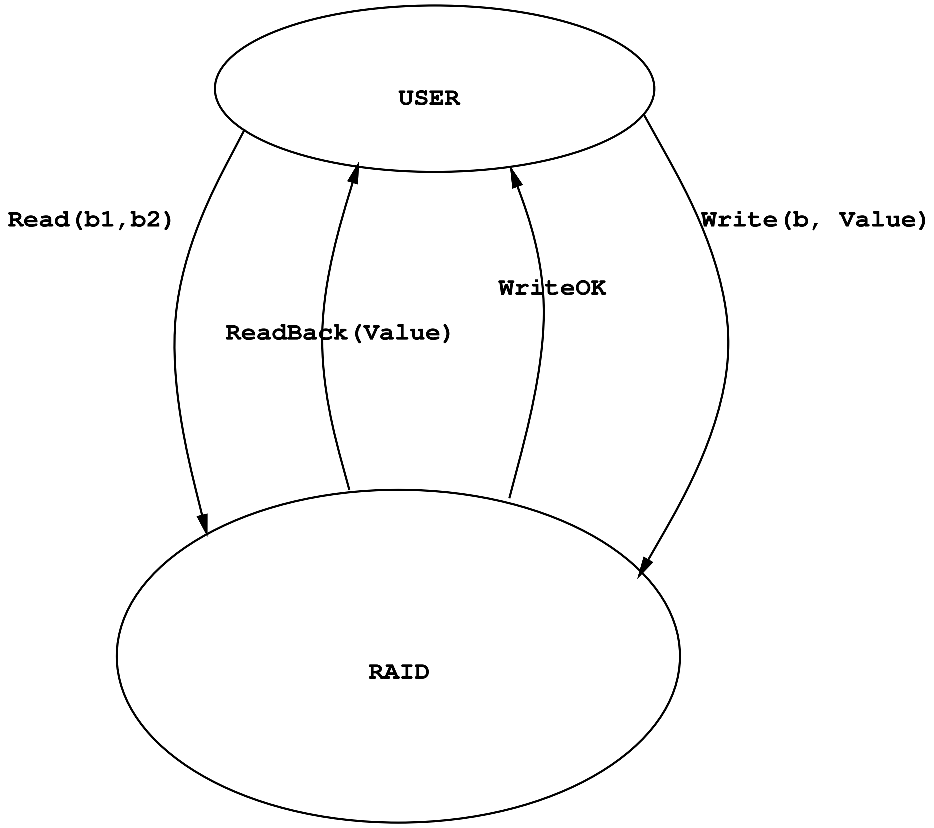

Figure 2-3: Overall System Architecture

- *Write(b, Value)* : Write portion of file starting at block number b with the data contained in $Value$, where $Value$ is an array of data blocks with a length l such that $l \leq n - b$.

The RAID system responds to the user's actions with the following:

- *ReadBack(Value)* : Data contained in the array $Value$ has been read from the disk array, where the length l of $Value$ is such that $l \leq n$.

- *WriteOK* : The disk array has been written successfully.

We describe the specification of the algorithm as an I/O automaton (Figure 2-4).

The specification automaton, which we call *Spec*, has the following state variables. The variable *Register* is an array of n data blocks. The variable *Data* is used to store data to be written to the *Register* or to be read from it and *UD* is used to indicate which disks are to

Spec

Signature

Inputs:
Read(b_1, b_2)
Write(b, Value) The length l of $Value$ is such that $l \leq n - b$.
Fail(i)
Internals:
Read
Write
Outputs:
ReadBack(Value) $Value$ is an array of data blocks of size l such that $0 \leq l \leq n - 1$.
WriteOK

State

Register	Array of n data blocks, indexed from 0 to $n - 1$, initially arbitrary values.
pc	Ranges over { *idle, read, write, readAck, writeAck* }, initially *idle*.
UD	Array of n booleans, indexed from 0 to $n - 1$, initially all *False*.
Data	Array of n data blocks, indexed from 0 to $n - 1$, initially arbitrary values.

Transitions

input: *Read(b_1, b_2)*
Eff: $pc := read$
For all i, $i \in \{b_1, ..., b_2\}$ do
$UD[i] := True$

internal: *Read*
Pre: $pc = read$
Eff: For all i, s.t. $(UD[i] = True)$ do
$Data[i] := Register[i]$
$pc := readAck$

output: *ReadBack(Value)*
Pre: $pc = readAck$
$Value = Concat(UD, Data)$
Eff: $pc := idle$
For all i, $i \in \{0, ...,$n-1$\}$ do
$UD[i] := False$

input: *Write(b, Value)*
Eff: $pc :=$ write
For all i, $i \in \{b, ..., b+$ Size$(Value)$ - 1$\}$ do
$UD[i] := True$
$Data[i] := Value[i - b]$

internal: *Write*
Pre: $pc = write$
Eff: For all i, $(UD[i] = True)$ do
$Register[i] := Data[i]$
$pc := writeAck$

output: *WriteOK*
Pre: $pc = writeAck$
Eff: $pc := idle$
For all i, $i \in \{0, ...,$n-1$\}$ do
$UD[i] := False$

Figure 2-4: I/O Automaton for the Specification be written or read. Finally, *pc* is used for the control of the automaton.

When *Spec* receives the input *Read(* b_1, b_2*)*, it reads the corresponding elements of *Register*. Similarly, when it receives the input *Write(b, Value)*, it writes to the elements of *Register*.

2.2.3 RAID

Figure 2-5 shows the organization of the RAID system. The controller, antecedence graphs, disks, and Failer module are modeled as separate I/O automata. The figure also shows the interfaces between the automata. The controller communicates with the graphs with the following actions:

- $ReadExecute_g(D, F)$: Signals Read graph g to start executing; D is a *UsedDisks* set, F a *FailedDisk* set, that may be empty.
- $WriteExecute_g(D, V, F)$: Signals Write graph g to start executing. D and F have the same meaning as above, V contains the data to be written.

Graphs communicate with the controller using the following actions:

- $ReadDone_g(V)$: Read graph g has completed execution successfully and is returning data in array V.
- $WriteDone_g$: Write graph g has completed execution successfully,
- $FailedGraph_g$: Graph g has stopped executing because of a disk failure in the middle of execution.

Graphs also communicate with the disks using the following actions, the index g is the index of the graph:

- RD_i: Read data from disk i.
- $WR_i(v)$: Write data v to disk i.

Disks respond to graph using the actions:

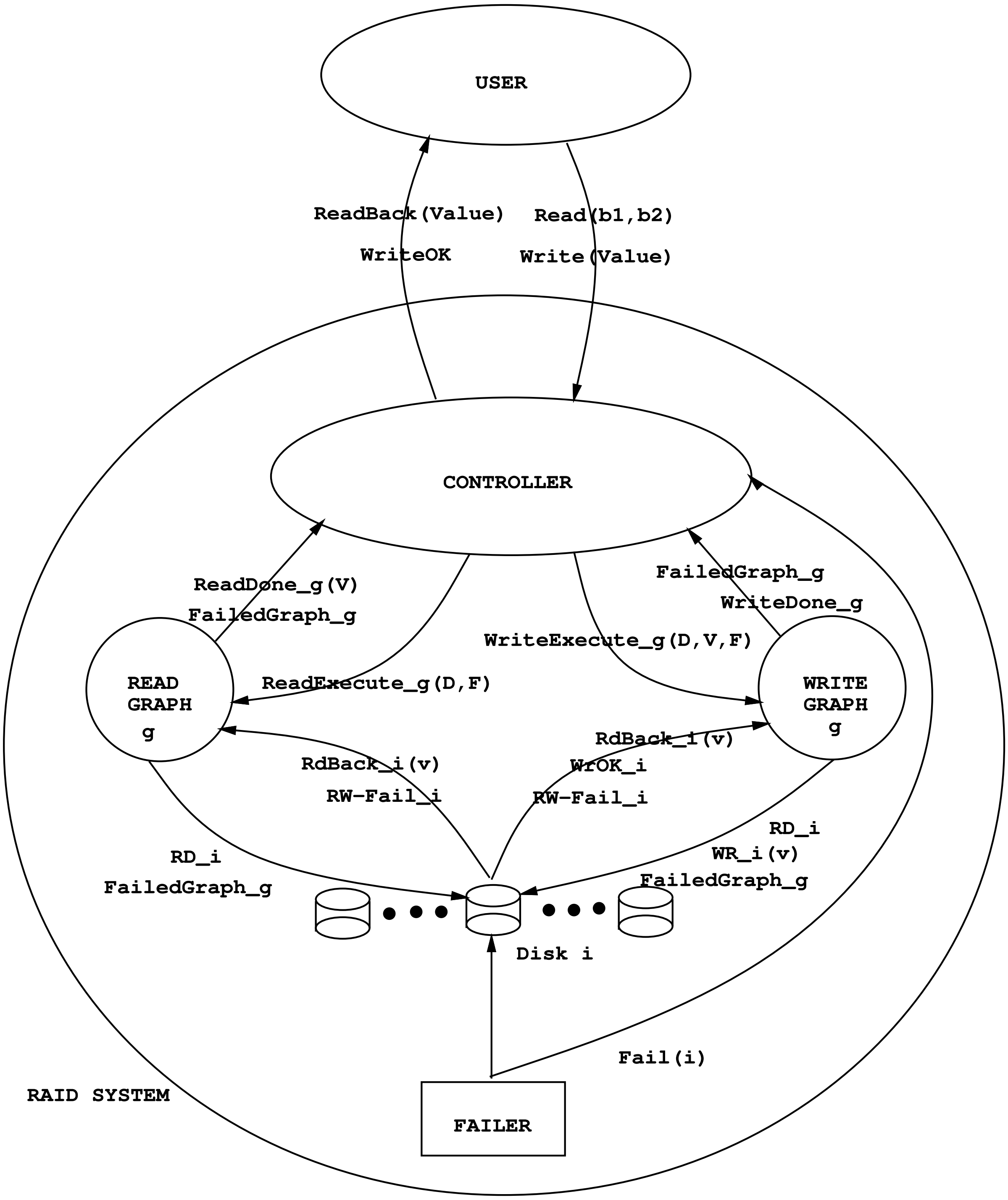

Figure 2-5: System Architecture

- $RdBack_i(v)$: Data v has been read from disk i.
- $WrOK_i$: Data has been written successfully to disk i.
- $RwFail_i$: Reading or writing disk i has failed.

Finally, the Failer module communicates with the disks and the controller with the following action.

- *Fail(i)*: Causes disk D_i to fail.

Informally, the system works as follows. Upon receiving an operation from the user, the controller chooses a graph based on the state of the disk array and sends an appropriate message to it. The graph then executes by sending messages to individual disks. When the graph finishes executing or fails because of a disk failure, it sends a message back to the controller. The controller then either acknowledges the user or chooses another graph to execute.

We now describe each I/O automaton in more detail.

Controller Automaton Figure 2-6 shows the I/O automaton for the controller. The controller automaton has the following state variables. The variable *UD* is an array of n booleans and indicates which disks are going to be read or written directly during the operation. The variable *Data* is an array of $n+1$ data blocks and is used as a temporary buffer. The variable *Graph* indicates the graph number that is currently executing. The variable *FailedDisk* is a set of disks that indicates which disks have currently failed, and *op* is used to indicate whether the system is currently performing a read or a write. The variable *rec* is a boolean that indicates whether the system is recovering from a failed graph or not, i.e. whether it is running a second graph to complete the operation. Finally *pc* is used for the control of the automaton.

When the controller receives an operation from the user, it first determines which disks are used in the operation. Then it chooses a graph to execute based on the state of the disk array and performs an appropriate output action to start its execution. If the graph finishes

Controller

Signature

Inputs:
Read(b_1, b_2), *Write(b, Value)*, *Fail(i)*, $ReadDone_g(V)$, $WriteDone_g$, $FailedGraph_g$

Internals:
$RChooseGraph$, $WChooseGraph$

Outputs:
ReadBack(Value), $WriteOK$, $ReadExecute_g(D, F)$, $WriteExecute_g(D, V, F)$

State

UD	Array of n booleans, indexed from 0 to $n-1$, initially all *False*.
Data	Array of $n+1$ data blocks, indexed from 0 to n, initially arbitrary.
Graph	Integer ranging from 1 to 6 included, initially arbitrary.
FailedDisk	Set of disk indeces, initially empty.
op	Ranging over { *read, write* }, initially arbitrary.
rec	Boolean, initially *False*.
pc	Ranging over { *ready, RchooseGraph, WchooseGraph, executeGraph, waiting, ackUser* }, initially *ready*.

Transitions

input: *Read(b_1, b_2)*
Eff: For all i in { b_1, ... , b_2 } do
$UD[i] := True$
$pc := RchooseGraph$
$op := read$

output: $ReadExecute_g(D, F)$
Pre: $pc = executeGraph$
$g = Graph$
$op = read$
$D = UD$
$F = FailedDisk$
Eff: $pc := waiting$

input: $ReadDone_g(V)$
Eff: For all i, s.t. ($UD[i] = True$) do
$Data[i] := V[i]$
$pc := ackUser$
$rec := False$

input: *Write(b, Value)*
Eff: For all i in
{ b, ... , $b + Size(Value) - 1$ } do
$UD[i] := True$
$Data[i] := Value[i-b]$
$pc := WchooseGraph$
$op := write$

output: $WriteExecute_g(D, V, F)$
Pre: $pc = executeGraph$
$g = Graph$
$op = write$
$D = UD$
$V = Data$
$F = FailedDisk$
Eff: $pc := waiting$

input $WriteDone_g$
Eff: $pc := ackUser$
$rec := False$

Figure 2-6: I/O Automaton for the Controller

output: *ReadBack(Value)*
Pre: $pc = ackUser$
$op = read$
$Value = Concat(UD, Data)$
Eff: $pc := ready$
For all i in $\{ 0, \ldots , n-1 \}$ do
$UD[i] := False$

input: *Fail(i)*
Eff: $FailedDisk := FailedDisk \cup \{i\}$

internal: *RChooseGraph*
Pre: $pc = RchooseGraph$
Eff: If $(FailedDisk = \{\}) \vee$
$((FailedDisk \neq \{\})$
$\wedge (n \notin FailedDisk)$
$\wedge (\forall i \in FailedDisk, UD[i] = False))$
Then $(Graph := 1)$
Else $(Graph := 2)$
$pc := executeGraph$

output *WriteOK*
Pre: $pc = ackUser$
$op = write$
Eff: $pc := ready$
For all i in $\{ 0, \ldots , n-1 \}$ do
$UD[i] := False$

input: $FailedGraph_g$
Eff: $rec := True$
If $(op = read)$
Then $pc := RchooseGraph$
Else $pc := WchooseGraph$

internal: *WChooseGraph*
Pre: $pc = WchooseGraph$
Eff: If $((FailedDisk = \{\})$
$\wedge (Number(UD) \leq n/2)) \vee$
$((FailedDisk \neq \{\})$
$\wedge (n \notin FailedDisk)$
$\wedge (\forall i \in FailedDisk, UD[i] = False))$
Then $(Graph := 3)$
ElseIf $((FailedDisk = \{\})$
$\wedge (n/2 < Number(UD) < n)) \vee$
$((FailedDisk \neq \{\})$
$\wedge (n \notin FailedDisk)$
$\wedge (\forall i \in FailedDisk, UD[i] = True))$
Then $(Graph := 4)$
ElseIf $(Number(UD) = n)$
Then $(Graph := 5)$
ElseIf $(FailedDisk = n)$
Then $(Graph := 6)$
$pc := executeGraph$

Figure 2-7: I/O Automaton for the Controller (Continued).

Graph1

Signature
Input:
$ReadExecute_1(D, F)$, $RdBack_i(v)$, $RwFail_i$
Output:
RD_i, $FailedGraph_1$, $ReadDone_1(V)$

State

UD	Array of n booleans, indexed from 0 to $n-1$, initially all *False*.
$ValueRead[n+1]$	Array of $n+1$ data blocks, indexed from 0 to n, initially arbitrary.
$RdDone[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$RdExecuted[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
executing	Boolean, initially *False*.
failureInExecution	Boolean, initially *False*.

Transitions

input: $ReadExecute_1(D, F)$
Eff: $UD := D$
$executing := True$

output: RD_i
Pre: $executing = True$
$RdExecuted[i] = False$
$UD[i] = True$
Eff: $RdExecuted[i] = True$

input: $RdBack_i(v)$
Eff: $ValueRead[i] := v$
$RdDone[i] := True$

input: $RwFail_i$
Eff: $failureInExecution := True$
$executing := False$

output: $ReadDone_1(V)$
Pre: $executing = True$
$\forall i.(UD[i] = True)$, $RdDone[i] = True$
$failureInExecution = False$
$V = ValueRead[]$
Eff: For all i s.t. $(UD[i] = True)$ do
$RdDone[i] := False$
$RdExecuted[i] := False$
For all i in $\{ 0, \ldots , n-1 \}$ do
$UD[i] := False$
$executing := False$

output: $FailedGraph_1$
Pre: $failureInExecution = True$
Eff: For all i s.t. $(UD[i] = True)$ do
$RdDone[i] := False$
$RdExecuted[i] := False$
For all i in $\{ 0, \ldots , n-1 \}$ do
$UD[i] := False$
$failureInExecution := False$

Figure 2-8: I/O Automaton for Graph 1 - Simple Read

successfully, it outputs $ReadDone(V)$ or $WriteDone$. Then the controller acknowledges the user with the appropriate output action. If the graph fails in its execution because of a disk failure, it outputs $FailedGraph$. This causes the controller to set *rec* to *True* and to choose another graph to complete the operation.

Graph Automata Figures 2-8 through 2-16 show the I/O automata for the antecedence graphs.

The graph automata have the following state variables:

Graph2

Signature

Inputs:
$ReadExecute_2(D, F)$, $RdBack_i(v)$
Internals:
XOR
Outputs:
RD_i, $ReadDone_2(V)$

State

UD	Array of n booleans, indexed from 0 to $n-1$, initially all *False*.
$ValueRead[n+1]$	Array of $n+1$ data blocks, indexed from 0 to n, initially arbitrary.
$RdExecuted[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$RdDone[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$XorDone$	Boolean, initially *False*.
$executing$	Boolean, initially *False*.
$failure$	in $\{ -1, \ldots, n \}$, initially arbitrary.

Transitions

input $ReadExecute_2(D, F)$
Eff: $UD := D$
$failure := F$
$executing := True$

output: RD_i
Pre: $executing = True$
$RdExecuted[i] = False$
$i \neq failure$
Eff: $RdExecuted[i] := True$

input: $RdBack_i(v)$
Eff: $ValueRead[i] := v$
$RdDone[i] := True$

internal: XOR
Pre: $executing = True$
$\forall i.((i \neq failure) \wedge (0 \leq i \leq n))$
$RdDone[i] = True$
$XorDone = False$
Eff: $XorDone := True$
$ValueRead[failure] :=$
$ValueRead[1] \oplus \ldots$
$\oplus ValueRead[failure-1]$
$\oplus ValueRead[failure+1]$
$\oplus \ldots \oplus ValueRead[n]$

output $ReadDone_2(V)$
Pre: $executing := True$
$XorDone = True$
$Value = ValueRead[]$
Eff: For all i s.t. $i \neq failure$ do
$RdDone[i] := False$
$RdExecuted[i] := False$
For all i in $\{ 0, \ldots, n-1 \}$ do
$UD[i] := False$
$XorDone := False$
$executing := False$

Figure 2-9: I/O Automaton for Graph 2 - Degraded Read.

Graph3

Signature

Inputs:
$WriteExecute_3(D, V, F)$, $RdBack_i(v)$, $WrOK_i$, $RwFail_i$,

Internals:
XOR

Outputs:
RD_i, $WR_i(v)$, $WriteDone_3$, $FailedGraph_3$

State

UD	Array of n booleans, indexed from 0 to $n-1$, initially all *False*.
$Data$	Array of $n+1$ data blocks, indexed from 0 to n, initially arbitrary.
$ValueRead[n+1]$	Array of $n+1$ data blocks, indexed from 0 to n, initially arbitrary.
$RdDone[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$RdExecuted[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$WrDone[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$WrExecuted[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$XorDone$	Boolean, initially *False*.
$executing$	Boolean, initially *False*.
$failureInExecution$	Boolean, initially *False*.

Transitions

input: $WriteExecute_3(D, V, F)$
Eff: $UD := D$
$Data := V$
$executing := True$

output: RD_i
Pre: $executing = True$
$RdExecuted[i] = False$
$UD[i] = True$
Eff: $RdExecuted[i] := True$

input: $RdBack_i(v)$
Eff: $ValueRead[i] := v$
$RdDone[i] := True$

input: $RwFail_i$
Eff: $failureInExecution := True$
$executing := False$

output: $WR_i(v)$
Pre: $executing = True$
If $(i \neq n)$
then $RdDone[i] = True$
$UD[i] = True$
else $XorDone = True$
$WrExecuted[i] = False$
$v = Data$[i]

Eff: $WrExecuted[i] = True$

input: $WrOK_i$
Eff: $WrDone[i] := True$

Figure 2-10: I/O Automaton for Graph 3 - Small Write

internal: XOR

Pre: $executing = True$

$XorDone = False$

$(\forall i_{((UD[i]=True)\vee(i=n))}$

$RdDone[i] = True$

Eff: $XorDone := True$

$Data[\mathrm{n}] :=$

$ComputeXOR(UD, Data, ValueRead[])$

output: $WriteDone_3$

Pre: $executing = True$

$failureInExecution = False$

$(\forall i.(UD[i] = True)) \wedge (i = n)$

$WrDone[i] = True$

Eff: $XorDone := False$

$executing := False$

For all i s.t. $(i = n)\vee$

$((i \neq n) \wedge (UD[i] = True))$

$RdDone[i] := False$

$WrDone[i] := False$

$RdExecuted[i] := False$

$WrExecuted[i] := False$

For all i in $\{ 0, \dots , n-1 \}$ do

$UD[i] := False$

$FailedGraph_3$

Pre: $failureInExecution = True$

Eff: $XorDone := False$

$failureInExecution := False$

For all i s.t. $(i = n)\vee$

$((i \neq n) \wedge (UD[i] = True))$

$RdDone[i] := False$

$WrDone[i] := False$

$RdExecuted[i] := False$

$WrExecuted[i] := False$

For all i in $\{ 0, \dots , n-1 \}$ do

$UD[i] := False$

Figure 2-11: I/O Automaton for Graph 3 - Small Write (Continued)

Graph4

Signature

Inputs:
$WriteExecute_4(D,V,F)$, $RdBack_i(v)$, $RwFail_i$, $WrOK_i$

Internals:
XOR

Outputs:
RD_i, $WR_i(v)$, $WriteDone_4$, $FailedGraph_4$

State

UD	Array of n booleans, indexed from 0 to $n-1$, initially all *False*.
$Data$	Array of $n+1$ data blocks, indexed from 0 to n, initially arbitrary.
$RdExecuted[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$RdDone[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$WrExecuted[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$WrDone[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$XorDone$	Boolean, initially *False*.
$executing$	Boolean, initially *False*.
$failure$	in $\{ -1, \ldots , n \}$, initially arbitrary.
$failureInExecution$	Boolean, initially *False*.

Transitions

input: $WriteExecute_4(D,V,F)$
Eff: $UD := D$
$Data := V$
$failure := F$
$executing := True$

output: RD_i
Pre: $executing = True$
$RdExecuted[i] = False$
$UD[i] = False$
Eff: $RdExecuted[i] := True$

input: $RdBack_i(v)$
Eff: $ValueRead[i] := v$
$RdDone[i] := True$

input: $RwFail_i$
Eff: $failureInExecution := True$
$executing := False$

output: $WR_i(v)$
Pre: $executing = True$
$(i \neq failure)$
$(i \neq n) \wedge (UD[i] = True)$
$\forall i.(UD[i] = False)$
$RdDone[i] = True$
$WrExecuted[i] = False$
$v = Data[i]$
Eff: $WrExecuted[i] := True$

input: $WrOK_i$
Eff: $WrDone[i] := True$

Figure 2-12: I/O Automaton for Graph 4 - Reconstruct Write.

internal: XOR

Pre: $executing = True$

$\forall i.(UD[i] = False)$

$RdDone[i] = True$

$XorDone = False$

Eff: $XorDone := True$

$Data[\mathrm{n}] :=$

$ComputeXOR_2(UD, Data, ValueRead[])$

output: $WR_n(v)$

Pre: $executing = True$

$XorDone = True$

$WrExecuted[n] = False$

$v = Data[\mathrm{n}]$

Eff: $WrExecuted[n] := True$

output $WriteDone_4$

Pre: $executing = True$

$\forall i.((UD[i] = True) \wedge (i = n))$

$WrDone[i] = True$

$failureInExecution = False$

Eff: For all i s.t. $(UD[i] = False)$ do

$RdDone[i] := False$

$RdExecuted[i] := False$

For all i s.t. $(i = n) \vee$

$(UD[\mathrm{i}] = True)$ do

$WrDone[i] := False$

$WrExecuted[i] := False$

For all i in $\{ 0, \ldots, n-1 \}$ do

$UD[i] := False$

$XorDone := False$

$executing := False$

output: $FailedGraph_4$

Pre: $failureInExecution = True$

Eff: For all i s.t. $(UD[i] = False)$ do

$RdDone[i] := False$

$RdExecuted[i] := False$

For all i s.t. $(i = n) \vee$

$(UD[\mathrm{i}] = True)$ do

$WrDone[i] := False$

$WrExecuted[i] := False$

For all i in $\{ 0, \ldots, n-1 \}$ do

$UD[i] := False$

$XorDone := False$

$failureInExecution := False$

Figure 2-13: I/O Automaton for Graph 4 - Reconstruct Write. (Continued)

Graph5

Signature

Inputs:
$WriteExecute_5(D,V,F)$, $WrOK_i$, $RwFail_i$,

Internals:
XOR

Outputs:
$WriteDone_5$, $FailedGraph_5$

State

UD	Array of n booleans, indexed from 0 to $n-1$, initially all *False*.
Data	Array of $n+1$ data blocks, indexed from 0 to n, initially arbitrary.
$WrExecuted[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$WrDone[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
XorDone	Boolean, initially *False*.
executing	Boolean, initially *False*.
failureInExecution	Boolean, initially *False*.

Transitions

input $WriteExecute_5(D,V,F)$
Eff: $UD := D$
$Data := V$
$executing := True$

output: $WR_i(v)$
Pre: $executing = True$
$WrExecuted[i] = False$
$v = Data[i]$
$UD[i] = True$
Eff: $WrExecuted[i] := True$

input $WrOK_i$
Eff: $WrDone[i] := True$

input $RwFail_i$
Eff: $failureInExecution := True$
$executing := False$

output: $WriteDone_5$
Pre: $executing = True$
$\forall i_{(0 \leq i \leq n)}$, $WrDone[i] = True$
$failureInExecution = False$
Eff: $XorDone := False$
$executing := False$
For all i in $\{$ 0, ... , n $\}$ do
$WrDone[i] := False$
$WrExecuted[i] := False$
For all i in $\{$ 0, ... , $n-1$ $\}$ do
$UD[i] := False$

output: $FailedGraph_5$
Pre: $failureInExecution = True$
Eff: $XorDone := False$
$failureInExecution := False$
For all i in $\{$ 0, ... , n $\}$ do
$WrDone[i] := False$
$WrExecuted[i] := False$
For all i in $\{$ 0, ... , $n-1$ $\}$ do
$UD[i] := False$

Figure 2-14: I/O Automaton for Graph 5 - Large Write

internal: XOR
Pre: $executing = True$
$XorDone = False$
Eff: $XorDone := True$
$Data[n] :=$
$Data[0] \oplus \ldots \oplus Data[n-1]$

output: $WR_n(v)$
Pre: $executing = True$
$XorDone = True$
$WrExecuted[n] = False$
$v = Data[n]$
Eff: $WrExecuted[n] := True$

Figure 2-15: I/O Automaton for Graph 5 - Large Write (Continued)

Graph6

Signature
Input:
$WriteExecute_6(D, V, F)$, $WrOK_i$
Output:
$WR_i(v)$, $WriteDone_6$

State

UD	Array of n booleans, indexed from 0 to $n-1$, initially all *False*.
$Data$	Array of $n+1$ data blocks, indexed from 0 to n, initially arbitrary.
$WrExecuted[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$WrDone[n+1]$	Array of $n+1$ booleans, indexed from 0 to n, initially all *False*.
$executing$	Boolean, initially *False*.
$failureInExecution$	Boolean, initially *False*.

Transitions

input: $WriteExecute_6(D, V, F)$
Eff: $UD := D$
$Data := V$
$executing := True$

output: $WR_i(v)$
Pre: $executing = True$
$UD[i] = True$
$WrExecuted[i] = False$
$v = Data[i]$

Eff: $WrExecuted[i] := True$

input: $WrOK_i$
Eff: $WrDone[i] := True$

output: $WriteDone_6$
Pre: $executing = True$
$\forall i.(UD[i] = True)$
$WrDone[i] = True$
Eff: $executing := False$
For all i s.t. $(UD[i] = True)$ do
$WrDone[i] := False$
$WrExecuted[i] := False$
For all i in $\{ 0, \ldots, n-1 \}$ do
$UD[i] := False$

Figure 2-16: I/O Automaton for Graph 6 - Parity Failed

- *UD* : Array of n booleans, indexed from 0 to $n-1$.
- *Data* : Array of $n+1$ data blocks, indexed from 0 to n.
- *ValueRead*$[n+1]$: Array of $n+1$ data blocks, indexed from 0 to n.
- *RdDone*$[n+1]$: Array of $n+1$ data blocks, indexed from 0 to n.
- *RdExecuted*$[n+1]$: Array of $n+1$ data blocks, indexed from 0 to n.
- *WrDone*$[n+1]$: Array of $n+1$ data blocks, indexed from 0 to n.
- *WrExecuted*$[n+1]$: Array of $n+1$ data blocks, indexed from 0 to n.
- *XorDone* : Boolean.
- *executing* : Boolean.
- *failure* : in $\{ -1, \ldots , n \}$
- *failureInExecution* : Boolean.

The variable *UD* is used to keep track of which disks are to be written or read directly. Variables *Data* and *ValueRead*[] are temporary buffers used to keep data to be written and read respectively. The variable *RdExecuted*$[i]$ (*WrExecuted*$[i]$) indicates whether a RD_i ($WR_i(v)$) action has executed. The variable *RdDone*$[i]$ (*WrDone*$[i]$) indicates whether a RD_i ($WR_i(v)$) action has finished executing. The variable *XorDone* indicates whether an XOR action has finished executing. The variable *executing* is *True* if and only if the graph is currently running and *failure* indicates a failure in the disk array before the graph started running. Finally *failureInExecution* is *True* when a failure occurred in the disk array while the graph was running.

There are two auxiliary functions used in the definitions for XOR actions. One of them is $ComputeXOR$(*UD*, *Data*, *ValueRead*[]). This function computes the exclusive Or of all the values *ValueRead*$[i]$ and *Data*[i] for which $UD[i] = True$ with *ValueRead*$[n]$. The other function is $ComputeXOR_2$(*UD*, *Data*, *ValueRead*[]). This function computes the exclusive or of all the values *ValueRead*$[i]$ for which $UD[i] = False$ with all the values *Data*[i] for which $UD[i] = True$.

$Disk_i$

Signature
Input:
RD_i, $WR_i(v)$, *Fail(i)*, $FailedGraph_g$
Output:
$RdBack_i(v)$, $WrOK_i$, $RwFail_i$

State

block	One block of Data, initially $Block0_i$.
failed	Boolean, initially *False*.
pc	Ranges over { *rd, wr, idle* }, initially *idle*.

Transitions

input: RD_i
Eff: $(pc := rd)$

output: $RdBack_i(v)$
Pre: $pc = rd$
$failed = False$
$v = block$
Eff: $pc := idle$

output: $RwFail_i$
Pre: $(pc = rd) \vee (pc = wr)$
$failed = True$
Eff: $pc := idle$

input: $WR_i(v)$
Eff: If $\neg failed$
Then $(block := v)$
$pc := wr$

output: $WrOK_i$
Pre: $pc = wr$
$failed = False$
Eff: $pc := idle$

input: *Fail(i)*
Eff: $failed := True$

input: $FailedGraph_g$
Eff: $pc := idle$

Figure 2-17: I/O Automata for a Disk

Disk Automata Figure 2-17 shows the I/O automaton for a disk.

A disk automaton has the following state variables:

- *block* : One block of data.
- *failed* : Boolean.
- *pc* : Ranges over { *rd, wr, idle* }.

The variable *block* is used to hold the data stored in the disk. The variable *failed* is true if the disk has failed. Finally *pc* is used for the control of the automaton. When the automaton recieves the input $FailedGraph_g$ it sets its *pc* to *idle*.

Failer Automata Figure 2-18 shows the I/O automaton for the Failer module.

Failer

Signature
Output:
Fail(i) : i is in { 0, ... , n }.

State

done Boolean, initially *False*.

Transitions

output: *Fail(i)*
Pre: $done = False$
Eff: $done = True$

Figure 2-18: I/O Automata the Failer module

The Failer has the following state variable:

- *done* : Boolean.

The variable *done* is set to *True* when the Failer produces a failure. This ensures that at most one failure occurs in any execution of the system.

Chapter 3

Proof of Correctness

In this chapter we prove that the composition consisting of the controller, graphs, disks automata (D_0 through D_n), and Failer implements *Spec*. We call this composition the RAID automaton.

We use proof by simulation. The outline of this section is the following. Section 3.1 presents some definitions. Then Section 3.2 describes some properties satisfied by RAID. Finally Section 3.3 presents the proof of correctness.

3.1 Definitions

Our first two definitions concern the Virtual Block value of a disk, which is the value implied by the system. We define a predicate that determines whether the Virtual Block value of a disk is defined in a state s of RAID.

Definition 3.1.1 *For all states s of RAID, $s.DefinedVB(i)$ is a boolean such that:* $s.DefinedVB(i) = \forall j_{(j \neq i)} \neg s.D_j.failed$.

We next define the Virtual Block (VB) value.

Definition 3.1.2 *For all states s of RAID, $s.VB(i)$ is one block of data such that:* $s.VB(i) = \bigoplus_{(j \in \{0,\ldots,i-1,i+1,\ldots,n\})} s.D_j.block$*, provided that* $s.DefinedVB(i) = True$*. Other-*

wise $s.VB(i)$ is undefined.

To carry out the proof, we add a history variable to RAID that records old values of the data blocks stored in the disks, at the time of completion of the previous operation. We augment each disk automaton D_i with a new state variable *hist* which is initially equal to $D_i.block$. We also add the following effect to the actions $ReadDone_g(V)$ and $FailedGraph_g$ for $g \in \{1, 2\}$, and $WriteDone_g$ for $g \in \{3, ..., 6\}$. For all disks D_i, such that $(0 \leq i \leq n)$, the actions do the following:

If $(\neg D_i.failed)$
Then $D_i.hist := D_i.block$
Else $D_i.hist := s.VB(i)$.

Note that if $D_i.failed$, then since the $Failer$ module produces at most one failure, $s.DefinedVB(i) = True$. So the second assignment above is valid.

We also add an effect to the action $FailedGraph_g$ for $g \in \{3, ..., 6\}$. For all disks D_i such that $\neg Controller.UD[i]$, the action has the additional effect above.

We add another history variable to each disk D_i, *init*, that records the values stored on the disks in the state when a Write graph starts executing. The difference between *hist* and *init* is that *hist* records old values of disks at the beginning of an operation, whereas *init* records old values at the beginning of the execution of a Write graph. For each disk D_i, $D_i.init$ has the initial value *undefined.* The action $WriteExecute_g(D, V, F)$ has additionally the following effect:

If $(\neg D_i.failed)$
Then $D_i.init := D_i.block$
Else $D_i.init := undefined.$

Our next two definitions concern the Virtual Init value of a disk. We define a predicate that determines whether $s.VI(i)$ is defined in a state s of RAID.

Definition 3.1.3 *For all states s of RAID, $s.DefinedVI(i)$ is a boolean such that:* $s.DefinedVI(i) = \forall j_{(j \neq i)}, s.D_j.init \neq undefined.$

We next define the Virtual Init (*VI*) value of a disk.

Definition 3.1.4 *For all states s of RAID, $s.VI(i)$ is one block of data such that:*
$s.VI(i) = \bigoplus_{(j \in \{0,\ldots,i-1,i+1,\ldots,n\})} s.D_j.init$*, provided that* $s.DefinedVI(i) = True$*. Otherwise* $s.VI(i)$ *is undefined.*

The following defines a predicate that determines whether a graph g is running in a state of RAID.

Definition 3.1.5 *Let s be a state of RAID. For all g, $s.running_g$ is a boolean variable such that:*
$s.running_g = s.Graph_g.executing \vee s.Graph_g.failureInExecution.$

Finally, we add a boolean variable *toBeWritten* to all disk automata, that determines whether a disk is to be written during the execution of a Write graph. For each disk D_i, such that $(0 \leq i < n)$, the variable $D_i.toBeWritten$ is initially *False*. Action $WriteExecute_g(D, V, F)$ has additionally the following effect:

For all i such that $Controller.UD[i]$,
$D_i.toBeWritten := True.$

Also action $WriteDone_g$ has the additional effect:

For all i, such that $(0 \leq i < n)$,
$D_i.toBeWritten := False.$

Note that when a Read graph is running, the variable *toBeWritten* is *False* for all disks. Also note that the above definition implies that a variable *toBeWritten* can only be set to *True* if a Write graph is running.

3.2 RAID Properties

In this section, we present some properties satisfied by RAID. These properties can be classified as follows:

- Basic Properties
- Parity and antecedence correctness for Write graphs,
- Read correctness for Read and Write graphs,
- Write correctness for Write graphs,
- Consistency,
- Properties Used in Proof of Correctness of RAID.

We describe each class of properties below.

Basic Properties

We will first present some basic properties of RAID systems. Lemmas 3.2.1 and 3.2.2 present properties of the *toBeWritten* variables. The Simple Consistency property (Lemma 3.2.3) expresses a simple relation between *block* and *hist* variables. Lemmas 3.2.4, 3.2.5, 3.2.6 and 3.2.7 concern the *init* variables. Finally Lemma 3.2.8 expresses antecedence relations that exist in $Graph_4$.

Parity and Antecedence Correctness for Write Graphs

The next class of properties deals with writing the parity correctly and the antecedences that must be in a graph to achieve that. Lemmas 3.2.9, 3.2.10 and 3.2.11 express parity and antecedence correctness.

Read Correctness for Read and Write Graphs

The third class of properties deals with reading the disk array correctly. Read graphs must read the disk array correctly so that they can return the right value back to the controller. Lemmas 3.2.12 and 3.2.13 express read correctness for Read graphs.

Write graphs need to read the disk array to compute the new parity. Lemma 3.2.14 expresses read correctness for Write graphs.

Write Correctness for Write Graphs

The third class of properties deals with writing the disk array correctly. These properties are only concerned with writing data disks and not the parity. Lemma 3.2.15 expresses write correctness in the absence of failures. Invariant 5 of Lemma 3.2.16 expresses write correctness concerning the VB value. Lemma 3.2.16 contains four other properties dealing with consistency. We have included the second write correctness invariant in this lemma to simplify the proofs. These invariants are proved by induction and simultaneously.

Consistency

The second requirement of Courtright and Gibson's error recovery method, is that each DAG must preserve consistency, meaning that the execution of each DAG must not change the values of disks that are not to be written. We call this property the General Consistency property. General Consistency is a property concerning the VB values of disks and their *block* and *hist* values, in the states at the end of execution of DAGs[1]. In order to prove the General Consistency property, we need some properties that are true in other states than just the end of execution of DAGs. For this purpose, we introduce the *consistency property* which has two components. These components are described below.

The first component of the consistency property is that the *block* value of every disk D_i that is not to be written and has not failed, must be equal to its *hist* value, in all states. This component essentially implies that DAGs must not write directly to disks that are not to be written.

The second component of the consistency property is concerned with the Virtual Block values of disks. The Virtual Block is the value inferred by the system for a disk. When the system is not currently running a graph, the Virtual Block value, if defined, of every disk D_i that is not to be written[2] must be consistent with the actual value of D_i, which is its *hist* value. The second component ensures that if D_i has failed, or will fail in some future state, then the system infers the right value for it. This is the core idea of the consistency property.

Note that if the Virtual Block is not defined for D_i then this means that a failure has occurred at a disk different from D_i and the failure of D_i will result in loss of data. This is not a problem since the system is designed to tolerate only one fault.

[1]The end of execution of a DAG is the state in which the action $ReadDone_g(V)$ or $WriteDone_g$ or $FailedGraph_g$ is enabled.

[2]Note that when the system is idle, then all disks are not to be written.

When the system is not currently running a graph, if D_i is a disk that has not failed, is not to be written, and for which the Virtual Block is defined, then the two components of consistency express an equality between the Virtual Block, the *block* value and the *hist* value.

The second component of the consistency property, namely the equality between the Virtual Block and *hist* values, is not an invariant[3], meaning that it does not hold in all reachable states of the system. If a Write graph is executing then the disk array can be partially updated. This means that if disk D_i is not to be written and has a defined Virtual Block value, its Virtual Block can be different from its *hist* value. If D_i fails at that point, the value inferred for it by the system is wrong, but that is not a problem since the execution of graphs cannot overlap and graphs are required to restore consistency at the end of their execution.

Lemma 3.2.17 expresses the General Consistency property. Lemma 3.2.3 expresses the first component of the consistency property and is used in the proof of Lemma 3.2.17. Lemma 3.2.16 includes four invariants that express the second component of the consistency property in different states of RAID. These are also used in the proof of Lemma 3.2.17.

Properties Used in Proof of Correctness of RAID

The final class of properties are those that are used directly in the proof of correctness of RAID. These lemmas do not add any new concept about the behavior of RAID, but generalize previous ones to make their application easier. These are Lemmas 3.2.17, 3.2.18 and 3.2.19.

There are no other auxiliary lemmas. However note that some very low-level properties have been omitted. These lemmas are proved easily by induction and have been omitted because they do not add to the understanding of the behavior of RAID. These include the property that graphs run one at a time, which is assumed throughout the following sections and is not mentioned. Other lemmas are mentioned each time they are used.

[3]The first component of the consistency property is an invariant.

3.2.1 Basic Properties

In this section we present the basic properties satisfied by RAID. These will be used in the proofs of lemmas presented in subsequent sections.

The first lemma indicates that if $UD[i]$ is *False* in a state s of RAID, then this implies that $s.D_i.toBeWritten$ is also *False*.

Lemma 3.2.1 *For all states s of RAID, for all i such that $(0 \leq i < n)$, $(\neg s.Controller.UD[i]) \Longrightarrow (\neg s.D_i.toBeWritten)$.*

Proof. We prove the invariant by induction. Let s_0 be an initial state of RAID. Initially, for all i such that $(0 \leq i < n)$, $s_0.Controller.UD[i] = False$ and $s_0.D_i.toBeWritten = False$. Therefore the invariant is satisfied in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant.

Case: $\pi =$ *Read(b_1, b_2)* or *Write(b, Value)*
Let i be such that $\neg s'.Controller.UD[i]$. Since no write graphs are running in s', we have $\neg s'.D_i.toBeWritten$. Therefore action π preserves the invariant.

Case: $\pi = WriteExecute_g(D, V, F)$, for $g \in \{3, ..., 6\}$
Action π has the effect of setting the variable $s.D_i.toBeWritten$ to *True* for all i such that $s.Controller.UD[i]$. Thus π does not change the variables $D_i.toBeWritten$ for i such that $\neg s.Controller.UD[i]$. Therefore π preserves the invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.1.
■

The next lemma expresses the fact that when a Write graph is running, then for all i, the $toBeWritten$ variable of disk D_i has the same value as $Controller.UD[i]$.

Lemma 3.2.2 *For all states s of RAID, if $s.running_g = True$ for $g \in \{3, ..., 6\}$, then $(s.Controller.UD[i]) \Longleftrightarrow (s.D_i.toBeWritten)$.*

Proof. We prove the invariant by induction. In any initial state of RAID, s_0, $s_0.running_g =$ *False*, for $g \in \{3, ..., 6\}$. Therefore the invariant is true vacuously in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant.

Case: $\pi = WriteExecute_g(D, V, F)$
For all i such that $s.Controller.UD[i]$, action π sets $s.D_i.toBeWritten$ to *True*. It also does not change the variables $s.D_i.toBeWritten$ for i such that $\neg s.Controller.UD[i]$. Therefore action π preserves the invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.2. ■

The following lemma is Simple Consistency and expresses the fact that if a disk has not failed and is not to be written, then its *block* value is equal to its *hist* value. This lemma expresses the first component of the consistency property.

Lemma 3.2.3 Simple Consistency
For all states s of RAID, for all i such that $\neg s.D_i.toBeWritten$:
If $(\neg s.D_i.failed)$ *then* $s.D_i.block = s.D_i.hist$.

Proof. We prove the lemma by induction. In any initial state s_0 of RAID, for all i, $s_0.D_i.failed = False$ and $s_0.D_i.block = s_0.D_i.hist$. Therefore the invariant is true in any initial state of RAID.

We now show that all actions of RAID preserve the invariant. Let (s, π, s') be a transition of RAID.

Case: $\pi = WR_i(v)$
Assume that i is such that $\neg s.D_i.failed$. This action assigns the value v to $s.D_i.block$ and does not change any other *block* or *hist* variables. The precondition of this action in all graph automata includes: $s.Controller.UD[i] = True$. Thus by Lemma 3.2.2, $s'.D_i.toBeWritten = True$, and the invariant is preserved trivially by action π.

Case: $\pi = ReadDone_g(V)$, or $WriteDone_g$

These actions have the effect of assigning the *block* value to the *hist* value of each disk D_i, such that $\neg s.D_i.failed$. Therefore these actions preserve the invariant.

Case: $\pi = FailedGraph_g$

If $g \in \{1, 2\}$, then this action has the effect of assigning the *block* value to the *hist* value of each disk D_i for all i, if $\neg s.D_i.failed$. Thus if $g \in \{1, 2\}$ action π preserves the invariant.

If $g \in \{3, ..., 6\}$, then this action has the effect of assigning the *block* value to the *hist* value of each disk D_i, for all i such that $\neg s.Controller.UD[i]$. Thus for all i such that $\neg s'.D_i.toBeWritten$, $s'.D_i.block = s'.D_i.hist$. Thus if $g \in \{3, ..., 6\}$, action π preserves the invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.3.

■

The next lemma expresses conditions under which the *init* variable is defined during the execution of Graphs 3 and 4.

Lemma 3.2.4 Conditions for the Definition of the *init* Variable

For all states s of RAID, if $s.running_g = True$ for $g \in \{3, 4\}$,

If i is such that $(\neg s.D_i.failed) \vee (s.Graph_g.RdDone[i] = True)$ then $(s.D_i.init \neq undefined)$.

Proof. We prove the invariant by induction. Let s_0 be any initial state of RAID. We have $s_0.running_g = False$ for $g \in \{3, 4\}$. Therefore the invariant is satisfied vacuously in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant.

Case: $\pi = WriteExecute_g(D, V, F)$ for $g \in \{3, 4\}$

First assume that i is such that $\neg s.D_i.failed$. In this case, action π assigns the value of $s.D_i.block$ to $s.D_i.Init$. Therefore $s.D_i.init \neq undefined$. Thus the invariant is preserved in this case.

Next assume that i is such that $s.D_i.failed$. A trivial proof by induction can be used to show that $s.Graph_g.RdDone[i] = False$. Therefore the invariant is preserved vacuously in this case.

Case: $\pi = Graph_g.RdBack_i(v)$, for $g \in \{3, 4\}$
This action has the effect of setting $s.Graph_g.RdDone[i]$ to *True*. The precondition of this action (in D_i) includes $\neg s.D_i.failed$. Therefore by the inductive hypothesis, ($s.D_i.init \neq$ *undefined*). Therefore action π preserves the invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.4. ■

The following lemma expresses the fact that if Graph 3 or 4 is running, then for all disks that are not to be written and have a defined *init* variable, their *init* variable is equal to their *hist* variable.

Lemma 3.2.5 Equality between *init* and *hist* Variables
For all states s of RAID, if $s.running_g = True$ for $g \in \{3, 4\}$,
for all i such that $(\neg s.Graph_g.UD[i]) \wedge (s.D_i.init \neq undefined)$,
$s.D_i.init = s.D_i.hist$.

Proof. We prove the invariant by induction. In any initial state s_0 of RAID, $s.running_g =$ *False* for $g \in \{3, 4\}$. Therefore the invariant is satisfied vacuously in any initial state.

Let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant.

Case: $\pi = WriteExecute_g(D, V, F)$ for $g \in \{3, 4\}$
Assume that i is such that $(\neg s.Graph_g.UD[i])$. Assume further that $\neg s.D_i.failed$. In this case, action π assigns the value of $s.D_i.block$ to $s.D_i.init$. We have $\neg s.D_i.toBeWritten$, by Lemma 3.2.1. This implies $s.D_i.block = s.D_i.hist$, by Lemma 3.2.3. Therefore $s'.D_i.init = s'.D_i.hist$. Therefore action π preserves the invariant in this case.

Assume now that $s.D_i.failed$. In this case, action π assigns the value *undefined* to $s.D_i.init$. Thus $s'.D_i.init =$ *undefined*. Therefore action π preserves the invariant in this case as well.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.5.

■

The next lemma is a property concerning the *init* variable for $Graph_3$. It expresses the fact that when a disk has not been read during the execution of $Graph_3$, then its *init* value is equal to its *block* value, provided that *init* is not *undefined*.

Lemma 3.2.6 Property Concerning *init* for Graph 3
For all states s of RAID, if $s.running_3 = True$,
then for all i such that $(0 \leq i \leq n)$,
if $(\neg s.Graph_3.RdDone[i]) \wedge (s.D_i.init \neq undefined)$, then $s.D_i.init = s.D_i.block$.

Proof. We prove the lemma by induction. Let s_0 an initial state of RAID. We have $s_0.running_3 = False$. Thus the invariant is satisfied vacuously in any initial state of RAID.

Let (s, π, s') a transition of RAID. We show that all actions of RAID preserve the invariant.

Case: $\pi = WriteExecute_3(D, V, F)$
We can show using a simple proof by induction that when this action is enabled in s, then for all i such that $(0 \leq i \leq n)$, $s.Graph_3.RdDone[i] = False$.
This action has the effect of assigning the value of $s.D_i.block$ to $s.D_i.init$, if $\neg s.D_i.failed$. Otherwise it assigns the value *undefined* to $s.D_i.init$. Therefore this action preserves the invariant.

Case: $\pi = Graph_3.WR_i(v)$
If $(i \neq n)$, then the precondition of this action includes:
$s.Graph_3.RdDone[i] = True$.
Therefore this action preserves the invariant vacuously if $(i \neq n)$.

If $(i = n)$, then the precondition of this action includes $s.Graph_3.XorDone = True$. We can show by a trivial proof by induction that this implies that $s.Graph_g.RdDone[n] = True$. Therefore this action preserves the invariant as well.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.6.
■

The following lemma is a similar property, but concerning $Graph_4$.

Lemma 3.2.7 Property Concerning *init* for Graph 4

For all states s of RAID, if $s.running_4 = True$,
then for all i *such that* $(\neg s.Controller.UD[i]) \vee (s.Graph_4.WrExecuted[i] = False)$,
if $(s.D_i.init \neq undefined)$ *then* $s.D_i.init = s.D_i.block$.

Proof. We prove the lemma by induction. In any initial state s_0 of RAID, $s_0.running_4 =$ *False*. Therefore the invariant is satisfied vacuously in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We show that all the actions of RAID preserve the invariant.

Case: $\pi = WriteExecute_g(D, V, F)$
This action has the effect of assigning the value of $s.D_i.block$ to $s.D_i.init$ for all i such that $(0 \leq i \leq n)$ and $\neg s.D_i.failed$. If $s.D_i.failed$ then this action assigns the value *undefined* to $s.D_i.init$. Therefore this action preserves the invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.7.
■

The next property expresses the fact that all RD_i actions in $Graph_4$ have precedence over all $WR_i(v)$ actions.

Lemma 3.2.8 Antecedence Relations for Graph 4

For all states s of RAID, if $s.running_4 = True$ *and there exists* i *such that*
$(\neg s.Graph_4.UD[i]) \wedge (\neg s.Graph_4.RdDone[i])$, *then*
$\forall j, s.Graph_4.WrExecuted[j] = False$.

Proof. We prove this invariant by induction. In any initial state s_0 of RAID, $s_0.running_4 = False$. Therefore the invariant is true in any initial state.

For the step condition, let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant.

The only action that sets $Graph_4.WrExecuted[i]$ to *True* is $Graph_4.WR_i(v)$. The precondition of this action includes:

$$\forall i_{(\neg s.Graph_4.UD[i])} s.Graph_4.RdDone[i] = True.$$

So this action is not enabled in any state in which there exists i such that $(\neg s.Graph_4.UD[i]) \wedge (\neg s.Graph_4.RdDone[i])$.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.8. ■

3.2.2 Parity and Antecedence Correctness

In this section we present parity and antecedence correctness for Write graphs 3, 4 and 5. Note that there is not a similar lemma for Write graph 6. This is because when this graph runs the parity has failed and therefore it is not updated. For the antecedences, these lemmas express the fact that all the relevant reads of the disk array must have been done if the parity has been written.

Lemma 3.2.9 Parity and Antecedence Correctness for Graph 3
For all states s of RAID, if $(s.running_3 = True)$,
$(s.Graph_3.WrExecuted[n] \wedge (\neg s.D_n.failed)) \Longrightarrow$

$$\begin{aligned} s.D_n.block &= (\bigoplus_{UD[j]} (s.Graph_3.ValueRead[j] \oplus s.Graph_3.Data[j])) \\ &\quad \oplus s.Graph_3.ValueRead[n] \end{aligned}$$

and,

$$\forall j_{(j=n) \vee (s.Graph_3.UD[j])},\ s.Graph_3.RdDone[j] = True.$$

Proof. We prove the invariant by induction. In any initial state of RAID, $running_3 = False$. Therefore the invariant is true vacuously in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant.

Case: $Graph_3.WR_n(v)$

The precondition of this action includes (in $Graph_3$): $XorDone = True$ and $v = Data[n]$.

A trivial proof by induction can show that (where the only case to consider is the action XOR) :

For all states s of RAID, if $s.Graph_3.XorDone = True$, then

$$s.Graph_3.Data[n] = ComputeXOR(s.Graph_3.UD, s.Graph_3.Data, s.Graph_3.ValueRead[]),$$

and

$$\forall j_{(j=n)\vee(s.Graph_3.UD[j])}, s.Graph_3.RdDone[j] = True.$$

So,

$$s.Graph_3.Data[n] = (\bigoplus_{UD[j]}(s.Graph_3.ValueRead[j] \oplus s.Graph_3.Data[j])) \oplus s.Graph_3.ValueRead[n].$$

The action $Graph_3.WR_n(v)$ sets the value of $D_n.block$ to v, if $\neg D_n.failed$. So we have,

$$s'.D_n.block = (\bigoplus_{UD[j]}(s'.Graph_3.ValueRead[j] \oplus s'.Graph_3.Data[j])) \oplus s'.Graph_3.ValueRead[n].$$

Also this action sets $Graph_3.WrExecuted[n]$ to $True$.

Since this action does not change the values of variables UD and $RdDone[j]$ we have :

$$\forall j_{(j=n)\vee(s'.Graph_3.UD[j])}, s'.Graph_3.RdDone[j] = True.$$

Therefore this action preserves the invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.9.

■

The following lemma expresses parity and antecedence correctness for $Graph_4$.

Lemma 3.2.10 Parity and Antecedence Correctness for Graph 4

For all states s of RAID, if $s.running_4 = True$,

$(s.Graph_4.WrExecuted[n] \wedge (\neg s.D_n.failed)) \Longrightarrow$

$$s.D_n.block = (\bigoplus_{UD[j]} s.Graph_4.Data[j]) \oplus (\bigoplus_{(\neg UD[j])} s.Graph_4.ValueRead[j]),$$

and,

$$\forall j_{(\neg s.Graph_4.UD[j])} s.Graph_4.RdDone[j] = True.$$

Proof. We prove the invariant by induction. In any initial state of RAID, $running_4 = False$. Therefore the invariant is true vacuously in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant.

Case: $Graph_4.WR_n(v)$
The precondition of this action includes (in $Graph_4$): $XorDone = True$ and $v = Data[n]$.

A trivial proof by induction can be used to show that (where the only case is the action XOR) :
For all states s of RAID, if $s.Graph_4.XorDone = True$ then

$$s.Graph_4.Data[n] = ComputeXOR_2(UD, Data, ValueRead[]),$$

and

$$\forall j_{(\neg s.Graph_4.UD[j])} s.Graph_4.RdDone[j] = True.$$

So,

$$s.Graph_4.Data[n] = (\bigoplus_{UD[j]} s.Graph_4.Data[j]) \oplus (\bigoplus_{(\neg UD[j])} s.Graph_4.ValueRead[j]).$$

The action $Graph_4.WR_n(v)$ sets the value of $D_n.block$ to v, if $\neg D_n.failed$. So we have,

$$s.D_n.block = (\bigoplus_{UD[j]} s'.Graph_4.Data[j]) \oplus (\bigoplus_{(\neg UD[j])} s'.Graph_4.ValueRead[j]).$$

Also this action sets $Graph_4.WrExecuted[n]$ to *True*.

Since this action does not change the values of variables UD and $RdDone[j]$, we have:

$$\forall j_{(\neg s'.Graph_4.UD[j])}, s'.Graph_4.RdDone[j] = True.$$

Therefore this action preserves the invariant.

All other actions preserve the action trivially. This completes the proof of Lemma 3.2.10. ■

The next lemma expresses parity and antecedence correctness for $Graph_5$.

Lemma 3.2.11 Parity Correctness for Graph 5

For all states s of RAID, if $s.running_5 = True$,

$(s.Graph_5.WrExecuted[n] \wedge (\neg s.D_n.failed)) \Longrightarrow$

$$s.D_n.block = \bigoplus_{UD[i]} s.Graph_5.Data[i].$$

Proof. We prove the invariant by induction. In any initial state of RAID, $running_5 = False$. Therefore the invariant is true vacuously in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant.

Case: $Graph_5.WR_n(v)$

The precondition of this action includes (in $Graph_5$): $XorDone = True$ and $v = Data[n]$.

A trivial proof by induction can be used to show that (where the only case is the action *XOR*):

For all states s of RAID, if $s.Graph_5.XorDone = True$,

then

$$s.Graph_5.Data[n] = \bigoplus_{UD[i]} s.Graph_5.Data[i].$$

Note that all disks D_i with index $i \in \{0, ...,\text{n-1}\}$ are such that $UD[i] = True$. The action $Graph_5.WR_n(v)$ sets the value of $D_n.block$ to v, if $\neg D_n.failed$. So we have,

$$s'.D_n.block = \bigoplus_{UD[i]} s'.Graph_5.Data[i].$$

Also this action sets $Graph_5.WrExecuted[n]$ to *True*. Therefore this action preserves the invariant.

All other actions preserve the action trivially. This completes the proof of Lemma 3.2.11. ■

3.2.3 Read Correctness for Read and Write Graphs

In this section, we present read correctness for Read and Write graphs. The following lemma is read correctness for Read graphs 1 and 2. It expresses the fact that while these graphs are running, if a disk has been read then the *hist* value for that disk is stored in the corresponding *ValueRead*[] variable.

Lemma 3.2.12 Read Correctness for Read Graphs
For all states s of RAID, if $s.running_g = True$, *for* $g \in \{1, 2\}$,
then $s.Graph_g.RdDone[i] \Longrightarrow s.D_i.hist = s.Graph_g.ValueRead[i]$,

Proof. We prove the Lemma by induction.
In any initial state of RAID, s_0, $Graph_g.executing = False$. Therefore $s_0.running_g = False$ and the Lemma is true vacuously in any initial state.

Let (s, π, s') be a transition of RAID. We show that all actions π preserve the invariant.

Case: $\pi = Graph_g.RdBack_i(v)$, for $g \in \{1, 2\}$
In this case $s'.running_g = True$ for $g \in \{1, 2\}$. The precondition of this action (in disk D_i) includes: $s.D_i.failed = False$ and $v = s.D_i.block$. We also have that $s'.D_i.failed = False$. This action sets $ValueRead[i]$ to v and $RdDone[i]$ to *True*.

This implies that:

$$s'.Graph_g.RdDone[i] \Longrightarrow s'.D_i.block = s'.Graph_g.ValueRead[i].$$

Since $s'.D_i.toBeWritten = False$, $s'.D_i.block = s'.D_i.hist$, by Lemma 3.2.3. Therefore,

$$s'.Graph_g.RdDone[i] \Longrightarrow s'.D_i.hist = s'.Graph_g.ValueRead[i].$$

Thus this action preserves the invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.12.
■

The following lemma is an additional read correctness condition for Graph 2. It expresses the fact that when a disk has failed and the XOR operation has been performed during the execution of Graph 2, then the VB value of that disk is stored in the corresponding $ValueRead[]$ variable.

Lemma 3.2.13 Additional Read Correctness for Read Graph 2
For all states s of RAID, if ($s.running_2 = True$) and ($s.Graph_2.XorDone$) then for all i such that $(0 \leq i < n)$,
if $\neg s.D_i.failed$, then $s.D_i.block = s.Graph_2.ValueRead[i]$, and
if $s.D_i.failed$, then $s.VB(i) = s.Graph_2.ValueRead[i]$.

Proof. We prove the Lemma by induction. In an initial state of RAID, s_0, $s_0.running_2 = False$. Therefore the Lemma is true vacuously in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We show that all actions π of RAID preserve the invariant. All actions preserve the invariant trivially except XOR.

Case: $\pi = XOR$
The precondition of this action (in $Graph_2$) includes:
$\forall i_{((i \neq failure) \wedge (0 \leq i \leq n))}, RdDone[i] = True$

By Lemma 3.2.12, we have

$$\forall i_{((i \neq failure) \wedge (0 \leq i \leq n))}, s.D_i.hist = s.Graph_2.ValueRead[i].$$

By Lemma 3.2.3, for all i such that $0 \leq i < n$, $s.D_i.block = s.D_i.hist$. Therefore:

$$\forall i_{((i \neq failure) \wedge (0 \leq i < n))}, s.D_i.block = s.Graph_2.ValueRead[i].$$

Note that the action does not change any of the above variables.

This action changes the value of $ValueRead[failure]$ as follows:

$$\begin{aligned}
s'.Graph_2.ValueRead[failure] &= \bigoplus_{(i \neq failure)} s.Graph_2.ValueRead[i] \\
&= \bigoplus_{(i \neq failure)} s.D_i.block \\
&= \bigoplus_{(i \neq failure)} s'.D_i.block \\
&= s'.VB(failure)
\end{aligned}$$

The action also sets the value of $Graph_2.XorDone$ to *True*.

Therefore this action preserves the invariant. This completes the proof of Lemma 3.2.13. ■

The third lemma is the read correctness condition for Write graphs 3 and 4. It expresses the fact that when these graphs read a disk, they store the *init* value of that disk in their corresponding *ValueRead*[ ]variable.

Lemma 3.2.14 Read Correctness for Write Graphs
For all states s of RAID, if $s.running_g = True$ for $g \in \{3, 4\}$,
then for all i such that $(0 \leq i \leq n)$,
$s.Graph_g.RdDone[i] \Longrightarrow s.Graph_g.ValueRead[i] = s.D_i.init$.

Proof. We prove the invariant by induction. In any initial state s_0 of RAID, $s_0.running_g = False$ for $g \in \{3, 4\}$. Therefore the invariant is true vacuously in any state of RAID.

Let (s, π, s') be a transition of RAID. We show that all actions of RAID preserve the invariant.

Case: $\pi = Graph_g.RdBack_i(v)$, for $g \in \{3, 4\}$

This action sets $s.Graph_g.RdDone[i]$ to $True$. It also sets $s.Graph_g.ValueRead[i]$ to v, which is equal to $s.D_i.block$.

SubCase: $g = 3$

A trivial proof by induction can be used to show that in a state in which the action $RdBack_i(v)$occurs, $s.Graph_g.RdDone[i] = False$. Therefore by Lemma 3.2.6, $s.D_i.block = s.D_i.init$. Thus $s'.Graph_3.ValueRead[i] = s'.D_i.init$. Therefore this action preserves the invariant.

SubCase: $g = 4$

A proof by induction can be used to show that when the action $RdBack_i(v)$ is enabled and $s.running_4 = True$, i is such that $\neg s.Controller.UD[i]$. The precondition of action π includes $\neg s.D_i.failed$, which implies by Lemma 3.2.4 that $s.D_i.init \neq undefined$. So by Lemma 3.2.7, $s.D_i.init = s.D_i.block$. Thus $s'.Graph_4.ValueRead[i] = s'.D_i.init$. Therefore this action preserves the invariant.

All other actions preserves the invariant trivially. This completes the proof of Lemma 3.2.14.
■

3.2.4 Write Correctness for Write Graphs

In this section, we present write correctness for Write graphs. The following lemma expresses write correctness for Write graphs in the absence of failures.

Lemma 3.2.15 Write Correctness for Write Graphs - No Failures
For all states s of RAID, if $s.running_g = True$ for $g \in \{3, ..., 6\}$,
if i is such that $(s.Graph_g.UD[i]) \wedge (\neg s.D_i.failed)$,
then $s.Graph_g.WrExecuted[i] \Longrightarrow s.D_i.block = s.Graph_g.Data[i]$.

Proof. We prove the Lemma by induction. In any initial state s_0 of RAID, $s.running_g = False$. Therefore the Lemma is true vacuously in any initial state.

Let (s, π, s') be a transition of RAID. We argue that all actions π preserve the invariant.

Case: $\pi = WR_i(v)$

The precondition of this action in all graphs g in { 3, ... , 6 } includes:

$s.Graph_g.UD[i] = True$ and $v = s.Graph_g.Data[i]$.

In $Graph_g$, this action sets $Graph_g.WrExecuted[i]$ to $True$. In disk D_i, this action sets the variable $D_i.block$ to v, if $\neg s.D_i.failed$.

Therefore this action preserves the invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.15.

■

The second property for the write correctness of Write graphs is presented in the following section as part of the lemma concerning consistency.

3.2.5 Consistency Invariants

In this section, we present four invariants that express formally the second component of the consistency property, in different states of the system. Invariants 2, 3 and 4 are used in the proof of the General Consistency property (Lemma 3.2.17). Invariant 1 is used in the proof of Invariants 3 and 4.

The fifth invariant presented in this section is the second write correctness property. It has been included in this section so that it can be proved simultaneously with the others.

This section may be skipped at first reading, since none of these invariants is directly used in the proof of correctness of RAID.

We present all five invariants as different components of one lemma to simplify their proofs. We prove them by induction and simultaneously.

We now describe the first four invariants. These are concerned with the second component of consistency, which is an equality between the Virtual Block value and the *hist* value of a disk that is not to be written and for which the Virtual Block is defined. Each invariant covers a different set of states.

Invariant 1 expresses the second component of the consistency property for the initial states of the execution of Write graphs 3 and 4. The property is captured by using the Virtual Init variable. Invariant 2 expresses the property for all states in which no Write graph is running. Invariant 3 expresses the property for states in which the action $WriteDone_g$ is enabled for $g \in \{3, ..., 5\}$. Finally, Invariant 4 expresses the property for states in which the action $FailedGraph_g$ is enabled for $g \in \{3, ..., 5\}$.

Note that none of these invariants is concerned with Write graph 6. This is because, this graph runs only when the parity disk has failed. Therefore at the end of its execution, there are no disks for which the Virtual Block is defined and thus the second component of consistency is satisfied vacuously.

Lemma 3.2.16 Invariants

For all states s of RAID, the following holds:

1. If $s.running_g = True$*, for* $g \in \{3, 4\}$*, for all* i *such that* $\neg s.D_i.toBeWritten$*,*
if $s.DefinedVI(i)$ *then* $s.VI(i) = s.D_i.hist$*.*

2. If $s.running_g = False$ *for* $g \in \{3, ..., 6\}$*,*
for all i *such that* $\neg s.D_i.toBeWritten$
if $s.DefinedVB(i)$ *then* $s.VB(i) = s.D_i.hist$*.*

3. If $s.running_g = True$ *for* $g \in \{3, ..., 5\}$*,*
if i *is such that* $(\neg s.D_i.toBeWritten) \wedge (s.DefinedVB(i))$
$\wedge(\forall j_{((j=n)\vee(s.Graph_g.UD[j]))}, (s.Graph_g.WrExecuted[j])$
then $s.VB(i) = s.D_i.hist$*.*

4. If the action $FailedGraph_g$ *for* $g \in \{3, ..., 6\}$*, is enabled in* s*, then:*
if i *is such that* $(\neg s.D_i.toBeWritten) \wedge (s.DefinedVB(i))$ *then:*
$s.VB(i) = s.D_i.hist$*.*

If $s.running_g = True$ *for* $g \in \{3, ..., 5\}$*,*
if i *is such that* $(s.Graph_g.UD[i]) \wedge s.DefinedVB(i)$
then $\forall j_{(j=n)\vee(s.Graph_g.UD[j]\wedge(j\neq i))}, (s.Graph_g.WrExecuted[j] = True) \Longrightarrow$

$$s.VB(i) = s.Graph_g.Data[i].$$

Proof. We prove the invariant by induction. Let s be an initial state of RAID. We have that $s.running_g = False$ for all g. Therefore Invariants 1, 3 and 4 are satisfied vacuously in any initial state of RAID.

We have that for all i, $\neg s.D_i.failed$ and $\bigoplus_i s.D_i.block = 0$. Therefore for all i, it is true that $s.DefinedVB(i)$. Furthermore $s.VB(i) = s.D_i.block = s.D_i.hist$. Therefore Invariant 2 is satisfied as well in any initial state of RAID.

Let (s, π, s') be a transition of RAID. We prove that all transitions of RAID preserve the invariant.

Case : $\pi = WriteExecute_g(D, V, F)$, $g \in \{3, 4\}$

Invariants 2, 3, 4 and 5 are preserved trivially with this action. We show that Invariant 1 is preserved as well.

Assume that i is such that $\neg s'.D_i.toBeWritten$ and $s'.DefinedVI(i)$. For all $j \neq i$, action π has the effect of assigning the value of $s.D_j.block$ to $s.D_j.init$, if $\neg s.D_j.failed$. Since for all j, $j \neq i$, $s'.D_j.init \neq undefined$, then for all $j \neq i$, $\neg s.D_j.failed$. Therefore for all $j \neq i$, $\neg s'.D_j.failed$ and thus $s'.DefinedVB(i)$.

We have that $s.running_g = False$ for $g \in \{3,...,6\}$. Therefore by Invariant 2, $s.VB(i) = s.D_i.hist$.

Thus $s'.VB(i) = s'.D_i.hist$. And since $s'.VI(i) = s'.VB(i)$, $s'.VI(i) = s'.D_i.hist$.

Therefore action π preserves Invariant 1 and the entire invariant.

Case: $\pi = ReadDone_g(V)$, or $FailedGraph_g$ for $g \in \{1, 2\}$

Invariants 1, 3, 4 and 5 are preserved trivially by action π. We prove that π also preserves Invariant 2.

Let i be such that $\neg s.D_i.toBeWritten$ and $s.DefinedVB(i)$.

First assume that $\neg s.D_i.failed$. In this case, action π has the effect of assigning the value of $s.D_i.block$ to $s.D_i.hist$. By Lemma 3.2.3, $s.D_i.block = s.D_i.hist$. Therefore $s'.D_i.hist = s.D_i.hist$. Since the value of $s.VB(i)$ does not change with action π, and $s.VB(i) = s.D_i.hist$, by the inductive hypothesis, we have: $s'.VB(i) = s'.D_i.hist$.

Now assume that $s.D_i.failed$. In this case, action π does not assign any value to $s.D_i.hist$. Therefore $s'.VB(i) = s'.D_i.hist$.

Thus π preserves the invariant.

Case : $\pi = WriteDone_g$

Invariants 1, 3, 4 and 5 are preserved trivially by action π. We prove that π also preserves Invariant 2.

If $g = 6$, then $s'.D_n.failed$ and $s'.DefinedVB(i) = False$ for all i such that $i \neq n$. Therefore Invariant 2 is preserved vacuously in this case.

Now we consider the case where $g \in \{3, ..., 5\}$. Action $WriteDone_g$ has the effect of setting $s.D_i.toBeWritten$ to $False$, for all i such that $(0 \leq i < n)$.

First assume that i is such that $\neg s.Controller.UD[i]$, $s.DefinedVB(i)$ and $\neg s.D_i.failed$. This implies that $\neg s.D_i.toBeWritten$, by Lemma 3.2.1. The precondition of $WriteDone_g$ includes:

$$\forall i_{(i=n)\vee(UD[i])}, s.Graph_g.WrDone[i] = True,$$

which implies by a trivial proof by induction that:

$$\forall i_{(i=n)\vee(UD[i])}, s.Graph_g.WrExecuted[i] = True.$$

This implies, by Invariant 3, that $s.VB(i) = s.D_i.hist$.

The action $WriteDone_g$ has the effect of assigning the value of $s.D_i.block$ to $s.D_i.hist$. Since $\neg s.D_i.toBeWritten$, $s.D_i.block = s.D_i.hist$, by Lemma 3.2.3. Therefore, $s'.VB(i) = s'.D_i.hist$. Thus Invariant 2 is preserved in this case.

Now assume that i is such that $s.Controller.UD[i]$, $s.DefinedVB(i)$, and $\neg s.D_i.failed$. Action $WriteDone_g$ has the effect of writing the value of $s.D_i.Block$ to $s.D_i.Hist$. Again we have that:

$$\forall i_{(i=n)\vee(UD[i])}, s.Graph_g.WrExecuted[i] = True.$$

Thus by Invariant 5, $s.VB(i) = s.Graph_g.Data[i]$. By Lemma 3.2.15, we have $s.D_i.block = s.Graph_g.Data[i]$. Therefore $s.VB(i) = s.D_i.block$. So $s'.VB(i) = s'.D_i.hist$. Thus action π preserves Invariant 2 in this case as well.

Finally, assume that i is such that $s.DefinedVB(i)$, and $s.D_i.failed$. In this case, action π assigns the value of $s.VB(i)$ to $s.D_i.hist$. Therefore $s'.VB(i) = s'.D_i.hist$. Thus action π

preserves Invariant 2 in this case as well.

Case: $\pi = FailedGraph_g$ for $g \in \{3, ..., 6\}$
Action π preserves Invariants 1, 3, 4 and 5 trivially. We show that it also preserves Invariant 2.
If $g = 5$, then there does not exist an i such that $\neg s.Controller.UD[i]$. Therefore there does not exist an i such that $\neg s.D_i.toBeWritten$. Thus there does not exist an i such that $\neg s'.D_i.toBeWritten$. Therefore Invariant 2 is preserved vacously in this case.

If $g = 6$, then $s'.D_n.failed$ and $s'.DefinedVB(i) = False$ for all i such that $i \neq n$. Therefore Invariant 2 is preserved vacuously in this case as well.

If $g \in \{3, 4\}$, assume that i is such that $\neg s.D_i.toBeWritten$ and $s.DefinedVB(i)$.
Assume further that $\neg s.D_i.failed$. By invariant 4, $s.VB(i) = s.D_i.hist$. This action has the effect of assigning the value of $s.D_i.block$ to $s.D_i.hist$. By Lemma 3.2.3, $s.D_i.block = s.D_i.hist$. Therefore, since $s.VB(i) = s'.VB(i)$, we have $s'.VB(i) = s'.D_i.hist$. Thus Invariant 2 is preserved in this case.

Assume now that $s.D_i.failed$. In this case, action π assigns the value of $s.VB(i)$ to $s.D_i.hist$. Therefore $s'.VB(i) = s'.D_i.hist$. Thus action π preserves Invariant 2 in this case as well. Therefore action π preserves the entire invariant.

Case : $\pi = Graph_3.WR_j(v)$
Action π preserves Invariants 1, 2 and 4 trivially. We first show that it also preserves Invariant 3 and then show that it also preserves Invariant 5.
Assume that there exists an i such that $(\neg s.D_i.toBeWritten) \wedge (s.DefinedVB(i))$. This implies that for all j such that $j \neq i$, $\neg s.D_j.failed$.

Assume further that this action causes the following to be true:

$$\forall j_{((j=n) \vee (s.Graph_3.UD[j]))}, (s'.Graph_3.WrExecuted[j] = True).$$

Then by Lemma 3.2.15,

$$\forall j_{(s.Graph_3.UD[j])}, s'.D_j.block = s'.Graph_3.Data[j].$$

Also by Lemma 3.2.3,

$$\forall j_{(\neg s.Graph_3.UD[j])\wedge(j\neq i)}, s'.D_j.block = s'.D_j.hist.$$

By Lemma 3.2.9,

$$s'.D_n.block = (\bigoplus_{UD[j]}(s'.Graph_3.ValueRead[j] \oplus s'.Graph_3.Data[j])) \oplus s'.Graph_3.ValueRead[n]$$

and,

$$\forall j_{(j=n)\vee(s'.Graph_3.UD[j])}, s'.Graph_3.RdDone[j] = True$$

The above equation together with Lemma 3.2.14 imply that:

$$\forall j_{(j=n)\vee(s'.Graph_3.UD[j])}, s'.Graph_3.ValueRead[j] = s'.D_j.init.$$

We have:

$$\begin{aligned}
s'.VB(i) &= \bigoplus_{(j\neq i)} s'.D_j.block \\
&= \bigoplus_{UD[j]} s'.D_j.block \\
&\quad \oplus \bigoplus_{(j\neq i)\wedge(\neg UD[j])} s'.D_j.block \\
&\quad \oplus s'.D_n.block \\
\\
&= \bigoplus_{UD[j]} s'.Graph_3.Data[j] \\
&\quad \oplus \bigoplus_{(j\neq i)\wedge(\neg UD[j])} s'.D_j.hist \\
&\quad \bigoplus_{UD[j]} s'.Graph_3.ValueRead[j] \\
&\quad \oplus \bigoplus_{UD[j]} s'.Graph_3.Data[j] \\
&\quad \oplus s'.Graph_3.ValueRead[n] \\
\\
&= \bigoplus_{(j\neq i)\wedge(\neg UD[j])} s'.D_j.hist
\end{aligned}$$

$$\bigoplus_{UD[j]} s'.D_j.init$$
$$\oplus s'.D_n.init$$

$$= \bigoplus_{(j \neq i) \wedge (\neg UD[j])} s'.D_j.init$$
$$\bigoplus_{UD[j]} s'.D_j.init$$
$$\oplus s'.D_n.init$$

Since for all j such that $j \neq i$, $\neg s'.D_j.failed$, we have by Lemma 3.2.4, $s'.D_j.init \neq undefined$. Using this fact we derive the last equality above using Lemma 3.2.5.

Since $s.DefinedVI(i)$, we have $s'.VI(i) = s'.D_i.hist$, by Invariant 1. So $s'.VB(i) = s'.D_i.hist$. Therefore this action preserves Invariant 3. We now show that action π also preseves Invariant 5.

Assume that there exists an i such that $(s.Graph_3.UD[i]) \wedge s.DefinedVB(i)$. Assume further that this action causes the following to be true:

$$\forall j_{((j=n) \vee (s.Graph_3.UD[j] \wedge (j \neq i)))}, (s'.Graph_3.WrExecuted[j] = True).$$

Then by Lemma 3.2.15,

$$\forall j_{(UD[j] \wedge (j \neq i))}, s'.D_j.block = s'.Graph_3.Data[j].$$

Also by Lemma 3.2.3,

$$\forall j_{(\neg UD[j])}, s'.D_j.block = s'.D_j.hist.$$

By Lemma 3.2.9,

$$s'.D_n.block = (\bigoplus_{UD[j]} (s'.Graph_3.ValueRead[j] \oplus s'.Graph_3.Data[j])) \oplus s'.Graph_3.ValueRead[n]$$

and,

$$\forall j_{(j=n) \vee (s'.Graph_3.UD[j])}, s'.Graph_3.RdDone[j] = True$$

The above equation together with Lemma 3.2.14 imply that:

$$\forall j_{(j=n)\vee(s'.Graph_3.UD[j])}, s'.Graph_3.ValueRead[j] = s'.D_i.init.$$

We have:

$$\begin{aligned}
s'.VB(i) &= \bigoplus_{(j\neq i)} s'.D_j.block \\
&= \bigoplus_{(j\neq i)\wedge UD[j]} s'.D_j.block \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.block \\
&\quad \oplus s'.D_n.block \\
\\
&= \bigoplus_{(j\neq i)\wedge UD[j]} s'.Graph_3.Data[j] \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.hist \\
&\quad \oplus \bigoplus_{UD[j]} s'.Graph_3.ValueRead[j] \\
&\quad \oplus \bigoplus_{UD[j]} s'.Graph_3.Data[j] \\
&\quad \oplus s'.Graph_3.ValueRead[n] \\
\\
&= s'.Graph_3.Data[i] \\
&\quad \oplus \bigoplus_{UD[j]} s'.D_j.init \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.hist \\
&\quad \oplus s'.D_n.init \\
&= s'.Graph_3.Data[i] \\
&\quad \oplus \bigoplus_{UD[j]} s'.D_j.init \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.init \\
&\quad \oplus s'.D_n.init
\end{aligned}$$

Note for all $j \neq i$, $\neg s'.D_j.failed$. Therefore by Lemma 3.2.4, $s'.D_j.init \neq undefined$. Using this fact and Lemma 3.2.5, we derived the last equality above. Also $s'.Graph_3.RdDone[i] = True$. Therefore $s'.D_i.init \neq undefined$.

Let j be such that $\neg s'.Graph_3.UD[j]$. So by Lemma 3.2.1, we have $\neg s'.D_j.toBeWritten$. We have that $s'.DefinedVI(j) = True$. Thus by Invariant 1, $s'.VI(j) = s'.D_j.hist$. By Lemma 3.2.5, $s'.D_j.hist = s'.D_j.init$. So $s'.VI(j) = s'.D_j.init$.

Thus $(\bigoplus_{UD[j]} s'.D_j.init) \oplus (\bigoplus_{(\neg UD[j])} s'.D_j.init) \oplus s'.D_n.init = 0$. Therefore we have:

$$s'.VB(i) = s.Graph_3.Data[i].$$

Thus this action preserves Invariant 5 and the entire invariant.

Case: $\pi = Graph_4.WR_j(v)$

Action π preserves Invariants 1, 2 and 4 trivially. We first show that it also preserves Invariant 3, and then show that it also preserves Invariant 5.

Assume that there exists an i such that $(\neg s.D_i.toBeWritten) \wedge (s.DefinedVB(i))$. This implies that for all j such that $j \neq i$, $\neg s.D_j.failed$. Assume further that this action causes the following to be true:

$$\forall j_{((j=n)\vee(s.Graph_4.UD[j]))}, (s'.Graph_4.WrExecuted[j] = True).$$

Then by Lemma 3.2.15,

$$\forall j_{(s.Graph_4.UD[j])}, s'.D_j.block = s'.Graph_4.Data[j].$$

By Lemma 3.2.10,

$$s'.D_n.block = (\bigoplus_{UD[j]} s'.Graph_4.Data[j]) \oplus (\bigoplus_{(\neg UD[j])} s'.Graph_4.ValueRead[j]),$$

and,

$$\forall j_{(\neg UD[j])} s'.Graph_4.RdDone[j] = True.$$

The above equation together with Lemma 3.2.14 imply,

$$\forall j_{(\neg UD[j])} s'.Graph_4.ValueRead[j] = s'.D_j.init.$$

And by Lemma 3.2.3,

$$\forall j_{(\neg UD[j]) \wedge (j \neq i)} s'.D_j.hist = s'.D_j.block.$$

We use the above equations in the equalities below:

$$\begin{aligned}
s'.VB(i) &= \bigoplus_{(j \neq i)} s'.D_j.block \\
&= \bigoplus_{UD[j]} s'.D_j.block \\
&\quad \oplus \bigoplus_{(j \neq i) \wedge (\neg UD[j])} s'.D_j.block \\
&\quad \oplus s'.D_n.block \\
\\
&= \bigoplus_{UD[j]} s'.Graph_4.Data[i] \\
&\quad \oplus \bigoplus_{(j \neq i) \wedge (\neg UD[j])} s'.D_j.hist \\
&\quad \oplus \bigoplus_{UD[j]} s'.Graph_4.Data[j] \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.Graph_4.ValueRead[j] \\
\\
&= \bigoplus_{(j \neq i) \wedge (\neg UD[j])} s'.D_j.hist \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.init
\end{aligned}$$

Since for all j such that $\neg UD[j]$, $s'.Graph_4.RdDone[j] = True$, then $s'.D_j.init \neq undefined$, by Lemma 3.2.4. Therefore for all j such that $\neg UD[j]$, $s'.D_j.init = s'.D_j.hist$, by Lemma 3.2.5. Therefore $s'.VB(i) = s'.D_i.hist$. Thus this action preserves Invariant 3. We now show that it also preserves Invariant 5.

Assume that there exists an i such that $(s.Graph_4.UD[i]) \wedge s.DefinedVB(i)$. Assume further that this action causes the following to be true:

$$\forall j_{((j=n)\vee(s.Graph_4.UD[j]\wedge(j\neq i)))}, (s'.Graph_4.WrExecuted[j] = True).$$

Then by Lemma 3.2.15,

$$\forall j_{(s.Graph_4.UD[j]\wedge(j\neq i))}, s'.D_j.block = s'.Graph_4.Data[j].$$

By Lemma 3.2.10,

$$s'.D_n.block = (\bigoplus_{UD[j]} s'.Graph_4.Data[j]) \oplus (\bigoplus_{(\neg UD[j])} s'.Graph_4.ValueRead[j]),$$

and,

$$\forall j_{(\neg s'.Graph_4.UD[j])} s'.Graph_4.RdDone[j] = True.$$

The above equation together with Lemma 3.2.14 imply,

$$\forall j_{(\neg UD[j])} s'.Graph_4.ValueRead[j] = s'.D_j.init.$$

And by Lemma 3.2.3,

$$\forall j_{(\neg UD[j])} s'.D_j.hist = s'.D_j.block.$$

We use the above equations in the equalities below:

$$\begin{aligned}
s'.VB(i) &= \bigoplus_{(j\neq i)} s'.D_j.block \\
&= \bigoplus_{(j\neq i)\wedge UD[j]} s'.D_j.block \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.block \\
&\quad \oplus s'.D_n.block \\
\\
&= \bigoplus_{(j\neq i)\wedge UD[j]} s'.Graph_4.Data[j] \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.block \\
&\quad \bigoplus_{(UD[j])} s'.Graph_4.Data[j]
\end{aligned}$$

$$\oplus \bigoplus_{(\neg UD[j])} s'.Graph_4.ValueRead[j]$$

$$
\begin{aligned}
&= s'.Graph_4.Data[i] \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.hist \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.init \\
&= s'.Graph_4.Data[i] \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.init \\
&\quad \oplus \bigoplus_{(\neg UD[j])} s'.D_j.init \\
&= s'.Graph_4.Data[i]
\end{aligned}
$$

The second two last equality is derived using Lemmas 3.2.4 and 3.2.5. Therefore this action preserves Invariant 5 and the entire invariant.

Case: $\pi = Graph_5.WR_j(v)$

Action π preserves Invariants 1, 2, 3 and 4 trivially. We show that it also preserves Invariant 5.

Assume that there exists an i such that $(s.Graph_5.UD[i]) \wedge (s.DefinedVB(i))$. Assume further that this action causes the following to be true:

$$\forall j_{((j=n)\vee(s.Graph_5.UD[j]\wedge(j\neq i)))}, (s'.Graph_5.WrExecuted[j] = True).$$

Then by Lemma 3.2.15,

$$\forall j_{(s.Graph_5.UD[j]\wedge(j\neq i))}, s'.D_j.block = s'.Graph_5.Data[j].$$

By Lemma 3.2.11,

$$s'.D_n.block = \bigoplus_{UD[j]} s'.Graph_5.Data[j].$$

We have

$$s'.VB(i) \quad = \quad \bigoplus_{(j\neq i)} s'.D_j.block$$

$$
\begin{aligned}
&= \bigoplus_{(j\neq i)\wedge UD[j]} s'.D_j.block \\
&\quad \oplus s'.D_n.block \\
&= \bigoplus_{(j\neq i)\wedge UD[j]} s'.D_j.block \\
&\quad \oplus \bigoplus_{UD[j]} s'.Graph_5.Data[j] \\
&= \bigoplus_{(j\neq i)\wedge UD[j]} s'.Graph_5.Data[j] \\
&\quad \oplus \bigoplus_{UD[j]} s'.Graph_5.Data[j] \\
&= s'.Graph_5.Data[i].
\end{aligned}
$$

Therefore this action preserves Invariant 5 and the entire invariant.

Case: $\pi = Graph_3.RwFail_i$

Action π preserves Invariants 1, 2, 3 and 5 trivially. We show that it also preserves Invariant 4.

This action sets the variable $Graph_3.failureInExecution$ to $True$. Thus the action $FailedGraph_3$ is enabled in s'. It can be shown that when the action $Graph_3.RwFail_i$ occurs, i is such that ($UD[i]$ or $i = n$) and $s'.D_i.failed$. Therefore for all j such that $(\neg s'.Graph_3.UD[j])$ we have $(\neg s.DefinedVB(j))$. Thus the invariant holds vacuously in s' and π preserves Invariant 4.

Case: $\pi = Graph_4.RwFail_i$

Action π preserves Invariants 1, 2, 3 and 5 trivially. We show that it also preserves Invariant 4.

This action sets the variable $Graph_4.failureInExecution$ to $True$. Thus $FailedGraph_4$ is enabled in s'. If i is such that $s'.Graph_4.UD[i]$ or $i = n$, then for all j such that $\neg s'.Graph_4.UD[j]$ we have $\neg s.DefinedVB(j)$. Therefore Invariant 4 is preserved vacuously in this case.

If i is such that $\neg s'.Graph_4.UD[i]$, then it can be shown that $s'.Graph_4.RdDone[i] = False$. Also we have by Lemma 3.2.1, that $\neg s'.D_i.toBeWritten$. Thus by Lemma 3.2.8, $\forall j, s'.Graph_4.WrExecuted[j] = False$.

Since $s'.D_i.failed$, we have that $\forall j_{(j\neq i)}, \neg s'.D_j.failed$.

Thus by Lemma 3.2.4, $\forall j_{(j \neq i)}, s'.D_j.init \neq \mathit{undefined}$.

Therefore by Lemma 3.2.7, $\forall j_{(j \neq i)}, s'.D_j.init = s'.D_j.block$.

So $s'.VB(i) = s'.VI(i)$. Thus by Invariant 1, $s'.VB(i) = s'.D_i.hist$. Therefore action π preserves Invariant 4 and the entire invariant.

Case: $\pi = Graph_5.RwFail_i$

Action π preserves Invariants 1, 2, 3 and 5 trivially. We show that it also preserves Invariant 4.

We have $s.running_5 = True$. Thus, there does not exist an i such that $(\neg s'.Graph_5.UD[i])$. Therefore this action preserves Invariant 4 and the entire invariant.

Case: $\pi = Graph_6.RwFail_i$

Action π preserves Invariants 1, 2, 3 and 5 trivially. We show that it also preserves Invariant 4.

We have $s.running_6 = True$. Thus, there does not exist an i such that $(i \neq n)$ and $s'.DefinedVB(i)$. Therefore this action preserves Invariant 4 and the entire invariant.

All other actions preserve the invariant trivially. This completes the proof of Lemma 3.2.16.

■

3.2.6 Properties Used in the Proof of Correctness of RAID

In this section, we present the properties used directly in the proof of correctness of RAID. The first property is the General Consistency property. It expresses the fact that at the end of the successful or unsuccessful execution of a graph, the *block* value of a non-failed disk that is not to be written is equal to its *hist* value, and that the VB value of a disk not to be written is equal to its *hist* value, if the VB value is defined.

Lemma 3.2.17 General Consistency

For all states s of RAID, if the action $ReadDone_g(V)$, $WriteDone_g$ or $FailedGraph_g$ is enabled in s, then:

For all i such that $\neg s.D_i.toBeWritten$:

1. If $(\neg s.D_i.failed)$ then $s.D_i.block = s.D_i.hist$.

2. If $s.DefinedVB(i)$ then $s.VB(i) = s.D_i.hist$.

Proof. Let s a state of RAID. Assume that the action $ReadDone_g(V)$, $WriteDone_g$ or $FailedGraph_g$ is enabled in s. Then by Lemma 3.2.3,
for all i such that $\neg s.D_i.toBeWritten$:
if $(\neg s.D_i.failed)$ then $s.D_i.block = s.D_i.hist$.

If the action $ReadDone_g(V)$ is enabled in s, $s.running_g = True$, for $g \in \{1, 2\}$. Then $s.running_g = False$ for $g \in \{3, ..., 6\}$. And by Invariant 2 of Lemma 3.2.16,
for all i such that $\neg s.D_i.toBeWritten$
if $s.DefinedVB(i)$ then $s.VB(i) = s.D_i.hist$.

Now assume that the action $WriteDone_g$ is enabled in s, for $g \in \{3, 4\}$. Then $s.running_g = True$ for $g \in \{3, 4\}$. The precondition of both actions includes:

$$\forall i_{(i=n)\vee(s.Graph_g.UD[i])}, s.Graph_g.WrDone[i] = True.$$

A trivial proof by induction can be used to show that

$$\forall i_{(i=n)\vee(s.Graph_g.UD[i])}, s.Graph_g.WrExecuted[i] = True.$$

For all i such that $(\neg s.D_i.toBeWritten)$, we have that, by Invariant 3 of Lemma 3.2.16:
if $(s.DefinedVB(i))$ then $s.VB(i) = s.D_i.hist$.

Assume that the action $WriteDone_5$ is enabled in s. In this case for all i such that $(0 \leq i < n)$, $s.Graph_5.UD[i] = True$. Thus there does not exist an i such that $(\neg s.D_i.toBeWritten)$. Therefore the invariant is satisfied vacuously for this action.

Assume that the action $WriteDone_6$ is enabled in s. $Graph_6$ executes when $D_n.failed$. Therefore there does not exist an i such that $s.DefinedVB(i)$. Thus the invariant is satisfied trivially for this action.

Assume the action $FailedGraph_g$ is enabled in s, for $g \in \{1, 2\}$. By Invariant 2 of Lemma 3.2.16, for all i such that $(\neg s.D_i.toBeWritten)$,
if $(s.DefinedVB(i))$ then $s.VB(i) = s.D_i.hist$. Therefore this action preserves the invariant.

Finally assume that $FailedGraph_g$ is enabled in s, for $g \in \{3, ..., 6\}$. By Invariant 4 of Lemma 3.2.16, for all i such that $(\neg s.D_i.toBeWritten)$,
if $(s.DefinedVB(i))$ then $s.VB(i) = s.D_i.hist$.
Therefore this action preserves the invariant.

This completes the proof of Lemma 3.2.17. ■

The next lemma expresses the general read correctness condition for read graphs. It phrases Lemmas 3.2.12 and 3.2.13 in a way that makes these properties easy to use in the proof of correctness of RAID.

Lemma 3.2.18 General Read Correctness for Read Graphs
For all states s of RAID, if the action $ReadDone_g(V)$ for $g \in \{1, 2\}$ is enabled in s, then: $\forall i_{s.Controller.UD[i]}, s.D_i.hist = s.Graph_g.ValueRead[i]$.

Proof. Let s a state of RAID. Assume that the action $ReadDone_g(V)$ is enabled in s for $g \in \{1, 2\}$. Then $s.running_g = True$. The precondition of action $ReadDone_1(V)$ includes:

$$\forall i_{s.Graph_1.UD[i]}, s.Graph_1.RdDone[i] = True.$$

Then by Lemma 3.2.12:
$s.D_i.hist = s.Graph_1.ValueRead[i]$.

The precondition of $ReadDone_2(V)$ includes: $s.Graph_2.XorDone = True$. Assume i is such that $(s.Controller.UD[i]) \wedge (\neg s.D_i.failed)$. Then by Lemma 3.2.13,
$s.D_i.block = s.Graph_2.ValueRead[i]$. Since $\neg s.D_i.toBeWritten$, $s.D_i.block = s.D_i.hist$, by Lemma 3.2.3. Thus $s.D_i.hist = s.Graph_2.ValueRead[i]$.

Now assume i is such that $(s.Controller.UD[i]) \wedge (s.D_i.failed)$. Then by Lemma 3.2.13, $s.VB(i) = s.Graph_2.ValueRead[i]$. Since $\neg s.D_i.toBeWritten$ and $s.DefinedVB(i)$, Invariant 2 of Lemma 3.2.16 implies:
$s.VB(i) = s.D_i.hist$.
Thus $s.D_i.hist = s.Graph_2.ValueRead[i]$.

This completes the proof of Lemma 3.2.18. ■

Finally, the next lemma expresses the general write correctness condition for Write graphs. It phrases Lemma 3.2.15 and Invariant 5 of Lemma 3.2.16 in a way that makes these properties easy to use in the proof of correctness of RAID.

Lemma 3.2.19 General Write Correctness for Write Graphs
For all states s of RAID, if the action $WriteDone_g$ for $g \in \{3, ..., 6\}$ is enabled in s, then: For all i such that $s.Controller.UD[i]$,
if $\neg s.D_i.failed$, then $s.Graph_g.Data[i] = s.D_i.block$
else $s.Graph_g.Data[i] = s.VB(i)$.

Proof. Let s be a state of RAID. Assume that the action $WriteDone_g$ is enabled in s. Then $s.running_g = True$. Assume i is such that $(s.Controller.UD[i]) \wedge (\neg s.D_i.failed)$. The precondition of action $WriteDone_g$ includes:

$$\forall i_{(s.Controller.UD[i])}, s.Graph_g.WrDone[i] = True.$$

A trivial proof by induction can be used to show that this implies the following:

$$\forall i_{(s.Controller.UD[i])}, s.Graph_g.WrExecuted[i] = True.$$

Thus by Lemma 3.2.15,

$$\forall i_{(s.Controller.UD[i])}, s.D_i.block = s.Graph_g.Data[i].$$

Now assume that i is such that $(s.Controller.UD[i]) \wedge (s.D_i.failed)$. In s the action $WriteDone_6$ cannot be enabled because $Graph_6$ executes only when disk D_n has failed. Thus we consider only graphs 3 through 5 in this case.
The precondition of action $WriteDone_g$ for $g \in \{3, ..., 5\}$ includes:

$$\forall j_{(j=n) \vee (s.Graph_g.UD[j] \wedge (j \neq i))}, s.Graph_g.WrDone[j] = True,$$

which implies:

$$\forall j_{(j=n) \vee (s.Graph_g.UD[j] \wedge (j \neq i))}, s.Graph_g.WrExecuted[j] = True.$$

Therefore by Invariant 5 of Lemma 3.2.16,
$s.VB(i) = s.Graph_g.Data[i]$.

This completes the proof of Lemma 3.2.19. ■

3.3 Correctness Proof

We show that RAID implements *Spec* by proving that there exists an abstraction function from the states of RAID to the states of *Spec*.

Simulation Function Let s and u be reachable states of RAID and *Spec* respectively and let f be the following function.

$f(s,u) \Leftrightarrow$

$$\forall i_{(0 \leq i < n)}, s.D_i.hist = u.Register[i]$$
$$\wedge (s.Controller.UD = u.UD)$$
$$\wedge (s.Controller.Data = u.Data)$$

Theorem 3.3.1 *f is a simulation function.*

Proof. Let s_0 be a start state of RAID. The variable $s_0.UD$ is initialized to an array of *False*. All initial states of *Spec* u_0 are such that $u_0.UD$ is an array of *False*. Since the *Register* variable of *Spec* can range over all of its possible values (the same holds for the variable *Data* of *Spec*), there exists a start state u_0 of *Spec* such that:

$$\begin{aligned} \forall i_{(0 \leq i < n)}, s_0.D_i.hist &= u_0.Register[i], and \\ s_0.UD &= u_0.UD, and \\ s_0.Data &= u_0.Data \end{aligned}$$

Therefore $f(s_0, u_0) = True$.

Let s a state of RAID and let u a state of *Spec* such that $f(s,u) = True$. Let (s, π, s') be a transition of RAID. We consider cases based on the type of actions performed by RAID.

Case : $\pi = Read(\ b_1,\ b_2)$

Let the corresponding execution fragment of *Spec* be $Read(\ b_1,\ b_2)$. Let u' the state of *Spec* such that $(u, Read(\ b_1,\ b_2), u')$ is a transition of *Spec* . By the code it is immediate that $s'.Controller.UD = u'.UD$.

The two actions do not change the value of variables *Data* in both automata. Therefore $s'.Controller.Data = u'.Data$.

$f(s, u)$ implies that

$$\forall i_{(0 \le i < n)}, s.D_i.hist = u.Register[i].$$

Action π does not change the variables *hist* and *Register*. Therefore,

$$\forall i_{(0 \le i < n)}, s'.D_i.hist = u'.Register[i].$$

Thus $f(s', u') = True$.

Case : $\pi = Write(\ b,\ Value)$

Let the corresponding execution fragment of *Spec* be $Write(\ b,\ Value)$. Let u' the state such that $(u, Write(\ b,\ Value), u')$ is a transition of *Spec* . A similar argument as above can be used to show that $f(s', u') = True$.

Case : $\pi = ReadDone_g(V)$

Let the corresponding execution fragment be *Read*. Let u' the state such that $(u, Read, u')$ is a transition of *Spec*.

$ReadDone_g(V)$ does not change $s.Controller.UD$. So

$$s.Controller.UD = s'.Controller.UD.$$

Similarly,

$$u.UD = u'.UD.$$

Since $f(s, u) = True$, $s'.Controller.UD = u'.UD$.

We now show that $s'.Controller.Data = u'.Data$. $f(s,u) = True$ implies that:

$$\forall i_{(0 \leq i < n)}, s.D_i.hist = u.Register[i]. \tag{3.1}$$

Since action $ReadDone_g(V)$ is enabled in s, we have by Lemma 3.2.18:

$$\forall i_{s.Controller.UD[i]}, s.D_i.hist = s.Graph_g.ValueRead[i].$$

Since $\forall i_{0 \leq i \leq}, s.Graph_g.ValueRead[i] = V[i]$, we have (by the effect of $ReadDone_g(V)$ in the Controller automaton):

$$\forall i_{s.controller.UD[i]}, s.D_i.hist = V[i].$$

Therefore by Equation 3.1:

$$\forall i_{s.Controller.UD[i]}, V[i] = u.Register[i].$$

Thus by the code of actions $ReadDone_g(V)$ and $Read$ in $Spec$:

$$\forall i_{s'.Controller.UD[i]}, s'.Controller.Data[i] = u'.Data[i].$$

We also have that $\forall i_{\neg s.Controller.UD[i]}$, $s'.Controller.Data[i] = u'.Data[i]$, since these values do not change with the transitions $ReadDone_g(V)$ and $Read$.
Thus:

$$\forall i_{(0 \leq i < n)}, s'.Controller.Data[i] = u'.Data[i].$$

We now prove that $\forall i_{(0 \leq i < n)} s'.D_i.hist = u'.Register[i]$. If i is such that $\neg s.D_i.failed$, then the action $ReadDone_g(V)$ assigns the value of $s.D_i.block$ to $s.D_i.hist$. Since $s.running_g$ for $g \in \{1, 2\}$, we have that $\forall j, s.D_j.toBeWritten = False$. Thus by Lemma 3.2.17, $s.D_i.block = s.D_i.hist$.
Therefore by Equation 3.1, $s'.D_i.hist = u'.Register[i]$.

If i is such that $s.D_i.failed$, then $s.DefinedVB(i) = True$ and action $ReadDone_g(V)$ assigns the value of $s.VB(i)$ to $s.D_i.hist$. Also by Lemma 3.2.17 $s.VB(i) = s.D_i.hist$. Therefore by Equation 3.1, $s'.D_i.hist = u'.Register[i]$. Thus

$$\forall i_{(0 \leq i < n)} s'.D_i.hist = u'.Register[i].$$

Therefore $f(s', u') = True$.

Case : $\pi = WriteDone_g$

Let the corresponding execution of *Spec* be $Write$. Let u' the state such that $(u, Write, u')$ is a transition of *Spec*. $WriteDone_g$ does not change $s.Controller.UD$ or $s.Controller.Data$. So

$$s'.Controller.UD = s.Controller.UD$$

and

$$s'.Controller.Data = s.Controller.Data.$$

Similarly,

$$u'.UD = u.UD$$

and

$$u'.Data = u.Data.$$

Since $f(s, u) = True$,

$$s'.Controller.UD = u'.UD$$

and

$$s'.Controller.Data = u'.Data.$$

We now show that $\forall i_{(0 \leq i < n)}, s'.D_i.hist = u'.Register[i]$.

Assume i is such that $\neg s.Controller.UD[i]$. Then $\neg s.D_i.toBeWritten$. Assume further that $\neg s.D_i.failed$. Then by Lemma 3.2.17, $s.D_i.block = s.D_i.hist$. Since the action $WriteDone_g$ has the effect of assigning $s.D_i.block$ to $s.D_i.hist$, and $s.D_i.hist = u.Register[i]$ (by the inductive hypothesis):

$s'.D_i.hist = u'.Register[i]$.

Assume now that i is as above such that $\neg s.Controller.UD[i]$, but that $s.D_i.failed$. Again we have $\neg s.D_i.toBeWritten$. And since the $Failer$ module produces at most one failure,

we have $s.DefinedVB(i)$. Then by Lemma 3.2.17, $s.VB(i) = s.D_i.hist$. Since the action $WriteDone_g$ has the effect of assigning $s.VB(i)$ to $s.D_i.hist$, and $s.D_i.hist = u.Register[i]$ (by the inductive hypothesis), we have:
$s'.D_i.hist = u'.Register[i]$.

Next assume that i is such that $s.Controller.UD[i]$ and that $\neg s.D_i.failed$. Then by Lemma 3.2.19, $s.Graph_g.Data[i] = s.D_i.block$.
Thus $s'.D_i.block = u'.Register[i]$.
Since the action $WriteDone_g$ has the effect of assigning $s.D_i.block$ to $s.D_i.hist$, we have:
$s'.D_i.hist = u'.Register[i]$.

Finally assume that i is such that $s.Controller.UD[i]$, but that $s.D_i.failed$. Then by Lemma 3.2.19,
$s.Graph_g.Data[i] = s.VB(i)$.
Thus $s'.VB(i) = u'.Register[i]$.
Since the action $WriteDone_g$ has the effect of assigning $s.VB(i)$ to $s.D_i.hist$, we have:
$s'.D_i.hist = u'.Register[i]$.

Therefore $\forall i_{(0 \leq i < n)}, s'.D_i.hist = u'.Register[i]$. Thus $f(s', u') = True$.

Case : $\pi = FailedGraph_g$
Let the corresponding execution fragment of *Spec* be no action. $FailedGraph_g$ does not change $s.Controller.UD$ or $s.Controller.Data$. Thus $s'.Controller.UD = u.UD$ and $s'.Controller.Data = u.Data$.

We now show that $\forall i_{(0 \leq i < n)} s'.D_i.hist = u'.Register[i]$.
Assume that i is such that $s.Controller.UD[i]$ and $\neg s.D_i.failed$. If $g \in \{1, 2\}$, then this action has the effect of assigning $s.D_i.block$ to $s.D_i.hist$. By Lemma 3.2.17 $s.D_i.block = s.D_i.hist$ and $s.D_i.hist = u.Register[i]$. Therefore, $s'.D_i.hist = u.Register[i]$.
If $g \notin \{1, 2\}$, then action π has no effect on $s.D_i.hist$. Therefore $s'.D_i.hist = u.Register[i]$.

Next assume that i is such that $s.Controller.UD[i]$ but that $s.D_i.failed$. Then by Lemma 3.2.17, $s.VB(i) = s.D_i.hist$. If $g \in \{1, 2\}$, this action has the effect of assigning $s.VB(i)$ to $s.D_i.hist$, and since $s.D_i.hist = u.Register[i]$, we have :
$s'.D_i.hist = u.Register[i]$. Again, if $g \notin \{1, 2\}$, then action π has no effect on $s.D_i.hist$.

Next assume that i is such that $\neg s.Controller.UD[i]$ and $\neg s.D_i.failed$. Then by Lemma 3.2.17, $s.D_i.hist = s.D_i.block$. Since this action has the effect of assigning $s.D_i.block$ to $s.D_i.hist$ and $s.D_i.hist = u.Register[i]$, we have:
$s'.D_i.hist = u.Register[i]$.

Finally, assume that i is such that $\neg s.Controller.UD[i]$ but that $s.D_i.failed$. Then by Lemma 3.2.17, $s.VB(i) = s.D_i.hist$.
Since this action has the effect of assigning $s.VB(i)$ to $s.D_i.hist$, and $s.D_i.hist = u.Register[i]$, we have :
$s'.D_i.hist = u.Register[i]$.

Therefore $\forall i_{(0 \leq i < 0)}, s'.D_i.hist = u.Register[i]$. So $f(s', u) = True$.

Case : $\pi =$ *ReadBack(Value)*
Let the corresponding action of *Spec* be *ReadBack(Value)*. Let u' the state such that $(u,$ *ReadBack(Value)*$, u')$ is a transition of *Spec*. $f(s, u) = True$ implies that
$s.Controller.Data = u.Data$ and $s.Controller.UD = u.UD$.
So the two *ReadBack(Value)* actions output the same value. The two actions also leave the variable *Data* unchanged and change *UD* in the same way. So $f(s', u') = True$.

Case : $\pi = WriteOK$
Let the corresponding action be $WriteOK$. Let u' the state such that $(u, WriteOK, u')$ is a transition of *Spec*. Both actions leave the variable *Data* unchanged and change the variable *UD* in the same way. So $f(s', u') = True$.

All other actions preserve the simulation relation trivially. This completes the proof of Theorem 3.3.1. ■

Chapter 4

Extensions

In this chapter we consider some extensions to the algorithm studied in the previous sections. The first section presents an algorithm in which each disk has more than one block. This allows us to consider the RAID Level 5 architecture in its entirety. The second section describes an algorithm for another RAID architecture, the RAID Level 6, and shows how we used our consistency property to find an error in this algorithm.

4.1 Disks with More than One Block

In the algorithm we considered in the previous sections, we modeled each disk as having only one block of data. Since the disk array is composed of n data disks, this implies that files have a maximum length of n. One natural extension of the algorithm is to allow disks to have an unbounded number m of blocks.

This extension allows us to represent the RAID Level 5 architecture in its entirety. The RAID Level 5 architecture is block-interleaved with distributed parity. A file is divided into blocks that are placed on several disks and parity blocks are distributed in a left-symmetric fashion, meaning that they are located on a diagonal as shown in Figure 4-1. In this figure, `P0` is the parity covering blocks `0`, `1` and `2`; `P1` is the parity block covering `3`, `4` and `5`, etc...

In an architecture where there is a single parity disk, the parity disk is accessed every time

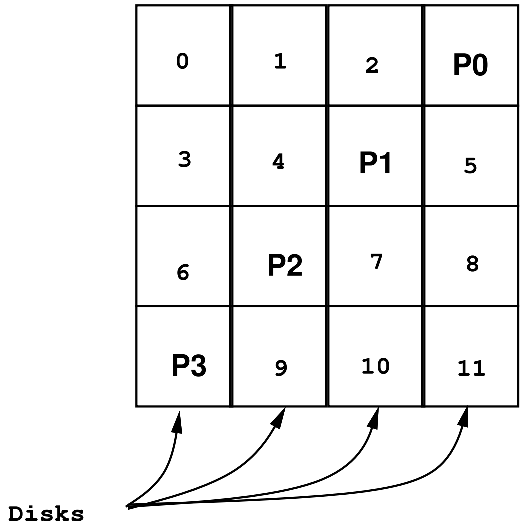

Figure 4-1: RAID Level 5 Architecture

it needs to be updated. Distributing the parity relieves the load on that disk and also makes $n+1$ disks available for data. So this scheme improves the overall performance.

We can represent the extension of having m blocks per disk in the following way. We introduce a new automaton *HLController* for High Level Controller that interfaces with the user. For each parity group we instantiate a *Controller* and a set of graphs identical to the ones we introduced before. Note that a parity group is a set of n data blocks together with the parity block that covers them.

The role of *HLController* is to find out what parity groups are going to be used and determine what disks are in the *UsedDisk* set for each parity group. Then *HLController* passes this information to each relevant *Controller*, along with the data to be written if any, and an indication of which disk has failed. Since the controllers do not share any data, they can run concurrently. When the controllers finish reading/writing their parity groups, they return to *HLController*, that in turn gets back to the user.

Note that this extension also allows us to have a system with more than one file. In this case *HLController* must maintain information about where the blocks of a file are, and deal with issues of allocation.

We can model *HLController* as an I/O Automaton. The algorithm is represented by the composition of this automaton with the controllers, graphs and disks automata. The proof of correctness for this algorithm is similar to the one we carried out before. Since the controllers do not share any data, this new composition does not require any special consideration.

4.2 Verifying Controller Algorithms for other RAID Architectures

In this section, we turn our attention to another RAID architecture: the RAID Level 6 system [Gibson95], which is two-fault tolerant. The general controller algorithm is identical to the one for RAID Level 5, i.e. after receiving and operation from the user, the controller chooses a graph to execute based on the state of the disk array; if that graph fails then the controller discards that graph and chooses another one to complete the operation.

The algorithm for the controller of the RAID Level 6 architecture differs from the algorithm we considered previously in the set of graphs available, and the logic for choosing them. We can model the RAID Level 6 algorithm using I/O Automata. The disk automaton will be identical to the one we used before, assuming we consider one block per disk again.

One essential property of graphs of controller algorithms that use Courtright and Gibson's error recovery method, is that they must satisfy General Consistency (Lemma 3.2.17). This property states that at the end of execution of a graph, all the disks D_i that are not to be written, satisfy two conditions:

1. If the disk has not failed, $D_i.block = D_i.hist$, and
2. If $VB(i)$ is defined, $VB(i) = D_i.hist$.

This property can be used for the RAID Level 6 architecture if we redefine the VB value. This value basically captures the architecture and the expression of the General Consistency property is the same for all systems.

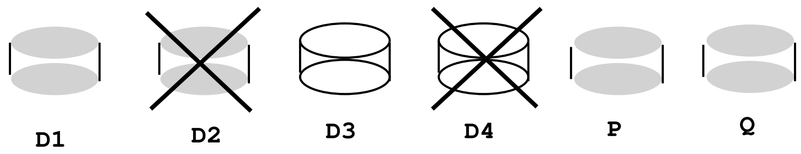

Figure 4-2: RAID Level 6 Architecture

4.2.1 RAID Level 6: Architecture Details

The RAID Level 5 system has a parity block for every n blocks. This allows the system to tolerate one disk failure. The RAID Level 6 system is an extension of RAID Level 5. It has two parity blocks per parity group and tolerates two failures. The first parity block (`P`) is identical to the parity in RAID Level 5. It is computed by performing the XOR of all the blocks in that parity group. The second parity (`Q`) is computed using Reed-Solomon codes. Figure 4-2 presents the RAID Level 6 architecture.

The graphs for RAID Level 6 are similar to the graphs for RAID Level 5. The controller can perform the Small Write and Reconstruct Write. In this case the graph must also update the second parity.

4.2.2 RAID Level 6: New Definition for VB

First we must define what it means for the VB value to be defined in this architecture. The VB value of a disk D_i is defined, i.e. $\mathit{DefinedVB} = \mathit{True}$, if and only if there is at most one failure among all disks other than D_i.

For this system, the VB value of a disk is computed as follows. If no other disks have failed, we compute VB using `P`. If another data disk has failed, VB is computed using both `P` and `Q`. If `P` has failed we compute VB using `Q`. Finally, if `Q` has failed, we compute it using `P`.

Note that, in the above explanation, we only present what data is needed to compute VB and we omit how to compute it. But this is enough for the purposes of applying the General Consistency property.

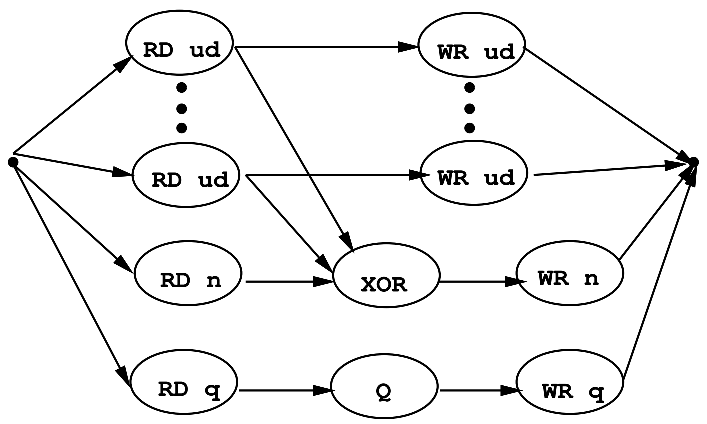

Figure 4-3: Small Write for RAID Level 6 - Non-recoverable Graph

4.2.3 Error Found in a RAID Level 6 DAG

We found an error in the Small-Write graph for RAID Level 6, that appears in [Gibson95]. This graph did not satisfy General Consistency and our property helped in finding a counterexample. The graph is shown in Figure 4-3. It consists of reading the disks to be written, reading the old `P` and `Q` values, computing the new parities and writing the new data and parities.

This graph does not satisfy the second condition of General Consistency. Consider the disk array shown in Figure 4-2. Assume that disks D_1 and D_2 are in the *UsedDisk* set and thus are the disks to be written. Assume that the graph reads and writes disk D_1. Then assume that D_2 fails before having been read. At this point the graph cannot complete successfully and the disk array has been partially updated. Now if D_4 fails, its VB value is different from its *hist* value, since the *block* value of D_1 has changed, but the *block* values of P and Q have not. Disk D_4 is not to be written and its VB is different from its *hist* at the end of the execution of the graph. Therefore this graph does not satisfy General Consistency.

The Small Write graph appears in [Gibson95]. It seems to be the case that there does not exist a DAG that performs the Small Write operation, while satisfying the General Consistency property, using Courtright and Gibson's error recovery method. In a recent work Gibson et al. have used a different controller algorithm for the RAID Level 6 that does not have this problem. This controller algorithm uses roll-away error recovery [Courtright96], rather than Courtright and Gibson's method [Courtright94].

Chapter 5

Conclusions

Summary

We proved the correctness of a controller algorithm for the RAID Level 5 system. We expressed the algorithm and its specification using I/O Automata and proved that the algorithm satisfies its specification by using the proof by simulation technique.

We then presented two extensions of this study. The first one is having more than one block per disk. This extension allows us to represent the RAID Level 5 architecture in its entirety. We did not show the proof of correctness for this extended algorithm, but it is similar to the one we presented.

The second extension is considering a controller algorithm for the RAID Level 6 system. We used the formalization of our General Consistency property, to find an error in the Small Write DAG of a RAID Level 6 algorithm.

Formal Methods and Practice

It is useful to employ formal methods to validate RAID controller algorithms because these algorithms are difficult to test and to reason about. When applied in early stages of design, formal methods can unveil errors that would be expensive to correct if they were propagated to implementation stages. However, practitioners generally do not use formal methods, because these are considered to be expensive. The time and effort required by hand-proofs or by proofs done with semi-automatic theorem provers, are considered to be prohibitively expensive.

Researchers have proposed that practitioners would use formal methods, if fully automatic tools were available. These tools would have to be easy to learn and to use. One example is the success of model checking in the hardware domain [Clarke94]. A model checker takes, as input, the description of a system and a property to verify. Then it generates the state-space of the system and checks it exhaustively. It outputs true if the system satisfies the property, and false otherwise. In the latter case, the model checker also outputs a counterexample. Model checkers cannot be used directly in the software domain, because software systems are not finite state machines, and model checkers can only verify finite state machines. This is the reason why researchers have been considering methods to combine theorem proving and model checking. However theorem provers augmented with model checking capabilities are semi-automatic tools that require the user to participate in the proof of correctness. Therefore, these tools would be considered expensive to use by practitioners.

On the one hand, practitioners would rather have fully automatic tools, and on the other, theoreticians see the benefits of semi-automatic ones. In these tools, the designer is involved in the proof of correctness, and can learn essential information about why the algorithm is correct. This information is very useful to the designer, for future design. The tradeoff for how much automation there should be in a verification tool for software systems is not clear.

A solution to this problem is to have "little" software verification tools. From the point of view of verification, hardware systems can be seen as a subdomain of software systems, namely a hardware system is a software system that has only booleans as data structures and that is finite state. Just as model checking is suitable for the hardware domain, we could develop fully automatic verification tools for other restricted domains of software. These tools would be developed by restricting a software domain, and proving the correctness of algorithms in that domain. These proofs would reveal essential information about why these algorithms are correct and a verification tool could be built based on it. This tool would be fully automatic and available to practitioners who would use the information without having to perform the proofs again. Instead of having one general-purpose software verification tool, designers would have many special purpose tools.

Future Work

Based on the previous discussion, we plan to build a verification tool for RAID controller algorithms that use Courtright and Gibson's error recovery method. This tool would take as input a definition for the VB value of a disk and a DAG, and would determine if the DAG satisfies consistency. By proving correctness of the RAID Level 5 algorithm, we found out why the algorithm is correct and formalized this information in the General Consistency property. A fully automatic verification tool for this property would allow designers to use this information without having to perform the proof again.

Bibliography

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

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

[Clarke94] Clarke, E., Grumberg, O., and Long, D. "Model Checking". *Proceedings of the International Summer School on Deductive Program Design.* Marktoberdorf, Germany, July 26 - August 27 1994.

[Courtright94] W. V. Courtright II and G. A. Gibson. "Backward error recovery in redundant disk arrays." *Proceedings of the 20th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG).* December 4–9 1994, pp. 63–74.

[Courtright96] William V. Courtright II, "A Transactional Approach to Redundant Disk Array Implementation." Computer Science Technical Report, 1996, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213.

[Gibson90] Garth Gibson. "Redundant Disk Arrays: Reliable, Parallel Secondary Storage". PhD thesis, University of California at Berkeley, 1990. Report UCB/CSD 91/613.

[Gibson93] G. A. Gibson and D. A. Patterson, "Designing disk arrays for high data reliability", *Journal of Parallel and Distributed Computing.* 17(1-2), 1993, 4-27.

[Gibson95] G. Gibson, W. Courtright II, M. Holland, and J. Zelenka, “RAIDframe: Rapid prototyping for disk arrays,” Computer Science Technical Report CMU-CS-95-200, Carnegie Mellon University, 1995.

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

[Reddy89] A. L. Narasimha Reddy and Prithviraj Banerjee, “An evaluation of multiple-disk I/O systems.” *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1680–1690.

[Kim86] M. Kim. “Synchronized Disk Interleaving”. *IEEE Transactions on Computers* 35(11), November 1986, pp 978-988.

[Lamport83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems,* 5(2):190–222, April 1983.

[Lawlor81] F. D. Lawlor. “Efficient Mass Storage Parity Recovery Mechanism”, *IBM Technical Disclosure Bulletin* 24(2):986-987, July 1981.

[Lynch87] N. Lynch and M. Tuttle. “Hierarchical correctness proofs for distributed algorithms.” Technical report MIT/LCS/TR-387, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.

[Lynch89] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI-Quaterly*, 2(3): 219-246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[Lynch95] Nancy Lynch and Frits Vaandrager. “Forward and Backward Simulations – Part I: Untimed Systems”. Information and Computation, 121(2), pages 214-233, September 1995.

[Lynch96] Nancy A. Lynch, “Distributed Algorithms”, Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[Patterson88] David A. Patterson, Garth A. Gibson, and Randy Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". *Proceedings SIGMOD International Conference on Data Management*, 1988, pp. 109-116.

[Patterson89] David A. Patterson, Peter Chen, Garth Gibson and Randy Katz. Introduction to Redundant Arrays of Inexpensive Disks (RAID). *Spring COMPCON'89* San Fransisco, CA, pp 112-17. IEEE, March 1989.

[Park86] Arvin Park and K. Balasubramanian. "Providing Fault Tolerance in Parallel Secondary Storage Systems". Technical Report CS-TR-057-86. Department of Computer Science, Princeton University, November 1986.

[Salem86] K. Salem and H. Garcia-Molina. "Disk Striping". *Proceedings of the 2nd International Conference on Data Engineering*, 1986, pp. 336–342.