Automation of RAID Controller Operations using ROSTI (Work in Progress)

Sameshan Perumal and P. S. Kritzinger
DNA Research Group
Department of Computer Science
University of Cape Town
Email: sperumal@cs.uct.ac.za

Abstract — The software underlying RAID Controllers is generally implementation specific, with no formal method for verification of correctness or optimal operation. We present work that formalises the specification of RAID protection schemes. From this formal specification, it should then be possible to automatically generate the logic for the corresponding RAID Controller, and analyse this for correctness and efficiency. The work is being tested within the framework provided by ROSTI (RAID Operations Simulator for Testing Implementations), a simulator we have developed that assesses the performance implications of various RAID architectures. Finally, we present possibilities for extending this scheme to automated recovery from errors during operation.

Index Terms — RAID, Correctness, Validation, Specification

I. INTRODUCTION

THE applications for RAID (Redundant Array of Independent Disks) [1],[2] in ESS (Enterprise Storage Systems) continue to grow as applications demand ever more secondary storage. To meet this demand, new protection schemes are being used in commercial systems, such as RAID 6 [4]. These schemes increase the number of disk failures that are tolerable before data loss occurs. In organisations with large arrays of disks, this is of particular importance, since the Mean Time To Failure (MTTF) of the array decreases as the number of disks increase. The increased number of disks across which data can be distributed can also improve data transfer rates for certain types of I/O operations, as well as allowing multiple operations to occur in parallel.

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

Our work primarily focuses on the performance implications of these schemes, with specific interest in the response time and throughput of the system. To this end, we have developed ROSTI (RAID Operations Simulator for Testing Implementations). During the implementation of ROSTI, it became apparent that a generalisation of the operation of a RAID Controller would be useful for comparing various such schemes.

To achieve this, we are attempting to construct a formal description of RAID protection schemes, independent of the implementation. This would allow the required individual operations for a given request to be derived from this description, whilst ensuring that such operations are optimal in terms of disks accessed and maximising parallelism in the array. Finally, this approach lends itself to integration with the work of Courtright [3], which describes a means of ensuring that array operations are executed correctly, even during disk failures.

II. ROSTI

ROSTI is a simulator we have built using OMNET++ [4], an open-source event-driven simulation framework. It works with a small set of module types, such as workload generators, storage devices, and RAID controllers. The communication protocol between each type of module is pre-defined. This allows for complicated storage architectures to be created from a set of building blocks, rather than by modifying code. This greatly simplifies the process of testing multiple configurations of interest. This also allows for simple extensions by creating custom modules that conform to the communication protocol.

The formalisation presented in the rest of this paper was inspired by the work done in implementing RAID Controller modules for ROSTI. We noticed that much of the code for a given operation (read or write) was repeated across code schemes, and could be easily generalised. It was discovered that the position of data on disk, rather than the scheme being utilised, accounted for the most significant differences in control logic. Finally, over several iterations of improvements to the Controller modules, it was discovered that several cases exhibited either incorrect or suboptimal behaviour as a result of coding errors. The work presented here is an attempt to prevent these sorts of errors by automating the process by which operations are processed, according to a well defined algorithm.

III. RAID TERMINOLOGY

Data in a RAID Array is distributed across several disks

in *Stripes*. Each *Stripe* consists of blocks referred to as *Strips*, which are distributed across available disks. Internally, each *Strip* is further subdivided into *Chunks*, which are the smallest unit of data we deal with. Within a *Stripe*, we refer to a group of *Chunks* at the same position in each *Strip* as a row. These concepts are illustrated in Figure 1.

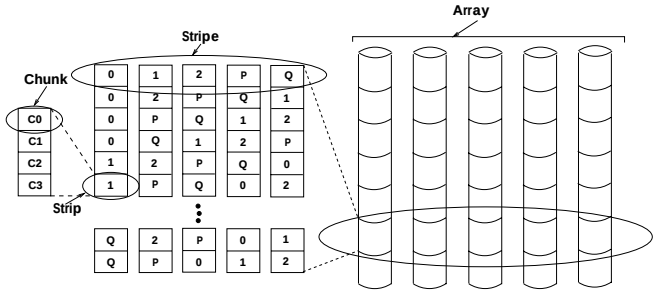

Figure 1 : Terminology used in reference to RAID systems

IV. FORMALISING CONTROLLER OPERATION

A. RAID LAYOUT SPECIFICATION

The layout of a RAID Array refers to the number and arrangement of Stripes, Strips and Chunks in the array. In Figure 1, the layout is for the RAID6 scheme. In general, this amounts to defining the type of each chunk as being either *data* (the chunk contains user data) or *parity* (data protection information). One approach is to hard code an algorithm based on the scheme in use.

Our approach is to differentiate between three types of parity:

- **Horizontal**: All parity chunks lie in a single row for each stripe.
- **Vertical**: All parity chunks lie in a single strip, for each stripe (eg. RAID 4).
- **Diagonal**: The position of parity chunks lie within a single strip, but the location of this strip rotates in a round-robin fashion (eg. RAID 6, see parity chunks P and Q in [5]).

Using these parity types, it is possible to specify a number of types of parities to apply to a layout, together with an order in which to apply them. This automatically produces the correct layouts for any combination of Stripe, Strip and Chunk size settings. A proof of concept of this technique has been implemented, and was tested with combinations of all 3 types of parity. In all cases, the layout produced was correct.

B. SPECIFYING PROTECTION GROUPS

The usefulness of the above formalism becomes more apparent when we consider that a given type of parity protects data chunks in consistent manner. Specifically, a Horizontal parity chunks protects all data chunks in its Strip, while Vertical and Diagonal parity chunks protect all data chunks in their Row. This web of protection can easily be represented using a dependency graph, linking data chunks to the parity chunks that protect them. This graph can be built as the above layout process occurs, based on a simple set of rules. The use of a dependency graph also allows for more complicated dependency mappings, but using the same three parity types. Thus, we could conceive of a Horizontal parity that protects all data chunks that lie in the same diagonal.

C. DERIVING ARRAY OPERATIONS

The dependency graph created above can now be used to automate the operation of the RAID Controller. Specifically, if data needs to be reconstructed for a read operation due to a failed disk, the dependency graph can be used to determine the list of all parity chunks protecting the data chunks in question. Using this information, it is possible to determine the minimum number of disk accesses required to read in appropriate parity and recover the data, thus addressing the optimality issue raised earlier.

For a write, parity chunks need to be updated to reflect changes in data. Using the dependency graph, we can ensure that all parity chunks related to changed data chunks are correctly updated. We can even minimise the number of disk I/O's issued for certain operations (eg. Large Stripe Write).

Since the operations are extracted from the dependency graph, rather than coded by hand, it is possible to guarantee that operations are executed correctly and efficiently, under the assumption that the dependency graph is correctly constructed. Additionally, a trace of the operation of a controller using this scheme can be used to verify whether another, hand-coded controller is operating correctly.

V. CORRECTNESS OF OPERATION

Although a RAID Controller may operate correctly during normal operation, it may still not handle error scenarios correctly. The most critical time during which errors can occur is if a disk failure happens while an I/O Request is being processed. The array may be left in an inconsistent state, and hand-coding of recovery schemes for all possible cases is a time-consuming and error-prone task. The work of [3] presents a formal method to prevent this, using Directed Acyclic Graphs (DAGs). It is our intention to use the dependency graph described above to automatically generate these DAG's, where possible, thus creating an end-to-end formalism governing the operation of the RAID Controller.

REFERENCES

[1] David A. Patterson, Garth Gibson and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", in *Proc. ACM SIGMOD International Conference on Management of data*, 1988.

[2] Sameshan Perumal and Pieter , "A Tutorial on RAID Storage Systems", Tech. Report CS04-05-00, Dept. of Computer Science, University of Cape Town, 2004.

[3] William V. Courtright II, "A Transactional Approach to Redundant Disk Array Implementation," Ph.D. dissertation, Carnegie Mellon Univ., 1997.

[4] András Varga, "The OMNeT++ Discrete Event Simulation System", in *Proc. European Simulation Multiconference*, 2001.

[5] P. M. Chen, et. al., "RAID: high-performance, reliable secondary storage," ***ACM Computing Surveys***, vol. 26, issue 2, pp. 145–185, June 1994.