Computer Science Department
University of Crete

DARC: *Design and Evaluation of an I/O Controller for Data Protection*

Master's Thesis

Markos Fountoulakis

May 2010

Heraklion, Greece

University of Crete
Computer Science Department

DARC: Design and Evaluation of an I/O Controller for Data Protection

Thesis submitted by
Markos Fountoulakis
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________________
Markos Fountoulakis

Committee approvals: ______________________________
Angelos Bilas
Associate Professor, Thesis Supervisor

____
Kostas Magoutis
Researcher

____
Dimitris Nikolopoulos
Associate Professor

Departmental approval: ______________________________
Panagiotis Trahanias
Professor, Director of Graduate Studies

Heraklion, May 2010

Abstract

Lately, with increasing disk capacities, there is increased concern about protection from data errors, beyond masking of device failures. In this thesis, we present a prototype I/O stack for storage controllers that encompasses two data protection features: (a) persistent checksums to protect data at-rest from silent errors and (b) block-level versioning to allow protection from user errors. Although these techniques have been previously used either at the device level (checksums) or at the host (versioning), in this work we implement these features in the storage controller, which allows us to use any type of storage devices as well as any type of host I/O stack. The main challenge in our approach is to deal with persistent metadata in the controller I/O path. Our main contribution is to show the implications of introducing metadata at this level and to deal with the performance issues that arise. Overall, we find that data protection features can be incorporated in the I/O path with a performance penalty in the range of 12% to 25%, offering much stronger data protection guarantees than today's commodity storage servers.

Supervisor Professor: Angelos Bilas

Περίληψη

Σχεδίαση και Αξιολόγηση ενός ελεγκτή Εισόδου Εξόδου για Προστασία Δεδομένων

Τον τελευταίο καιρό οι εφαρμογές απαιτούν περισσότερα χαρακτηριστικά προστασίας δεδομένων, πέρα από την απόκρυψη αποτυχιών υλικού, ώστε να επιτύχουν ακεραιότητα δεδομένων και να επαναφέρουν τυχόν λάθη χρήστη. Σε αυτή τη μεταπτυχιακή εργασία παρουσιάζουμε ένα πρωτότυπο στοίβας Εισόδου/Εξόδου (Ε/Ε) με εκτεταμένα χαρακτηριστικά προστασίας δεδομένων: (α) μονίμως διατηρούμενα ψηφία ελέγχου για την προστασία αποθηκευμένων δεδομένων από σιωπηλά λάθη, και (β) τήρηση εκδόσεων σε επίπεδο μπλοκ για την προστασία από λάθη χρηστών. Παρά το γεγονός ότι οι τεχνικές αυτές έχουν χρησιμοποιηθεί στο παρελθόν είτε στο επίπεδο συσκευής (checksums) είτε στο επίπεδο του διακομιστή (versioning), σε αυτή τη δουλειά υλοποιούμε αυτά τα χαρακτηριστικά στο επίπεδο ελεγκτή αποθήκευσης, το οποίο μας επιτρέπει να χρησιμοποιήσουμε οποιουδήποτε τύπου συσκευές αποθήκευσης καθώς και οποιαδήποτε στοίβα Ε/Ε. Η προσέγγισή μας απαιτεί ο ελεγκτής να διαχειριστεί μόνιμα αποθηκευμένα μεταδεδομένα στο μονοπάτι Ε/Ε. Η κύρια συνεισφορά μας είναι η εξέταση των συνεπειών της εισαγωγής μεταδεδομένων σε αυτό το επίπεδο και η αντιμετώπιση των θεμάτων επίδοσης που ανακύπτουν. Βρίσκουμε ότι τα χαρακτηριστικά προστασίας δεδομένων μπορούν να ενσωματωθούν στο μονοπάτι Ε/Ε με επιδείνωση επίδοσης στο εύρος 12% - 15%, προσφέροντας πολύ ισχυρότερες εγγυήσεις προστασίας δεδομένων από τους σημερινούς εξυπηρετητές αποθήκευσης.

Επόπτης καθηγητής: Άγγελος Μπίλας

Acknowledgements

We thankfully acknowledge the support of the European Commission under the 6th and 7th Framework Programs through the STREAM (FP7-ICT-216181), HiPEAC (NoE-004408), and HiPEAC2 (FP7-ICT-217068) projects.

I would like to thank my supervisor, Angelos Bilas and my colleauges, Manolis Marazakis and Michail D. Flouris.

Markos Fountoulakis
Heraklion, May 2010

Ευχαριστίες

Ευχαριστώ ιδιαιτέρως τον Απροσάρμοστο Μαραζάκη για την καταπίεση και τη συγγραφή. Ευχαριστώ τον Ανυποψίαστο Φλουρή για τη θετική ενέργεια που εξέπεμπε όταν δεν αναρωτιόταν τι κάνω. Ευχαριστώ τον Αχαΐρευτο Νικηφόρο για τη γκρίνια και το Γιώργο Τζενάκη για τη συμβολή του στο χρονοπρογραμματισμό. Ευχαριστώ τον Άκυρο Κλονάτο και τον Αδιόρθωτο Μακατό για τον αποπροσανατολισμό. Επίσης ευχαριστώ την Κρύστα και τη Ζίρα. Ευχαριστώ τον Ανυπόφορο Πασσά για την απουσία του όταν έγραφα αυτήν την εργασία. Ευχαριστώ το Μιχάλη Αλβανό για τη διαπίστωση πως όλα τα προβλήματα έχουν μια προφανή λύση που αγνοώ. Ευχαριστώ τη Ζωή Σεμπέπου και τον Άσφαιρο Τσαλιαγκό για την πανηγυρική ατμόσφαιρα. Ευχαριστώ τον Αμόρφωτο Χασάπη για το χαμόγελο της Κρεστ. Ευχαριστώ το Γιάννη Μανουσάκη και του εύχομαι ευτυχισμένο αρραβώνα. Ευχαριστώ την οικογένεια μου που με αντέχουν ακόμα. Ευχαριστώ τους φίλους μου και τη Μαρία.

Μάρκος Φουντουλάκης
Ηράκλειο, Μάιος 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last three decades there has been a dramatic increase of storage density in magnetic and optical media, which has led to significantly lower cost per capacity unit. Today, most I/O stacks already encompass RAID features for protecting from device failures. However, recent studies of large magnetic disk populations indicate that there is also a significant failure rate of hardware and software components in the I/O path, many of which are silent until data are used by applications [26, 34, 28, 4, 19]. Dealing with silent data errors on storage devices becomes critical as more and more data are stored on-line, on low-cost disks. In addition, storage snapshots and versions have been an important technique for dealing with various types of failures, including human errors. As the amount of information maintained in storage systems increases, both silent and human errors become more pressing problems. With current technology trends, it is important to examine cost-effective solutions to those problems that are applicable to commodity systems.

Today, solutions for silent errors appear in high-end disks that provide additional block space for storing checksum information, e.g 520-byte sectors to store 512 bytes of user-accessible data [1], and may perform data checks during transfers. Such features are not available in commodity disks and I/O controllers widely used today in most storage systems. Integrity checking introduces two challenges. First, computing the checksum needs to occur at a point in the I/O path that minimizes the impact on I/O throughput. Second, it requires maintaining checksums per data block in a persistent manner, and performing the checksum retrieval and comparison with minimum impact on response time. Supporting additional data protection features, such as versioning of storage volumes, also necessitates keeping persistent data structures with sizes that grow with the capacity of the original volumes. The key challenge is to sustain high I/O rates for versioned volumes, while verifying the integrity of the accessed data blocks. State information needs to be computed, stored, and updated in-line with I/O processing.

In this thesis we design and implement a high-performance storage controller that provides silent transparent silent error detection and data versioning. There are studies of virtualized storage systems or storage controllers supporting such functionality [22, 9, 20]. To our knowledge, no currently available commodity controller deals with these issues at high I/O rates. In addition, although data protection features such as checksums are supported by enterprise-grade storage systems, there is no detailed full-system design and evaluation study in the open literature. With high-throughput connectivity to hosts, controllers exhibit challenging performance and failure-mode characteristics. To achieve high performance, the controller has to stage transfers of data between the host and the storage devices. For this reason it needs to provide an efficient data-path. High I/O rates put pressure on the controller's hardware resources, including its CPU, internal memory, and DMA engines. Considering current trends in system interconnects (e.g. PCI-Express v.2), as well as in storage device technology, e.g. solid-state drives, we expect I/O controllers to become the main bottleneck in overall system performance for I/O intensive workloads.

Our system, DARC *(DAta pRotection Controller)* is able to deliver high I/O performance, both in terms of throughput and IOPS (I/O operations per second), while transparently and comprehensively offering protection from three major types of data storage hazards:

1. *Data integrity for data-at-rest* by employing a combination of two mechanisms: (i) error detection through the computation and persistent storage of data checksums per block, and (ii) error correction using existing data redundancy schemes.

2. *Human errors* through *transparent online versioning* at the block level. Frequent storage versions reduce the risk for data lost due to human errors and allow users to easily recover accidentally deleted or modified data by accessing a previously captured version [12, 33]. Block-level versioning can also achieve recovery from data corruption, due to buggy OS or filesystem software or a malicious attack compromising the system. In such situations, versioning allows users to recover the system to a safe and operating state with the minimum possible downtime [38, 39, 6].

3. *Storage device failures* are masked, and data availability is provided, by employing traditional RAID techniques that provide availability.

DARC does not provide high-availability features for the system itself, such as redundancy in the I/O path, memory mirroring to a second controller, and fail-over in the event of controller failure that are typical for high-end, enterprise storage. These features are outside the scope of this work. We focus on data protection features that we believe will become increasingly important for all storage systems.

Data protection features can be offered at various levels in the I/O stack. We provide error detection/correction and versioning at the storage controller level to achieve transparency both to the host file-system as well as the block devices used. In addition to transparency, this approach allows us to perform expensive bit operations during data transfers and using controller DMA support.

We discuss the challenges of implementing a full system and identify limitations. We present a detailed evaluation of our system and its data protection features, contrasting it to a commercially available controller with comparable hardware resources. We contribute to the discussion of whether storage systems will be a deciding factor in overall system performance.

The rest of this thesis is organized as follows. Chapter 2 discusses related work. In Chapter 3 we present the key considerations, and our corresponding design decisions, in building a high performance I/O controller with data protection features, based on a specific embedded system platform. In Chapter 4 we present experimental results, comparing our prototype with a commercially-available controller based on the same hardware platform. Finally, Chapter 5 summarizes our conclusions.

Chapter 2

Related Work

Modern high-end storage systems incorporate increasingly advanced features at the block level, such as thin provisioning, snapshots, volume management, data deduplication [35], and block remapping. AutoRAID [42] implements a two-level managed storage hierarchy within the storage array controller. Implementing such features requires storing substantial block-level metadata, however, only for transparently re-mapping individual blocks. We use similar mappings to implement versioning, but also store checksum values per block, to be used in transparent integrity checks upon access. The need to maintain metadata persistent affects the design and performance of the I/O path, as I/O processing becomes more stateful than in the case of a simpler RAID controller.

CERN has published a study of data corruption risks [26], providing estimates of the probability of various errors leading to data corruption. The error rates are estimated at the 10^{-7} level, considerably higher than the 10^{-14} or 10^{-15} level expected from SATA or SAS/FC disk drives. This study considers the entire chain of components from disks to host CPUs and back, including network data transfers. A main conclusion is that mitigating data corruption risks, via checksum checks, "will lead to a doubling of the original required I/O performance on the disk servers" and "an increase of the available CPU capacity on the disk servers". This conclusion is based on the premise that applications will read files twice, first for a checksum verification and then again for processing. We believe that providing checksum checking at the block-level in commodity systems offers a more efficient alternative.

Recent studies of large magnetic disk populations indicate that there is a significant failure rate of hardware and/or software components in the I/O path, many of which are silent until the data are accessed [34, 28, 4, 19]. Studies of I/O software components, especially filesystems, have shown that they are complex, buggy, and hard to scale, debug, and tune for specific application domains [29, 44, 30, 17]. We believe that these argue for increased data protection features at the block level.

Storage versioning has been previously examined mostly at the filesystem level with Elephant [33], Plan 9 [27], ZFS [5] and other filesystems [25, 31, 38]. Filesystem-dependent versioning at the block level has also been implemented in commercial products, such as NetApp's WAFL [12]. In this work, instead of implementing versioning at the host side, we argue that versioning functionality can be transparently implemented within storage controllers. Pushing different types of advanced functionality closer to the disk has been previously been proposed [2, 8, 10]. Self-securing storage [39] and CVFS [38] also operate at the filesystem level, but they are not intended for long-term data protection, retaining versions only during a *detection window*.

Transparent block-level versioning was first proposed in Petal [21], which supports a copy-on-write algorithm. *DARC* is based on the remap-on-write approach of [6]. HP Olive [3] and Timeline [24] also use versioning at the block level. However, they operate within the host OS, not at the I/O controller level, and they are targeted towards scalable distributed storage. Olive, implemented within the FAB clustered block-level storage system [32], offers the ability to create writable storage branches, while maintaining high-availability. Timeline supports versioning for a persistent object store distributed over a wide area network. Systems such as VDisk [43] have focused on "secure" versioning at the block level, where versioning is protected within a virtual machine. The advantage of VDisk is that storage data and metadata are additionally protected from operating system bugs that may cause corruption. We consider this functionality to be complementary to what our system offers. On the other hand, VDisk uses expensive metadata management techniques which result in 50% performance penalty on block writes. Laden et al [20] propose 4 storage architectures for CDP support in storage controllers. They categorize our approach as similar to the logging architecture, but with more efficient access to newer versions.

The Panasas tiered parity [18] scheme recognizes the need for comprehensive data protection, with three levels of parity-based data integrity checks: across devices (by RAID controllers), within each device, and finally at the client-side. In our work, we embed data integrity checks within the I/O controller, with a focus on commodity-grade systems and with the explicit goal to evaluate the performance impact.

Parity layouts across disks have been studied in [13], aiming to improve recovery time and user-perceived throughput during RAID array reconstruction. In our work, we have opted for a simple mapping of checksums to dedicated metadata space on the storage devices, and have so far not addressed the issue of array reconstruction time. Since even users of commodity-grade storage systems increasingly require continuous operation, for rapidly increasing volumes of data, this issue will require further investigation.

Data integrity assurance techniques are explored in [37]. Type-safe disks [36] are aware of the pointer relationships between disk blocks and higher layers, and can be extended to incorporate data integrity assurance techniques with low overhead [41], at the disk, rather than the storage controller level. Data integrity and high performance is also discussed in [11], where the notion of the I/O shepherd is introduced. The I/O shepherd exports an interface to the file-system so that it can implement diverse reliability policies via block-level interposition.

Chapter 3

System Design and Architecture

The key challenges in building a high-performance storage controller for data protection are:

1. High-rate and low-latency data transfers between the host and the controller.

2. Efficient buffer management.

3. Efficient context and event scheduling.

4. Low overhead usage of data-block checksums.

5. Integration of persistent metadata management.

We address these challenges, taking into consideration the characteristics of a specific embedded system platform.

3.1 Preliminaries

In the host-controller I/O path there exist the notions of I/O commands, I/O completions and control information in general.

I/O commands are descriptors of user generated I/O requests. They are usually accompanied by scatter-gather lists of host memory physical addresses. Additional control information can be transferred alongside I/O commands such as custom commands for the storage virtualization layers (e.g. take a snapshot, mark a segment as the start of a CRC chain etc). This kind of information, that is passed from the host to the controller, is part of the I/O issue path.

I/O completions are references to user generated I/O requests along with status code (e.g. success, failure etc). This kind of information is generated at the controller side and reaches the host side, travelling through the I/O completion path.

3.2 Embedded System Platform

We have developed the *DARC* prototype using a development board from Intel, based on the 81348 SoC [14]. This board offers two non-coherent XScale

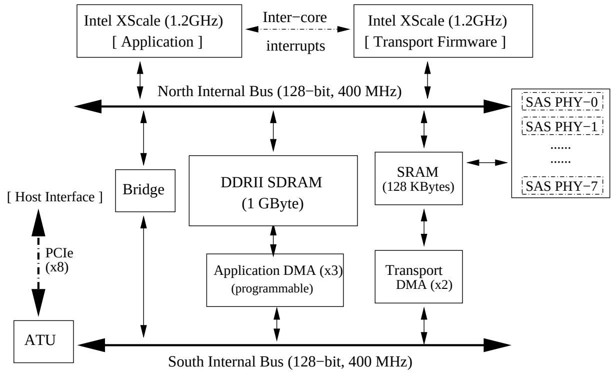


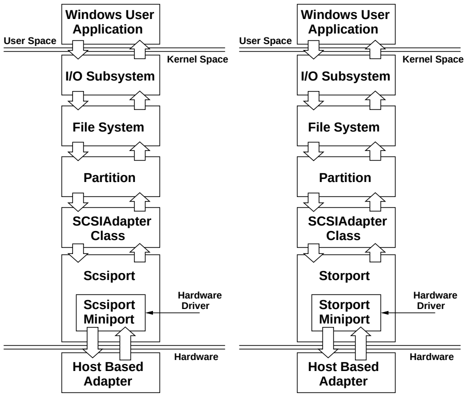
FIGURE 3.1: Data path in *DARC*.

cores, running at 1.2GHz, an 8-port SAS controller, three programmable DMA engines, and 8-lane PCI-Express (v. 1.1) host connectivity. The run-time environment on this board is based on the Linux kernel, v.2.6.24, augmented with the Intel XScale IOP patches [15]. Figure 3.1 shows a simplified view of the internal organization of the Intel 81348 SoC board [14]. The board has two 128-bit buses, running at 400 MHz. The XScale cores and the 1GB DDR-II SDRAM are connected by the North Bus, while three programmable Application DMA (ADMA) channels are on the South Bus, and are connected to the memory controller. ADMA channels support chaining of DMA descriptors, can provide information about the current transfer taking place, and are able to interrupt the application CPU when one or more transfers have finished. ADMA channels can perform XOR calculations and compute a CRC32-C checksum while performing data transfers to/from the host. The host communicates with the board through the *Messaging Unit* of the Address Translation Unit (ATU). The *Transport core* runs the *SAS transport firmware* exclusively, while the *Application core* runs Linux. The transport firmware sends SCSI commands to the drives, and is accessed by a Linux SCSI driver (`isc813xx`, part of Intel's XScale IOP patches) running on the Application core. Our modules run entirely on the Application core, and access disk drives by issuing block-level commands. The block-level commands are translated, via the Linux SCSI driver framework to SCSI command for the `isc813xx` driver.

3.3 Host System Software Platform

We use Microsoft Windows as our host operating system. Microsoft Windows give two choices for programming SCSI devices, such as our storage controller, inside the operating system kernel. The SCSIport framework and the more recent STORport framework (as of Windows kernel version 5.2). We can see in Figure 3.2 the Windows kernel I/O stack for both SCSIport and STORport. We notice that both frameworks are placed at the same place in the host system I/O path. However, there are significant differences in their performance characteristics.

We can see in Figure 3.3 the most important differences between SCSIport



(a) SCSI-port (b) STOR-port

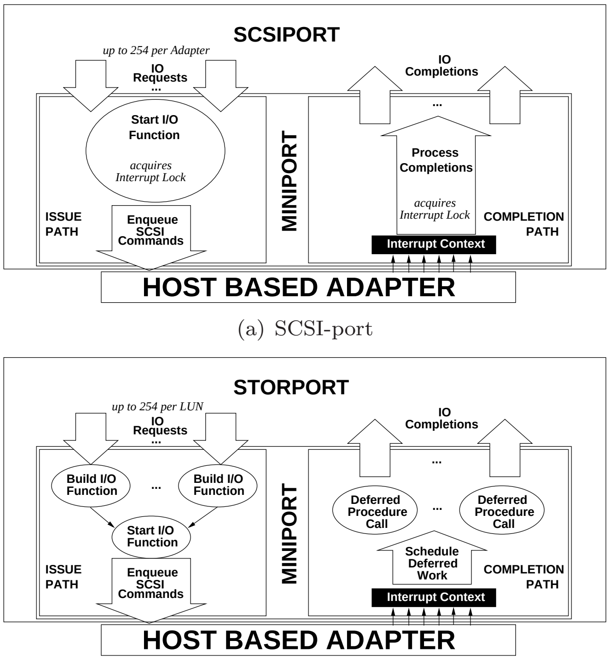
Figure 3.2: Windows I/O stack.

and STORport. First of all, SCSIport has a practical limitation of 254 maximum outstanding I/O requests per adapter (storage controller), whereas STORport is limited per LUN (Logical Unit Number, usually characterises a disk device). Moreover, SCSIport acquires an interrupt lock for all I/O processing in the I/O issue path and the I/O completion path. Microsoft calls this half-duplex mode, because I/O commands can not be processed at the same time as the I/O completions. In STORport, issue path processing can be parallelized in the build I/O routines, and completion processing can be scheduled at bottom half context, resulting in what is called the full-duplex mode. In our implementation we use the STORport framework to take advantage of all the performance benefits it provides.

3.4 Host-controller I/O Path

In Figure 3.4 we can see the full system I/O stack of our design. Application generated I/O requests traverse the host operating system I/O stack until they are ultimately translated into SCSI I/O commands. These are propagated to the controller using suitable data structures, usually some kind of command queues, through an I/O device bus. The SCSI commands are then translated into linux block requests inside the controller. These block requests are processed in the virtualization layers of the controller side I/O stack, and are finally completed. When they are completed, suitable completion messages are generated to be sent to the host SCSI layer. These messages pass through the completion queues to reach the host, across the I/O bus. Finally, the host level SCSI commands are marked as completed, and the host application is notified of the original I/O request completion.

I/O commands and control information need to be transferred from the host to the controller, and I/O completions or other information from the controller memory must be able to reach the host memory. User data are transferred in both directions. Two mechanisms can be used to perform these transfers: Programmed I/O (PIO) and Direct Memory Access (DMA). PIO occurs as a result of host processor loads and stores targeted at the controller's exported address space, whereas DMA is performed by specialized hardware units of the



(a) SCSI-port

(b) STOR-port

Figure 3.3: Windows framework characteristics.

controller. DMA is best suited for large transfers, supports high rates, but suffers from set-up time latency. PIO offers very low response times, but its throughput potential is at least an order of magnitude lower than DMA and costs host CPU cycles. Data buffers are typically transferred via DMA.

In the controller-to-host direction, one can insert completion information in the data-buffer traffic of the DMA engines, or even piggy-back this completion information along with its corresponding data buffers. In the latter case, there is no need to check separately for the completion of the data transfer: when the I/O completion arrives at the host, the corresponding data will have arrived as well, in the case of read I/O requests.

In the host-to-controller direction, DMA is not the best option. The main problem with DMA transfers of I/O commands is that they can only be initiated by the controller itself. This means that the controller must frequently suspend its processing of I/O commands and completions in order to query the host for new I/O commands and then initiate a DMA transfer for these commands. There is a risk of completing the processing in the controller side and only then initiating the transfer of new commands, which results in not exploiting the natural overlap of hardware resource usage, and suffering the full extent of DMA latency. To make matters worse, even if the controller manages to initiate a DMA for the commands at the exact moment they are available, if there are only a few outstanding I/O requests we will still suffer relatively long response times due to the DMA overhead. Host-initiated PIO does not suffer from any of these drawbacks, so the only concerns are the low throughput and host CPU utilization.

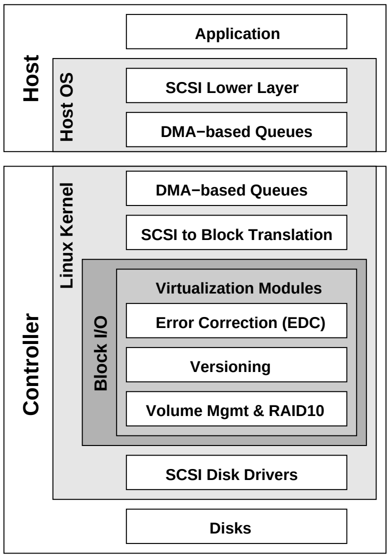


Figure 3.4: *DARC* I/O stack.

Figure 3.6 illustrates the flow of I/O issue and completion from the host to the controller, and back, in our design. Commands are transferred from the host to the controller, using PIO, in a circular FIFO queue. As shown in Section 4.2, PIO throughput is more than enough to transfer commands, whose related data are many orders of magnitude larger and the theoretical maximum data throughput exceeds the natural limits of the host-controller interconnects. Information about the scatter-gather lists of DMA segments is transferred along with the commands. As DMA segments become larger, they require less processing on the controller side (impact shown in Section 4.3). In order to use CRC32-C checksums per storage block, it is convenient to “chop” the scatter-gather list to 4KB segments, at the host side (usage is described in Section 3.7).

I/O completions are transferred back to the host via a FIFO queue that is mirrored between the controller and the host (shown in Figure 3.6). Completions are transferred to host memory via DMA transfers. They cannot be piggy-backed with the actual data transfer when using integrity checks, because the ADMA engines write back the checksums upon completion of the transfers. We must wait for transfers to finish to perform the integrity checks, and then complete the associated I/O operation (or deal with data corruption). Completions can also be transferred via controller initiated PIO. However, this demands a lot of controller CPU cycles, which proves to be a scarce resource in our platform.

The FIFO queues are statically allocated, and consist of fixed-size elements. Completions, commands and related control information may consume multiple consecutive queue elements of the FIFO queues.

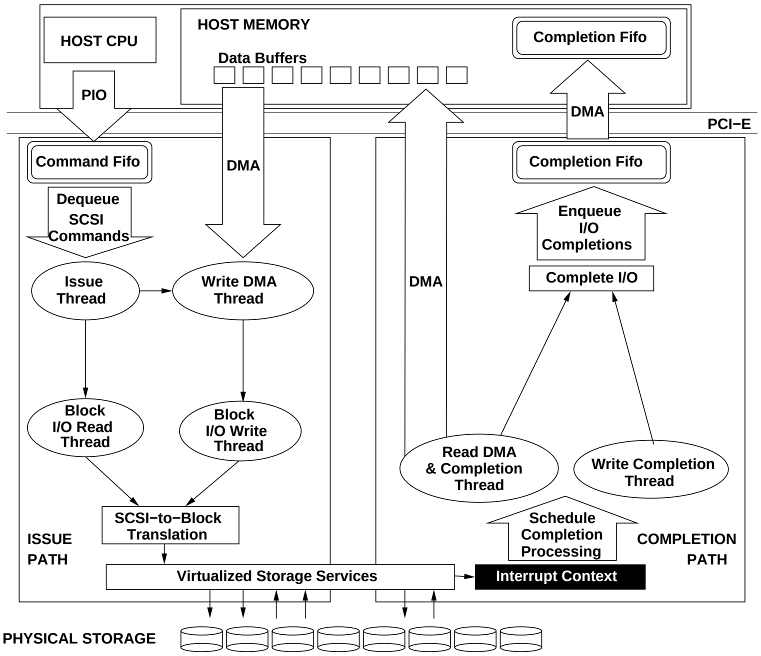


FIGURE 3.5: I/O issue and completion path in *DARC* without CRC support.

3.5 Buffer Management

Data buffers and associated information must remain resident in the controller memory until the I/O operation has been handled by the controller software. At high data rates the number of these buffers increases and so does the overhead of allocation or deallocation operations. Our experience has shown that it becomes important to guarantee constant overheads for buffer allocation and deallocation. This motivates the use of a memory pool with pre-allocated buffers. We also use an ”offsetting” technique to associate many objects with only a few allocations, reducing the memory allocation overhead. Thus we can coalesce related pieces of information using a record-like layout in only one buffer.

The buffers allocated for the I/O commands and their completions are different in one important respect from other buffers used by the controller: Command buffers are allocated on the controller, but are filled in by the host. Likewise, completion buffers are allocated on the host, but they are filled in by the controller. The entity that needs to fill in these buffers has to rely on point-to-point communication to find out their addresses. To reduce communication to a minimum, we use circular FIFO queues to hold commands and completions, because just sending the head and tail pointers is adequate. This data structure also guarantees constant overhead for allocations and deallocations. We avoid the need to transmit the tail pointers alltogether, by using the last bit of each FIFO element as a validity flag. This way, the controller or the host can poll its local memory to find out if there are any new commands or completions respectively.

In our design the information of SCSI or block-layer I/O descriptors remains in controller memory. The life-cycle of this information is the same as

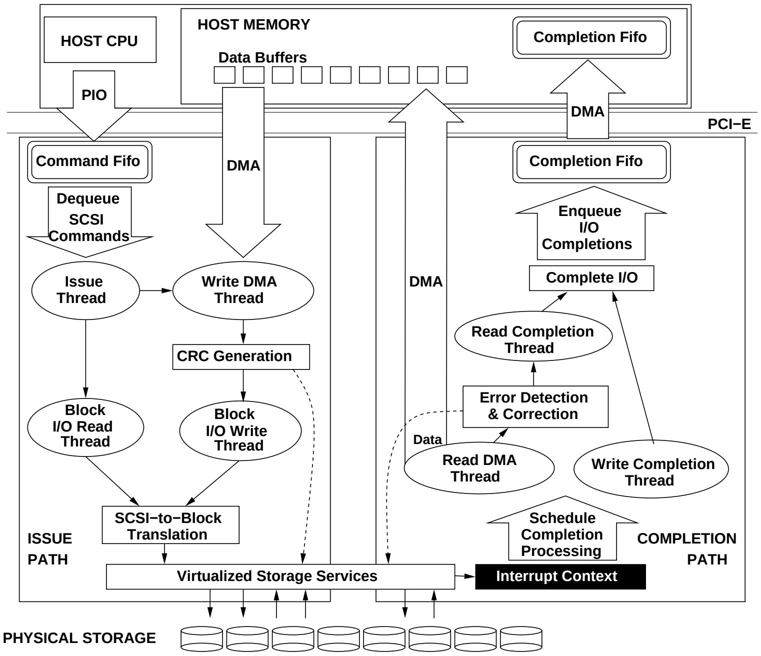


FIGURE 3.6: I/O issue and completion path in *DARC* with CRC support.

that of data buffers, which also need to be allocated efficiently. We have used a buffer allocator that maintains a pool of pre-allocated fixed-size objects. Individual pieces of information are positioned in these buffers at data-dependent offsets. Buffers can be deallocated either lazily or immediately when there is memory pressure. We show the impact of common allocators versus fixed-size pre-allocated buffers in Section 4.3. We use circular FIFO queues for both the SCSI commands and their completions (Figure 3.6).

3.6 Context Scheduling

To achieve high performance in a controller system, we must ensure that every hardware resource is given useful work to perform with minimal delay, along the issue and completion paths of I/O request processing (Figure 3.6). One must identify all the points in time, along both paths, where waits may occur, and eliminate them. Two specific examples are when the host is issuing a write in the issue path, and when the controller is completing a read in the completion path. When a write command is processed in the issue path, the controller initiates a DMA operation to transfer, from the host, the data to be written to the storage devices. Before the controller can issue the I/O operation, it must wait for the completion of this DMA transfer. Likewise, when the controller has to complete a read command, it must first make sure that the data have been transferred from its local memory to the host.

To deal with such issues we break the paths into *stages*, which can be states in a Finite State Machine (FSM), or threads in the operating system. FSMs are hard to maintain as functionality is added to the design, so we choose a thread-based design. This is more flexible as far as scheduling is concerned and provides a natural way to map processing onto multiple CPU cores. Having identified points where there is potential for waits, we break the I/O issue and completion paths into stages. If there are available CPU cores to spare, each stage can be assigned to a core. If, however, the stages are more than the cores, whenever it becomes necessary to wait while executing a stage, we suspend this stage and perform useful work in other stages or paths. An important issue is that interrupt contexts cannot be preempted. Therefore, it is necessary to schedule work out of interrupt-context as soon as possible, so that the interrupt context does not trigger any waiting periods. This leads to a design that maximizes the overlap of stages for multiple concurrent I/O requests.

In our design we perform all I/O processing operations in kernel-thread context, as opposed to interrupt context. We explicitly arrange the order of execution of stages whenever we identify a wait, because there is only one Application core in our platform. Threads in *DARC*, besides context switching when blocking on Linux events, voluntarily ”yield” to one another at explicitly-managed blocking conditions, e.g host DMA completions. This improves performance significantly compared to the default non-preemptive Linux kernel thread scheduling. Context switch cost (on the order of 1.1 microseconds) between stages is mitigated by batching at various levels.

We evaluated a non-preemptive priority-based scheduling policy (`SCHED_FIFO`, directly supported by the Linux kernel), motivated by the observation that this policy fits our design, where threads voluntarily yield the CPU to one another. This policy leads to a noticeably lower context-switch cost (on the order of 0.4 microseconds). Preliminary results indicate that with RAID-0 we can achieve the maximum I/O throughput level for 8 SAS drives, as opposed to 85% with the default fair scheduling policy (see Section 4.2.1), but IOPS performance is not significantly improved. However, with RAID-10 we have seen no significant performance gains. Based on these observations, in this thesis we present results using the default fair-scheduler.

3.7 Error Detection and Correction

Our approach for correcting errors is based on a combination of two mechanisms: (i) error detection through the computation of data checksums, which are persistently stored and checked on every read, and (ii) error correction through data reconstruction using available data redundancy schemes.

Error detection and correction (EDC) in *DARC* operates through the use of redundancy schemes (e.g. RAID-1), where the corrupted data are reconstructed using the redundant data block copies or parity. Redundancy schemes, such as RAID-1, 5, or 6, are commonly used for availability and reliability purposes in all installed storage systems that need to tolerate disk failures. Our system uses the same redundant data blocks to correct silent data errors, based on the assumption that the probability of data corruption on both mirrored copies of one disk block are very small. A similar assumption is made for other redundancy schemes, such as RAID-5/6, but the probability of a valid data reconstruction increases with the amount of redundancy maintained. The probability of a second data error occurring within the same stripe group (typically, 4-5 data blocks and 1-2 parity blocks) in RAID-5/6 is higher than for the two data block copies in RAID-10.

Existing controller designs have considered non-persistent checksum compu-

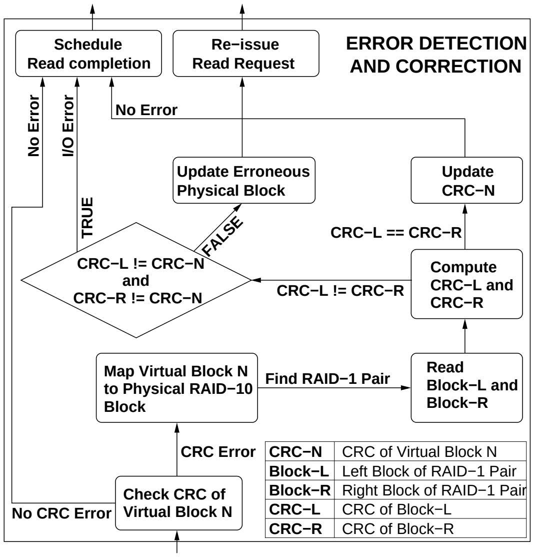

ERROR DETECTION AND CORRECTION
Schedule Read completion
Re−issue Read Request
No Error
I/O Error
No Error
Update Erroneous Physical Block
Update CRC−N
TRUE
FALSE
CRC−L == CRC−R
CRC−L != CRC−N and CRC−R != CRC−N
CRC−L != CRC−R
Compute CRC−L and CRC−R
Map Virtual Block N to Physical RAID−10 Block
Find RAID−1 Pair
Read Block−L and Block−R
CRC Error
Check CRC of Virtual Block N
No CRC Error
CRC−N CRC of Virtual Block N
Block−L Left Block of RAID−1 Pair
Block−R Right Block of RAID−1 Pair
CRC−L CRC of Block−L
CRC−R CRC of Block−R

Figure 3.7: Error correction procedure in the controller I/O path (Figure 3.6).

tation for detecting errors caused by faulty controller hardware (e.g. corruption due to faulty memory), or through faulty software, as the data pass through the I/O path. In such cases the checksum computation and check needs to be performed when the data are transferred to/from the host to the controller. Thus, current controller architectures have placed hardware resources for high-speed checksum (CRC) computation in the DMA engines of I/O controllers. Detecting and correcting data errors occurring in the storage devices themselves requires the low-overhead computation and management of checksums in the controller's memory. However, the checksum capabilities of DMA engines in our platform incur negligible overhead only when used during the DMA data transfers to/from the host. Thus, we have used checksums to protect host-accessible data blocks only. To maintain high performance we have not used EDC for metadata and data generated on the controller itself, such as versioning metadata, or RAID superblocks. More efficient hardware designs that allow for high-speed checksum computation in the controller's memory would minimize overheads for all data and metadata stored on the disks.

An advantage of only protecting host-accessible data blocks is that, RAID reconstruction, in the event of *device failure*, is unaware of the existence of persistent checksums. Therefore, RAID reconstruction need not be modified. Moreover, the RAID reconstruction procedure could take advantage of the data structure storing the checksums to determine which blocks can be safely skipped, as they have not been written so far. This would reduce the RAID reconstruction time.

For the evaluation reported in this thesis we have built a module stack that combines RAID-10, versioning, and CRC-checking functionality. Figure 3.6 shows the I/O path. CRC computations are performed by the ADMA channels, while transferring data to/from the host. The EDC layer persistently stores a 32-bit checksum for each 4KB block written to the storage volumes of the controller. The computed checksums are stored along with the corresponding block addresses on the disks as persistent metadata, and they are retrieved for checking when completing read I/O requests. The checksums are stored within a dedicated metadata block range that is striped across all the available storage devices. If there is a checksum mismatch, the error correcting module uses the replicas of the RAID-10 layer and the stored checksum to determine which block contains the correct data value. If both RAID-10 blocks are identical, then both blocks are assumed valid, the stored checksum is assumed erroneous, and is updated. In the case of one corrupted data block, the EDC module corrects the error on the disk by issuing a synchronous write, and also returns the correct data to the host, by re-issuing the I/O read operation. The error correction algorithm is shown in Figure 3.7, and the associated overheads in Section 4.6.

3.8 Storage Virtualization

DARC uses an existing kernel-level, virtualization framework, Violin [7], that augments the current Linux block-level I/O stack. Violin allows for (i) easy development of extension modules implementing new virtualization mechanisms (e.g. versioning or EDC) and (ii) combining of these modules to create modular hierarchies with advanced functionality. The combined modules essentially create a stack of virtual devices, each of which has full access to both the request and completion paths of I/Os. Violin supports asynchronous I/O, which is important for performance reasons, but also raises significant challenges when implemented in real systems. Also, the framework deals with metadata persistence for the full storage hierarchy, off-loading the related complexity from individual virtualization modules. As shown in Figure 3.4, Violin is located right above the SCSI drivers in the controller. Violin already provides versioning [6] and RAID modules.

In this work we design one new virtualization module within Violin, EDC, that offers persistent CRC32-C checksums per 4KB block. We also examine the implications of using versioning at the controller level as opposed to the host. Finally our EDC module requires modifications to RAID modules, to support recovery after a failed CRC check.

3.9 On-Board Cache Considerations

Typically, I/O controllers dedicate most of their on-board memory to caching and pre-fetching purposes. Caching of data blocks improves performance for I/O patterns that exhibit locality. Write-caching in particular is also beneficial in the implementation of RAID-5/6 volumes, since it hides the costs of the extra reads and writes that are incurred as a result of write requests with sizes smaller than the RAID chunk size. Pre-fetching aims to improve performance for sequential I/O patterns. *DARC* currently lacks these caching mechanisms, leaving all the on-board memory available for use by our custom buffer allocators, and for storing metadata objects of the block-level virtualization modules.

3.10 Summary of Design Choices

Table 3.1 summarizes our design choices for *DARC*.

Table 3.1: Design choices in *DARC*.

Challenge	**Decision**
Host-controller I/O path	PIO for commands, DMA for data and completions
Buffer Management	Fixed-size pre-allocated buffer pools, lazy de-allocation, static circular FIFO queues for commands and completions with fixed-size elements
Context Scheduling	Map I/O path stages to threads, explicit scheduling, no processing in IRQ context
Error Detection & Correction	CRC32-C checksums computed by DMA engines for 4KB blocks, persistently stored on dedicated meta-data space
Storage Virtualization	Violin block-level virtualization framework: supports RAID-10 volume management versioning and EDC modules
Data-block cache	On-board cache omitted in *DARC*

3.11 Platform-specific implementation details

The platform iop348 supports hardware assisted programmable FIFO queues. However, the queue element size is 4 bytes, which increases the management overheads considerably due to the fact that our elements need to be at least 50 bytes each. For this reason, we have implemented our FIFO queues entirely in software, and have mapped the queues' space directly onto the controller and host memory for the command FIFO and the completion FIFO respectively.

The DMA engines of iop348 (Figure 3.1) use memory resident DMA descriptors, that can be chained statically or dynamically. They engines can also be queried whether they are idle or running, and what the last/current DMA descriptor was/is. Finally, they are able to generate interrupts per completion of transfer. However, this amount of information is not enough to provide a simple implementation where one could examine whether a transfer has finished at any given time.

For this reason, we have designed a ring buffer of custom DMA transfer descriptors. Each custom descriptor is a concatenation of the transfer descriptor index in the ring buffer with an epoch number. Every time the ring buffer wraps around the epoch number is incremented by 1. This way, given any custom descriptor we can deduce whether the respective transfer has finished or not, without the need to allocate additional space for old transfers.

The real DMA engine descriptors are placed in uncacheable but bufferable memory pages, because we need to write them fast in a consistent manner with respect to the DMA hardware. The completion FIFO queue resides in uncacheable but bufferable memory pages for the same reason. All the controller registers, and controller generated PIO reside in uncacheable and unbufferable memory ranges. The registers cannot be mapped in any other type of memory, to evade the sideffects of caching and buffering. However, controller PIO would benefit from being placed in bufferable memory space, but this is not supported by the specified Linux kernel. The command FIFO queue is placed in cacheable memory, and we issue explicit invalidations and write-backs to synchronize with the ATU and DMA hardware. The reason is that we need fast read and write access, and this can only be provided by the CPU cache.

Chapter 4

Experimental Evaluation

We center our evaluation around achieving high I/O performance, the impact of the data protection mechanisms and the effort required to recover from faults.

4.1 Base Performance

We present evaluation results for a storage system built using the following components: 8 SAS disk drives (Seagate Cheetah 15K.5 ST374355SS, 15K RPM, 73.4 GBytes), the *DARC* controller, and a Tyan S5397 motherboard with two 4-core Intel Xeon 5400 processors running at 2GHz, and 4Gbytes of DDR-II DRAM. The host OS is Microsoft Windows Server 2003 R2 Standard Edition (32-bit). We compare our prototype with a commercially available I/O controller based on the same hardware components: an Areca ARC-1680D-IX-12 SAS controller based on the Intel 81348 SoC, running at 1.2 GHz, with 1 GByte of on-board DRAM, and an 8-lane PCI-Express host connection. The 8 disks are configured either as RAID-0 (only for the synthetic I/O patterns) or as a RAID-10, with a chunk size of 64KB. Each disk is capable of up to 250 small-sized random IOPS, and up to 125 MB/sec read transfers (115 MB/sec for writes).

4.2 DMA and PIO Performance

DARC is capable of transferring data to and from the host via DMA at rates exceeding 1 GByte/sec, for transfer sizes of 4KB (1 page) and higher. This is shown in Figure 4.1, with the curves marked "DMA". These measurements are obtained by initiating a sequence of transfers by one of the Application DMA channels, using the same API used by our firmware. I/O write operations originating from the host entail DMA transfers of one or more pages from the host's memory to the controller, where the maximum DMA throughput is 1.4 GBytes/sec. I/O read operations entail DMA transfers from the controller to the host's memory, where the maximum DMA throughput is 1.6 GBytes/sec.

In terms of programmed-I/O (PIO) operations, the *DARC* board is capable of sustaining on the order of 10 million 32-bit transfers per second (host-to-board). This measurement justifies our design decision to use PIO for transferring I/O command descriptors: With the descriptor's size equal to 64 bytes, PIO transfer allows up to 625K descriptors per second. One descriptor corresponds to a 4KB I/O request. Descriptors can be chained, in the case of larger I/O request sizes; for 64KB I/O requests, we need 6 descriptors to be transferred,

Table 4.1: IOP348 Microbenchmarks.

IOP348 clock cycle	**0.833 nsec** (1.2 GHz)	
Interrupt delay, Context Switch	837 nsec	1004.8 cycles
Memory store	99 nsec	118 cycles
Local-bus store	30 nsec	36 cycles
Outbound store (PIO write, to host)	114 nsec	136 cycles
Outbound load (PIO read, from host)	674 nsec	809 cycles
Outbound load with DMA transfers	3390 nsec	4069.6 cycles
Outbound load with DMA transfers and inbound PIO-Writes from host	5970 nsec	7166.8 cycles

therefore PIO allows issuing over 100K such I/O requests per second.

However, host issued PIO has significant impact on the controller DMA performance. When the controller performs 8KB DMA transfers without PIO interference, the rate achieved is about 1.3 GBytes/sec for the host to controller direction, 1.62 GBytes/sec for the controller to host direction and 2 GBytes/sec for both directions at the same time. On the other hand, when the host interferes with consecutive PIO stores, these numbers drop to 780 MBytes/sec, 1.59 GBytes/sec and 1 GByte/sec respectively.

In more detail, the host clock cycle is 0.5 nsec (2.0 GHz), and the host-initiated PIO write takes 100 nsec (200 cycles) to complete. The controller side microbenchmarks can be seen in Table 4.1. We notice that the store and load performance is also affected by DMA transfer activity.

4.2.1 Synthetic I/O Patterns

We use the `iometer` workload generator [16], with five I/O pattern specifications:

- `WS - Write Stream`: sequential 64KB writes.

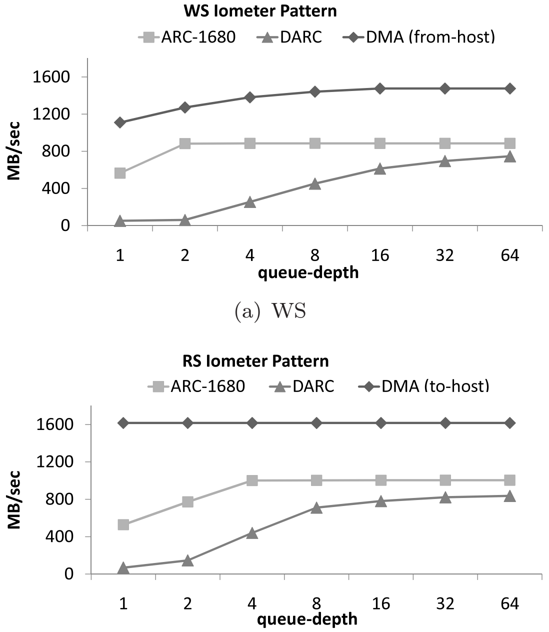
- `RS - Read Stream`: sequential 64KB reads.

- `OLTP - Online Transaction Processing`: random 4KB reads and writes, and a 33% write frequency.

- `FS - File Server`: random reads and writes, with a 20% write frequency and varying sizes: 80% of requests up to 4KB, 2% of requests for 8KB, 4% of requests for 16KB, 4% of requests for 32KB, 10% of requests for 64KB.

- `WEB - Web Server`: random reads, with varying sizes: 68% of requests up to 4KB, 15% of requests for 8KB, 2% of requests for 16KB, 6% of requests for 32KB, 7% of requests for 64KB, 1% of requests are 128KB, and 1% of requests for 512KB.

We obtain seven data-points for each of these I/O patterns, corresponding to issuing 1, 2, 4, 8, 16, 32, and 64 concurrent I/O requests. We first present results with a RAID-0 (striped, non-redundant) disk array, to illustrate the performance potential of the experimental platform. The RAID-0 results for the throughput-intensive and the IOPS-intensive access patterns are shown in Figures 4.1 and 4.3, respectively. To put the results in Figure 4.1 in perspective, the maximum read I/O throughput obtainable from the 8 attached disks is 1 GByte/sec (8 x 125 MB/sec), while the maximum obtainable write I/O throughput is 920



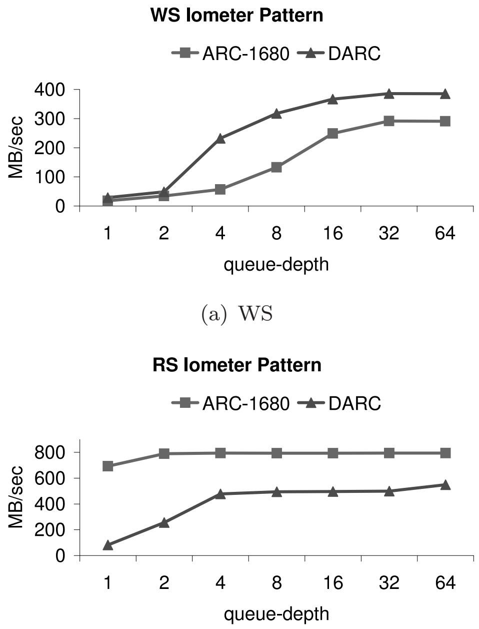
(a) WS

(b) RS

Figure 4.1: I/O throughput for sequential 64KB reads and writes with 8 disks in RAID-0.

MB/sec (8 x 115 MB/sec). DMA throughput is well above these device-imposed limits. The corresponding results with a RAID-10 (striped, redundant) disk array are shown in Figures 4.2 and 4.4, respectively. Each data-point comes from a 5-minute run, after a 1-minute warm-up interval. The WS and RS I/O patterns serve to characterize the throughput potential of the storage systems that we measure. The OLTP, FS and WEB I/O patterns serve to characterize the IOPS capability of the storage systems that we measure. The OLTP pattern stresses the capability for serving small-sized random I/O requests with latency constraints. The FS and WEB patterns include large-sized I/O requests as well, pressing the storage systems even further to keep up with multiple concurrent I/O requests. The OLTP, FS and WEB I/O patterns exhibit no locality, therefore we expect that the ARC-1680 controller will not obtain significant performance gains from caching data in its on-board memory. However, for the RS and WS I/O patterns, we expect ARC-1680 to obtain performance gains due to read-ahead and write-caching, respectively.

In terms of sequential I/O throughput, we observe that the ARC-1680 controller is able to reach the maximum I/O throughput of the attached 8 disks, in the RAID-0 experiments: 1 GByte/sec for the RS access pattern, and 885 MB/sec for the WS pattern. The *DARC* controller achieves 85% of this performance level, with several outstanding I/O requests. In the RAID-10 experiments, we again observe better read I/O throughput for the ARC-1680 controller (794 Mb/sec), as compared to the *DARC* prototype (549 MB/sec). However, the *DARC* prototype outperforms the ARC-1680 controller for RAID-10 se-



(a) WS

(b) RS

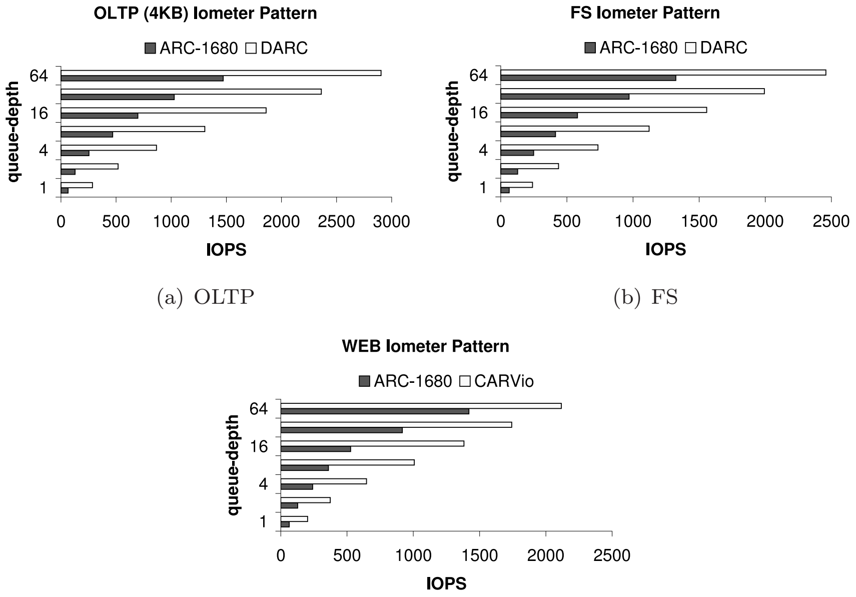
FIGURE 4.2: I/O throughput for sequential 64KB reads and writes with 8 disks in RAID-10.

quential write throughput, achieving 385 MB/sec, as opposed to 290 MB/sec. We believe that this points to a possible problem in the ARC-1680 controller's firmware. In terms of IOPS performance, the *DARC* prototype consistently outperforms the ARC-1680 for all the tested I/O access patterns and queue-depths, for both RAID-0 and RAID-10. The ARC-1680 controller delivers around 67% of the IOPS performance of *DARC* (only 50% in the case of the `OLTP` and `FS` access patterns with RAID-0). Therefore, we conclude that our I/O path is indeed capable of low-latency operation.

4.3 Impact of Allocator and DMA Segments

We use iometer with the `WS` and `RS` access patterns, for a RAID-0 disk array, to illustrate the performance impact of buffer allocation. As shown in Figure 4.5, with the curves marked `DFLT-ALLOC`, the default memory allocator results in throughput collapse for large numbers of concurrent read I/O requests (up to 64%). Performance for write I/O requests remains unchanged. We see the need for comprehensive tuning in the I/O path, as the seemingly simple change of the buffer allocator results in a 177% throughput improvement at high I/O rates (queue-depth higher than 16).

We also explore the impact of the DMA segment sizes generated by the host. Having observed that the host generates predominantly 4KB DMA segments, we have built a version of the host driver that allocates 64KB DMA segments for I/O requests. This means that for the `WS` access pattern only one DMA segment is transferred to/from the host, instead of 16 4KB segments as is the



(a) OLTP

(b) FS

(c) WEB

FIGURE 4.3: IOPS performance for OLTP, FS, and WEB access patterns with 8 disks in RAID-0.

TABLE 4.2: Server workload parameters.

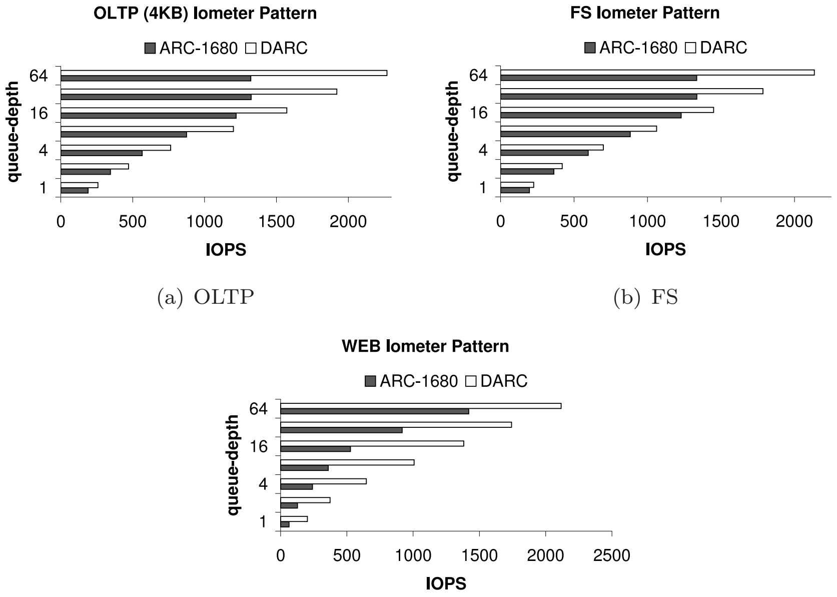
TPC-H	Dataset size: 4 GBytes (≈ 6.5 GBytes on disk)
JetStress	1000 mailboxes of 100 MBytes each, 1.0 IOPS per mailbox Operation Mix: 25% insert, 10% delete, 50% replace, 15% read.

normal case. As shown in Figure 4.5, with the curves marked `LARGE-SG`, with large DMA segments the *DARC* matches the read I/O throughput of ARC-1680, even though it lacks the read-ahead capability of ARC-1680. Large DMA segments also lead to performance gains for write I/O requests.

4.4 Server Workloads

We quantify the impact of the data protection mechanisms using results from the TPC-H database benchmark [40]. We expect the ARC-1680 controller to be able to serve several read I/O requests from its on-board cache, as well as to absorb small writes in its write-back cache. Our *DARC* controller has no data-block cache, so all I/O requests are served from the attached disks. We also present results from running the `JetStress` benchmark (v. 08.01.0177), which emulates the I/O patterns generated by the Microsoft Exchange Server [23]. All the results in this section were obtained with the storage system described in the previous section, with the 8-disk RAID-10 configuration.

The key parameters of each of the benchmarks are summarized in Table 4.2. For the TPC-H experiments, we execute a sequence consisting of ten queries:



(a) OLTP

(b) FS

(c) WEB

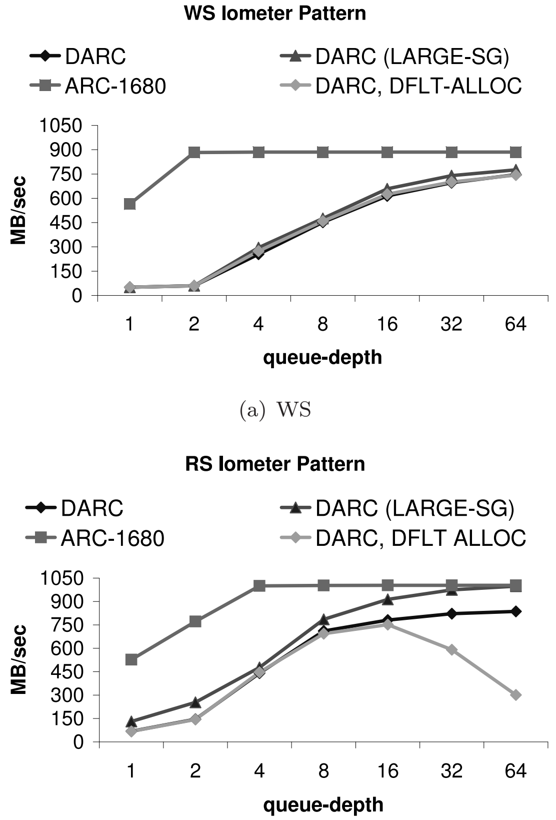
Figure 4.4: IOPS performance for OLTP, FS, and WEB access patterns with 8 disks in RAID-10.

1, 3, 5, 6, 7, 8, 11, 12, 14, and 19. The database management system used is MySQL Server, v.5.1 running on top of an NTFS filesystem. Running on top of a filesystem, we observe a number of additional I/O requests, both reads and writes, that are not directly generated by the benchmark codes; however, this additional I/O load is a very small percentage of the total I/O traffic.

The configuration marked "*DARC*, NO-EDC" corresponds to our controller with the 8 disks configured as a RAID-10 array. The configuration marked "*DARC*, EDC" adds the error detection and correction layer, with CRC32-C values computed and checked for all I/O accesses, but no versions captured automatically. Finally, the configuration marked "*DARC*, EDC, VERSION" extends the previous configuration by capturing volume versions automatically every 60 seconds. In this last configuration the version garbage collector is also running, purging previous versions. Our configurations are compared with the ARC-1680 controller. Unless otherwise indicated, all ARC-1680 results are with the read-ahead and write-back features enabled.

4.5 Impact of Data Protection Mechanisms

Figure 4.6 summarizes results from executing the TPC-H benchmark. The reported metric is the total execution time for the 10-query sequence. This workload only issues read I/O requests, but since it runs on top of a filesystem we also see a number of small write I/O requests. Taking the execution time with the ARC-1680 as the baseline, the figure shows the relative change in the response time. Even without an on-board cache, *DARC* matches the performance of ARC-1680. With EDC checking enabled, we observe an increase of



(a) WS

(b) RS

Figure 4.5: Impact of buffer allocator and DMA segment sizes: Sequential I/O Throughput with 8 disks in RAID-0.

the overall execution time by around 12%. Periodic version capture and purging add an additional 2.5% overhead for this workload. Note that in TPC-H successive volume versions are very small, as the 10-query sequence executed does not result in database updates.

Table 4.3 summarizes the results from JetStress. JetStress reports the I/O rate (IOPS) for the data and log volumes it uses. We observe that the *DARC* configurations achieve much fewer concurrent I/O operations on the log volume, as compared to the ARC-1680 configuration. The average size of a write request to the log volume is 2.9KB, and without an on-board write-back cache we cannot batch these requests, despite them being mostly sequential. To verify this, we repeat the JetStress experiment for ARC-1680, with the write-back feature disabled. We find that the number of IOPS on the log volume drops dramatically.

Focusing on READ IOPS for the data volume, the overhead induced by the data protection mechanisms is around 20% for EDC checks, and an additional 5% for the periodic version capture and purge. The higher overhead for EDC checks, as compared with previous results, is due to the much higher number of

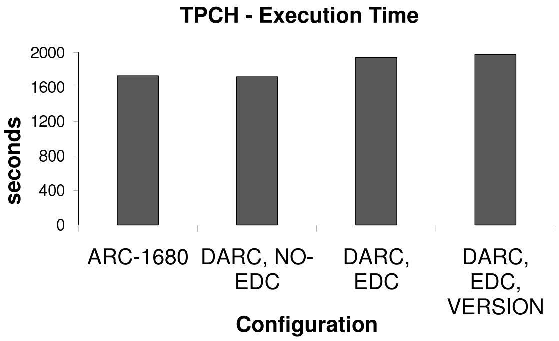


FIGURE 4.6: Impact of data protection on TPC-H.

TABLE 4.3: JetStress results (IOPS).

Configuration	**IOPS data-volume**		**IOPS log-volume**
	READ	WRITE	
***DARC*, NO-EDC**	679.32	556.79	70.32
***DARC*, EDC**	544.41	456.17	54.2
***DARC*, EDC, VERSION**	516.43	434.65	53.95
ARC-1680, write-back	774.94	703.1	626.4
ARC-1680, write-through	505.32	425.93	39.99

concurrent read I/O requests. Volume versions in these experiments are quite large, as JetStress generates a large number of write requests. The parameters in the JetStress experiments set a performance goal of at least 1000 IOPS for the data volume. As seen by summing the READ and WRITE IOPS columns in Table 4.3, the base *DARC* configuration achieves this performance goal, with 1236 IOPS, as opposed to 1478 IOPS for the ARC-1680 configuration with write-back (but only 931 with write-through). With EDC checks enabled, we barely achieve 1000 IOPS for the data volume, whereas with periodic version capture and purge the performance goal is missed, with a score of 951 IOPS.

4.6 Cost of Data Recovery

Next, we quantify the cost of the data recovery procedure described in Section 3.7. This data recovery procedure involves the execution of a number of synchronous I/O accesses to determine if the data corruption fault is isolated to one of the disks in the RAID-10 array, and if so, to restore a consistent state (data blocks that match the stored checksum). If this is achieved, then the original read I/O request is re-issued (expecting the resulting data to pass the CRC32-C check). Table 4.4 summarizes the operations involved in each of the cases shown in Figure 3.7. Checksum checks in the controller's I/O path are able to detect data corruption faults and under certain conditions recover from them, without the host being affected. The cost of the recovery procedure is the time to execute three I/O operations and two CRC computations, all for 4KB data blocks. These operations need to be synchronous to maintain the

Table 4.4: Recovery costs (operations).

Case	**data I/Os**	**CRC I/Os**	**CRCs calc'ed**	**Outcome**
RAID-1 pair data differ, CRC matches one block	3	0	2	Data Recovered, Re-Issue
RAID-1 pair data same, CRC does not match	2	1	2	CRC Recovered
RAID-1 pair data differ, CRC does not match	2	0	2	Data Error, Alert Host

storage volume and its associated checksums in a consistent state; therefore, in the event of a data corruption fault the host will suffer a noticeable performance penalty. If recovery is not possible, the data corruption fault is propagated to the host. In this event, there is the option to initiate rollback of the affected storage volume to a previously captured version.

Chapter 5

Conclusions

In this thesis we address the question of how to incorporate data protection features in a commodity I/O controller, in particular integrity protection using persistent checksums and versioning of storage volumes at the data-block level. We identify the key challenges in implementing an efficient I/O path between the host machine and the controller, and present a prototype implementation using real hardware. The performance of our prototype is competitive to the performance of a commercially available I/O controller based on comparable hardware components, while offering substantial support for data protection.

Using the performance of the base *DARC* configuration as the point of reference, both in terms of I/O throughput and IOPS, we find the performance overhead of EDC checking to be between 12% and 20%, depending on the number of concurrent I/O requests. This overhead is affected by the decision to keep checksum at the granularity of 4KB data blocks. Version capture and purging add an additional performance penalty between 2.5% and 5%, depending on the number and total size of I/O write requests between versions. We expect this overhead to become much less noticeable if the capture period is increased, and, more importantly, if versions are purged less frequently.

Overall, our work shows that I/O controllers are not so much limited from host connectivity capabilities, but from internal resources and their allocation and management policies. More specifically, we believe:

- CPU capacity is the most important limitation in I/O controllers at high I/O rates. Even worse, although the main design assumption is that the I/O controller's CPU does not "touch" the data, we expect this to change as more advanced data I/O features are incorporated in the I/O path. For this reason it makes sense to offload work from the controller CPU on every opportunity. We believe there will be a growing need for acceleration of operations on data in-flight.

- Understanding data-path intricacies is essential to achieve high I/O rates. I/O controllers offer specialized data-paths and special-purpose hardware units, such as DMA engines, messaging and address translation units that need to be carefully co-ordinated for efficient I/O processing. It is essential to overlap transfers as efficiently as possible, not only to/from the host but also to/from the attached storage devices.

- I/O controllers will need to handle increasing volumes of persistent metadata, to be used along the I/O path. Handling such metadata will consume even more of the controller's CPU and memory.

Finally, through the evolution of our prototype, we have come to realize that under high I/O rates subtle design decisions have substantial impact. Although certain design details may at first seem rather obvious, we find they play a significant role in the overall system performance. We believe that the analysis we provide of their combined impact on end-to-end application-perceived performance using common server workloads, is not only useful today, but also valuable for designing future storage controller architectures.

5.1 Future Work

We are now working on integrating a user I/O cache in the I/O path. There are many choices one can make. It can:

- be either direct-mapped or set associative.
- be a block cache or have larger than block cache lines.
- perform read-ahead and access pattern prediction.
- be write-allocate or write-no-allocate.
- write-back or write-through.
- have various replacement policies.

Preliminary results of a direct mapped, write-back, block I/O cache show that maximum DMA performance is matched for 100% hit-ratio.

Bibliography

[1] T10 DIF (Data Integrity Field) standard. http://www.t10.org.

[2] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *Proceedings of the 8th ASPLOS*, pages 81–91, San Jose, California, Oct. 1998.

[3] M. K. Aguilera, S. Spence, and A. Veitch. Olive: Distributed point-in-time branching storage for real systems. In *In Proc. Third NSDI*, pages 367–380, 2006.

[4] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *Proc. of the 6th USENIX Conf. on File and Storage Technologies (FAST'08)*, pages 223–238, 2008.

[5] J. Bonwick and B. Moore. ZFS: the last word in file systems. http://www.sun.com/software/solaris/zfs_lc_preso.pdf.

[6] M. D. Flouris and A. Bilas. Clotho: transparent data versioning at the block i/o level. In *Proceedings of 12th IEEE/NASA Goddard (MSST2004) Conference on Mass Storage Systems and Technologies*, pages 315–328, 2004.

[7] M. D. Flouris and A. Bilas. Violin: A framework for extensible block-level storage. In *Proceedings of 13th IEEE/NASA Goddard (MSST2005) Conference on Mass Storage Systems and Technologies*, pages 128–142, Monterey, CA, Apr. 2005.

[8] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. of the 8th ASPLOS Conference*. ACM Press, Oct. 1998.

[9] J. S. Glider, C. F. Fuente, and W. J. Scales. The software architecture of a san storage control system. *IBM Syst. J.*, 42(2):232–249, 2003.

[10] J. Gray. What next? A few remaining problems in information technology (Turing lecture). In *ACM Federated Computer Research Conferences (FCRC)*, May 1999.

[11] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 283–296, Stevenson, Washington, October 2007.

[12] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX '94 Winter Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[13] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. *SIGPLAN Not.*, 27(9):23–35, 1992.

[14] Intel. Intel 81348 I/O Processor: Developer's Manual. http://www.intel.com/design/iio/docs/315036.htm.

[15] Intel. Intel Xscale IOP Linux Kernel Patches. http://sourceforge.net/projects/xscaleiop/files/.

[16] Intel. IoMeter. http://www.iometer.org.

[17] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are disks the dominant contributor for storage failures? A comprehensive study of storage subsystem failure characteristics. In *Proc. of the 6th USENIX Conf. on File and Storage Technologies (FAST'08)*, pages 111–125, 2008.

[18] L. Jones, M. Reid, M. Unangst, and B. Welch. Panasas tiered parity architecture. Panasas White Paper, 2008.

[19] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity lost and parity regained. In *Proc. of the 6th USENIX Conf. on File and Storage Technologies (FAST'08)*, pages 127–141, 2008.

[20] G. Laden, P. Ta-Shma, E. Yaffe, M. Factor, and S. Fienblit. Architectures for controller based cdp. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies (FAST'07)*, pages 21–21, Berkeley, CA, USA, 2007. USENIX Association.

[21] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 84–93. ACM SIGARCH/SIGOPS/SIGPLAN, Oct. 1996.

[22] J. Menon and J. Cortney. The architecture of a fault-tolerant cached raid controller. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 76–87, New York, NY, USA, 1993. ACM.

[23] Microsoft. Optimizing Storage for Microsoft Exchange Server 2003. http://technet.microsoft.com/en-us/exchange/default.aspx.

[24] C.-H. Moh and B. Liskov. Timeline: a high performance archive for a distributed object store. In *NSDI*, pages 351–364, 2004.

[25] M. A. Olson. The design and implementation of the inversion file system. In *Proceedings of USENIX '93 Winter Technical Conference*, Jan. 1993.

[26] B. Panzer-Steindel. Data integrity. CERN/IT Internal Report, http://tinyurl.com/yqxdou, Apr. 2007.

[27] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from bell labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[28] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of the 5th USENIX Conf. on File and Storage Technologies (FAST'07)*, pages 2–2, Berkeley, CA, USA, 2007.

[29] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proc. of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005.

[30] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[31] W. D. Roome. 3DFS: A time-oriented file server. In *Proceedings of USENIX '92 Winter Technical Conference*, Jan. 1992.

[32] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGARCH Comput. Archit. News*, 32(5):48–58, 2004.

[33] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. *SIGOPS Oper. Syst. Rev.*, 33(5):110–123, 1999.

[34] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *Proc. of the 5th USENIX Conf. on File and Storage Technologies (FAST'07)*, pages 1–1, Berkeley, CA, USA, 2007.

[35] Sean Quinlan and Sean Dorward. Venti: a new approach to archival data storage. In *Proc. of the 1st USENIX Conf. on File and Storage Technologies (FAST'02)*, pages 89–102. USENIX, Jan. 28–30 2002.

[36] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[37] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: techniques and applications. In *Proceedings of the First ACM Workshop on Storage Security and Survivability (StorageSS 2005)*, pages 26–36, FairFax, VA, November 2005. ACM.

[38] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proc. of the 2nd USENIX Conf. on File and Storage Technologies (FAST'03)*, pages 43–58, Berkeley, CA, USA, 2003. USENIX Association.

[39] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 165–180, Berkeley, CA, 2000.

[40] Transaction Processing Performance Council. TPC Benchmark H (TPC-H). http://www.tpc.org/tpch/.

[41] K. Vijayasankar, G. Sivathanu, S. Sundararaman, and E. Zadok. Exploiting type-awareness in a self-recovering disk. In *Proceedings of the Third ACM Workshop on Storage Security and Survivability (StorageSS 2007)*, pages 25–30, Alexandria, VA, October 2007. ACM.

[42] J. Wilkes, R. A. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, Feb. 1996.

[43] J. Wires and M. J. Feeley. Secure file system versioning at the block level. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 203–215, New York, NY, USA, 2007. ACM.

[44] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proc. of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 273–288, December 6-8, 2004, San Francisco, California, USA. USENIX Association, 2004.