Microprocessors and Microsystems 52 (2017) 461–469

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Reliability and performance enhancements for SSD RAID

Alistair A. McEwan*, Muhammed Ziya Komsul

Department of Engineering, University of Leicester, LE1 7RH, UK

ARTICLE INFO

Article history:
Received 29 December 2015
Revised 7 October 2016
Accepted 14 November 2016
Available online 18 November 2016

Keywords:
SSD RAID
Real time storage
SSD garbage collection
SSD RAID architecture
Hotswapping and data migration
Parity distribution

ABSTRACT

NAND based solid state storage devices are almost ubiquitously used in safety-critical embedded devices, and recent advances have demonstrated RAID architectures specific to solid state storage devices resulting in increased data reliability, with architectural enhancements to solve the age convergence problem. However, these techniques require devices to be taken off-line while components are replaced—consequently these devices are of limited use in hard real time systems. There are further real time issues in that the conventional architectures ignore other characteristics of solid state devices such as garbage collection and meta data management. In this paper we investigate techniques that support the replacement of aged devices in the array in such a way that we provide continuous system reliability. We also improve the performance overhead of the reconstruction process using a novel data migration policy. The techniques are implemented and tested in a trace-driven simulator, and results demonstrate that average I/O response time is improved by up to 39% with improvement by up to 45% in its standard deviation, overheads in terms of device replacement time are negligible, and read performance is improved by an average of 8%.

1. Introduction

Many embedded systems, including those that are safety critical, have to observe strict constraints in terms of shock resistance, energy consumption, physical size, and other factors. Magnetic hard discs are typically not well suited to systems with these constraints due to their mechanical nature—however solid state storage (SSD), otherwise known as flash memory, enjoys many advantages. However, perhaps the single most important issue when using SSD devices in high reliability environments is the fact that they physically wear out over time—normally related to the number of times parts of the device are erased—meaning that data stored on the device will become unreliable over time. This failure characteristic is different to traditional magnetic mediums that typically fail non-deterministically.

One approach to enhance reliability in the case of wear-out is the use of Error Correction Codes (EEC). This is parity-style data that is stored in the meta data of each page of memory, and is used to cross-check, or repair, the data it relates to at the point where data is read. However there are limits to how much it can improve reliability as the potential to repair is limited. Where capacity of storage is an issue, multiple level cell (MLC) technology may offer benefits over single level cell (SLC) devices. However MLC devices suffer from greater unreliability—particularly when the device ages—than SLC due to a lower erase endurance. MLC also do not lend themselves to ECC techniques as the size of meta data areas is limited.

Recent advances in storage technology have applied Redundant Array of Independent Disk (RAID) to SSD storage in order to improve reliability and data integrity when a single device or component fails. Common RAID techniques such as RAID 4 and RAID 5 hold parity data to reconstruct original data in case of block errors. However, the usage of these techniques in SSD suffers the problem of wearing out all the devices simultaneously.

In previous work a novel RAID-based architecture was presented to enhance the reliability of an SSD storage system [1], mitigating this problem by guaranteeing wear imbalance between components in the array. This is done using two primary techniques. The first is an uneven parity distribution—which refers to ratios of the parity data across devices of the array—that ensures erases across components are distributed unevenly, and the second is a device copy/swap algorithm that moves data around and manages lifespan as components reach endurance limits. The limitation of this architecture is that this copy/swap operation requires the array to be taken off line and so whilst this mechanism significantly enhances reliability, it restricts is usage in hard real time systems as it is not able to serve requests during the component replacement period. Furthermore, it does not consider flash specific operations such as garbage collection and meta data manage-

* Corresponding author.
E-mail address: aam19@le.ac.uk (A.A. McEwan).

http://dx.doi.org/10.1016/j.micpro.2016.11.012
0141-9331/

ments, both of which may affect the real time characteristics of a system.

The contribution of this paper is an investigation into several novel techniques that may be incorporated into an SSD RAID that improve the efficiency of the replacement process for hard real time applications—proactive hot swapping, data migration that coordinates operations with a garbage collector, and a parity redistribution mechanism. To utilize the benefits of hot swapping, a semi hybrid RAID mechanism is also introduced that enhances performance when there is no active device replacement process.

The paper is structured as follows: in Section 2 we present motivation and background for this work. Section 3 presents the architectural design of the system. Section 4 presents proactive hot swapping, Section 5 presents data migration, and Section 6 presents the parity redistribution mechanism. Section 7 presents the semi hybrid RAID configuration. Experimentation and analysis are given in Section 8, and we draw some conclusions and identify areas of future work in Section 9.

2. Motivation and related work

Early works in the area of SSD RAID storage include [2], which explores the viability of using RAID architectures as a reliability enhancement. Age convergence mechanisms—the process by which parity is distributed and thereby component ageing managed—were further developed in [3]. The contribution of these works was to minimise risk of simultaneous device failure by unevenly distributed parity techniques. However these works do not take into consideration real time and performance related issues of flash arrays.

Mir and McEwan [4] describes the implementation of a flash management framework in synthesizable Verilog that was used in a series of experiments exploring out of order execution, dynamic scheduling, and multi chip parallelism. The framework was expanded in [5] to allow for simulation and experimentation in the case of real time issues.

The device replacement process was originally presented in [1], and limitations of this process with respect to real time systems were discussed in [6]. In summary, the replacement process involves changing the most aged component with a hot spare, whilst also ensuring that uneven parity distribution is maintained by redistributing it (using the parity distribution of [7]). This involves moving data and reconstructing parity on other devices. These operations necessarily increase write amplification and device replacement time—*write amplification* refers to the additional writes caused by operations such as garbage collection and wear levelling, and is formulated as the ratio of total writes performed to the writes requested by the host [8]. This presents limitations in real-time environments.

A number of studies aimed at providing on line and efficient RAID reconstruction have been conducted for magnetic hard disks, such as [9–12]. Although they offer on line replacement, these mechanisms considerably increase the amount of I/O to the storage systems and thus the average response time of the system grows, as shown in [11]. Existing SSD based replacement techniques either do not provide online replacement [2] or apply HDD based RAID reconstruction mechanisms [13,14].

Write amplification in RAID has been extensively discussed in literature. MiPiL, for instance, minimise data migration while maintaining uniform data and the parity distribution of RAID-5 [15]. Moreover, a diagonal coding scheme is introduced for system-level wear levelling which prevents rapid wear out due to updating dependencies between actual and parity data [16]. To create wear imbalance in the case of sequential workloads a forced random write approach with partial stripe is presented in [3]—however this approach increases write amplification while reducing lifespan. To address this an efficient lifetime management that prevents additional parity updates while creating age differentiation is given in [17]. These mechanisms only consider parity related write amplification, ignoring the high write amplification caused by device replacement and parity redistribution.

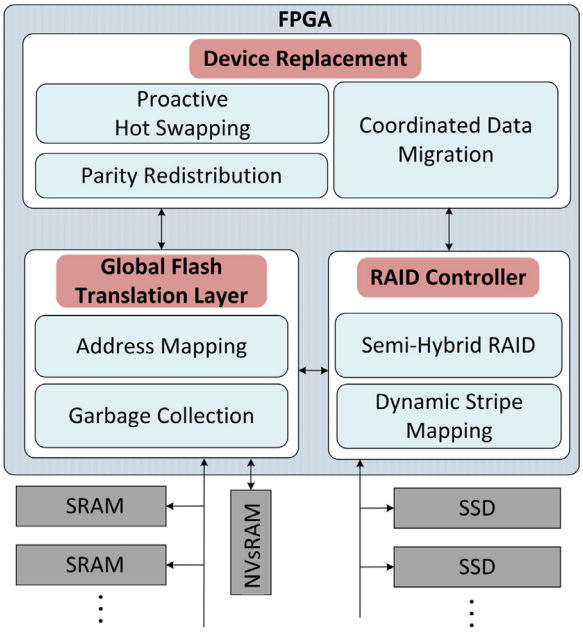

Fig. 1. System architecture block diagram.

There have been several techniques proposed to replace solid state devices in case of failure. Diff-RAID [2] provides a device replacement which shifts parity according to the next parity assignment—however a significant difference is that it applies a reconstruction method based on magnetic devices and therefore increases write amplification and device replacement time due to additional parity movement operations. To reduce to parity data overheads, a configurable RAID mechanism for SSDs is presented in [14]. The mechanism stores less important data using RAID 0 (which does not provide redundant data for recovery in case of failure) while more important data is stored using parity based RAID levels. Although this reduces performance overheads incurred from parity data, it does not provide a replacement policy for a complete device failure—only partial levels of data recovery.

Non-deterministic behaviours of NAND flash memory have been investigated in several works, including garbage collection confliction with I/O. These studies propose either partial cleaning policies with the help of additional memory [18,19], file system support [20], or pre-emptible garbage collection [21], but they do not consider the context of RAID.

To guarantee average I/O latency while migrating data, [22] presents a control-theoretic approach which dynamically adjusts the speed of data migration by periodically measuring I/O performance of the magnetic storage devices. Thus, it migrates the majority of data during idle time periods or low density streams [23]. presents an idle time detection method that achieves zero impact on the foreground application whilst rebuilding the RAID.

3. Architectural design

The architecture of the system and internal communication paths are illustrated as a block diagram in Fig. 1 and is based on [7]. It consists of the solid state devices, memory components to store meta data information, and the FPGA-based management components. The management component consists of 3

Data structure 1 Global view of a single device.

```
struct ssd_metadata {
  boolean GC;
  DR_type DR;
  integer parity_percent;
  integer erasures;
  integer freeblocks;
  integer hardthreshold;
  queue ioreq request_queue;
  sequence lpa_table;
  sequence page_status_table;
}
```

Data structure 2 Global view of RAID array.

```
struct RAID {
  RD_type raid_type;
  integer number_devices;
  integer threshold;
  boolean active_replacement;
  sequence ssd_metadata ssd_array;
  stripe_map table;
}
```

main blocks. The first block—Device Replacement—contains proactive hot swapping, coordinated data migration, and parity redistribution functionality. It monitors the Global Flash Translation Layer (FTL) in order to make decisions about activating or deactivating these functions.

The second block is the Global FTL as it takes a holistic view of the whole array, rather than a view of a specific SSD device—this is the view described by the pseudo code data structure in Data structure 1—an array of this structure is held globally, with one array entry per device and this is what enables the raising of various functional behaviours to a global level. For each device a record is kept as whether or not garbage collection is currently active, device replacement is ongoing, the percentage of parity to be stored on the device, the number of erasures performed, the number of free blocks remaining, the hard threshold for garbage collection, the queue of incoming I/O requests, a table of logical to physical addresses and the page status (valid data, invalid data, or free space). The Global FTL manages address mapping (logical to physical addresses) using these tables, and garbage collection functionality. To reduce the performance overhead of meta data operations, the FTL physically stores the page status tables in NvSRAM, and SRAM is used for physically storing address mapping tables and the statistical information for each individual device in the array.

The third block—the RAID controller—provides primary RAID functionality. The view of the overall RAID array is described by the pseudo code data structure in Data structure 2, and consists of a note of the type of the array (RAID-4, RAID-5, Diff RAID, or semi hybrid), the number of devices in the array (not counting spares), the erasure limit threshold for the devices in the array (we assume

Data structure 3 Stripe mapping table entry.

```
struct stripe_map {
  list devices;
  integer parity_device;
  integer logical_address;
}
```

Data structure 4 Device specific I/O request.

```
struct ioreq {
  integer blkno;
  integer time;
  request_type type;
  integer size;
  request_priority priority;
}
```

the devices are homogenous), a boolean variable indicating if the device replacement process is currently active, an array of metadata structures describing the state of each device in the array, and the stripe mapping table which contains details of all stripes of data stored in the system. This stripe mapping table is stored in SRAM memory. The semi hybrid controller dynamically reconfigures components in the array, and the global view of the array, after device replacement in order to improve read performance by manipulating this structure.

An entry in the stripe mapping table is described by the pseudo code data structure in Data structure 3 and consists of a list of the physical devices that the data in question is stored on, the device index where parity is stored, and the logical address of the data. It is only necessary to store a single logical address, as this refers to an entry in the $ssd_metadata$ address mapping table lpa_table where, for a given datum the logical address maps to the physical address where the datum is stored.

Each device maintains its own queue of incoming requests, described by the pseudo code in Data structure 4. A request consists of a logical block number that the request targets, arrival time, the type of request (which may be read, write, erase, or generated by garbage collection or device replacement), the size of the request, and the priority—which may be either high or low.

4. Proactive hot swapping

The proactive hot swapping mechanism presented in this section enables device replacement and array reconstruction to be carried out while the array remains on line before the given device reaches a critical bit error rate. When device replacement is needed, incoming write requests are redirected to a hot spare device at FTL level and the stripe mapping table in the FTL is updated. However reading from the failing device causes additional I/O, due to the need to read and recalculate using parity blocks.

A device can be considered in one of three states by the Global FTL. S_0 is a fully enabled state—the device is available for all I/O read/write activity. S_1 is a read only state where the device only responds to read instructions and does not serve any writes. S_2

is a disabled state where the device is not accessible for any I/O read/write activity. (Algorithm 1)

Algorithm 1 Managing proactive hot swapping.

```
1:  ssd_array[].state, spare ← S0, RAID.number_devices+1
2:  while true do
3:      repeat
4:          eldest ← greatest(ssd_array[].erasures)
5:      until eldest > RAID.threshold
6:      ssd_array.eldest.DR, active_replacement ← S1, true
7:      update(ssd_array[].parity_percent)
8:      while active_replacement do
9:      end while
10:     if RAID_raidtype = hybrid then
11:         ssd_array[eldest].DR, spare ← S1, spare+1
12:     else
13:         ssd_array[eldest].DR, spare ← S2 spare+1
14:     end if
15: end while
```

All devices in the array—including all future devices to be used are initialised to S_0 (Line 1) and the next spare device identified, before the hot swap manager enters the infinite loop that monitors the whole system (Line 2). The index of the most aged device is noted on each iteration of the loop that follows in the local variable *eldest*—which then terminates when one device exceeds the safety threshold (Line 3–Line 5). The hot swap manager then records that it is in the process of hot swapping the device by setting the *active_replacement* flag and the state of the device in question (Line 6)—this ensures that the FTL knows to migrate write operations from the old device to the new device. Parity limits are then redefined across the array for all the remaining devices and the new device (Line 7). The hot swap manager then busy waits until all data has been migrated—indicated by the FTL resetting the active replacement flag (when the cold data migration process reports that all data has been migrated) (Line 8, Line 9). If a hybrid RAID configuration has been selected (Line 10) then the replaced device can be retained for read purposes (Line 11), else it is discarded (Line 13), and in both cases the next spare device is identified.

Dynamic I/O location is the process by which I/O destination requests are determined during hot swapping and is described informally in Algorithm 2. Read operations that hit the old device

Algorithm 2 Dynamic I/O location.

```
1:  while true do
2:      spare ← RAID.number_devices+1
3:      eldest ← greatest(ssd_array.erasures)
4:      ? RAID.table[x].logical_address = request.blkno :
5:                          devices_needed ← RAID.table[x].devices
6:      if eldest ∈ needed_devices ∧ active_replacement ∧ re-
    quest.type = update then
7:          write(spare, request)
8:          invalidate(eldest, request)
9:      else
10:         Do I/O operations as normal
11:     end if
12: end while
```

remain unchanged, write and update operations are always redirected to the new device, with update also invalidating the data on the old device. Firstly, we identify the spare end eldest devices (Line 2, Line 3) and then extract the set of devices which are needed to service the current request (Line 5) by identifying

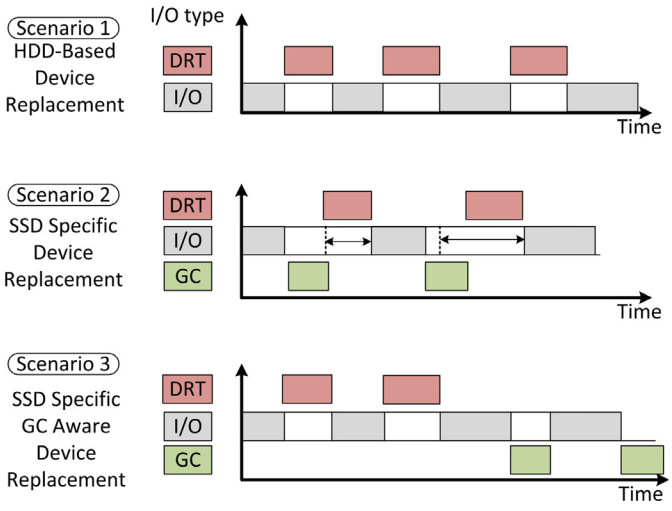

Fig. 2. A comparison of device replacement mechanisms.

the entry in the stripe mapping table that corresponds to the logical address in the meta data (Line 4) identified in the incoming request. If the eldest device is part of the request, and there is a device replacement ongoing, and the request is for an update, then it needs special consideration (Line 6). In this case, updates are redirected to the new device and the old data is invalidated (Line 7, Line 8), but read and write operations proceed as normal (Line 10). Update operations that do not target the eldest device, or when not in device replacement mode also proceed as normal.

The proactive technique reduces the probability of failure in multiple devices simultaneously as it starts data migration before a device fails. However, there will generally be some data which is not updated during the data migration process as it has not been accessed and will have to be actively migrated before next device replacement operation. To address this, the following section introduces a cold (coordinated) data migration operation where cold data in the failing device is partially migrated to the new device using an idle time detection approach.

5. Coordinated data migration

Device replacement and garbage collection tasks both generate (internal) I/O requests. Cold data migrations are usually triggered when there is an idle time, or low density workload patterns are detected. In the case where garbage collection also uses idle time detection (see [24]) as implemented in this architecture, the possibility exists that the sum total of idle time periods is insufficient for completing all of the internal I/O requests generated. Therefore when the migration operation overlaps with an ongoing garbage collection process in the target device performance of the system degrades.

Fig. 2 examples this problem in three scenarios. The first is an idle time reconstruction mechanism on a traditional magnetic disk array, and shows that to minimise the performance overhead in reconstruction, I/O requests generated to serve reconstruction (DRT) are triggered when an idle time period (white space on the *X*-axis) is detected. However this scenario enjoys two important properties: there is no garbage collection, and there are no time constraints as it is assumed other disks are not degrading.

These two properties are not enjoyed by the SSD array and this is shown in the other two scenarios. In the second, both data migration and garbage collection occur on the new spare device (only the new spare device needs considered as no new data is written to the old device, and garbage collection is not performed on the old device)in the same idle time periods and generate a number

of internal I/O requests. The drawback is that the idle time periods need to be longer in order for all these requests to be served. Whenever the mechanism detects an idle time period, the requests generated are inserted into the I/O queue of the corresponding device. Since the device replacement time must be as short as possible, these tasks can be assigned a higher priority than general I/O instructions. However invoking the garbage collector causes incoming requests to be blocked and I/O is adversely affected, indicated by the horizontal arrows in the white space idle time.

The third scenario is one in which coordinated data migration is introduced to interleave the overlapped requests. It monitors the ongoing garbage collection processes in the array via the Global FTL (*RAID.ssd_array.GC*). If garbage collection and device replacement processes attempt to access the target device during the same idle time period, the mechanism reschedules them by manipulating the I/O request priority levels to guarantee system response time. In normal operation, I/O request priorities are fixed. However if the target device is running out of free space and consequently has to garbage collect immediately then I/O requests generated by garbage collection take a higher priority than those of data migration, otherwise the data migration I/O requests maintain a higher priority. The third scenario of Fig. 2 illustrates this and shows that no I/O block is delayed, and the highest priority operations can complete.

The decision process for an individual device (indexed x) is informally described in Algorithm 3. Initially the priorities of all the

Algorithm 3 Coordinated data migration during hot swapping.

```
1:  while true do
2:      ∀ req ∈ ssd_array[x].request_queue |
3:              req.type = DRT ∨ req.type = GC • req.priority ← low
4:      if RAID.active_replacement. ∧ ssd_array[x].GC ∧ idletime
        then
5:          if ssd_array[x].freeblocks > ssd_array[x].hardthreshold
            then
6:              ∀ req ∈ ssd_array[x].request_queue | req.type = DRT •
7:                              req.priority ← high
8:          else
9:              ∀ req ∈ ssd_array[x].request_queue | req.type = GC •
10:                             req.priority ← high
11:         end if
12:     else
13:         Normal scheduling is performed
14:     end if
15: end while
```

I/O requests generated by reconstruction and garbage collection tasks are set *low* (Line 2, Line 3). If there is a request pending for both device replacement and garbage collection (Line 4) the resultant action is dependent on the number of free blocks (Line 5). If this is higher than a predetermined threshold (Line 5) then device replacement tasks may be assigned a high priority (Line 6, Line 7), otherwise garbage collection is set to the higher priority (Line 9, Line 10) as the memory is running out of free space. If there is no overlapping of these requests, scheduling continues as normal (Line 14).

As improving the efficiency of idle time detection is not in the scope of this study, a basic idle time detection function is adopted. To detect an idle time, the last access time of the corresponding memory is stored in a register. If there is not any request after a predefined time period this is considered as an idle time, and idle time is reset when the next request arrives.

6. Cost effective parity redistribution

The SSD RAID presented in this paper and in related work permit a parity redistribution process during device replacement—and require strict control over parity distribution percentages to maintain acceptable ageing ratios. However redistributing parity at device replacement time has two main bottlenecks: it increases write amplification, and it requires additional data movement between old and new devices—this adversely affects I/O performance. The cost effective parity distribution mechanism in this section addresses these problems and it achieves this without needing additional expensive data movements. This mechanism redistributes the parity data from the old device during the execution of (host) writes (hot data migration), or during a cold data migration period.

6.1. Parity redistribution with hot data migration

In this section we describe parity redistribution with hot data migration by exampling two scenarios. The scenarios differ from each other with regards to the status of the target stripe (full or partial), and the location of parity and data in the stripe. In the first scenario, given a partial stripe update with a component targeting the device being replaced, updated parity is not directly relocated to the hot spare device as, being the new device it is only allocated a small parity percentage. Instead, the device with the highest parity percentage that does not hold a component of the partial stripe being updated is selected and the updated parity written to it. This eliminates the need for an extra write operation and stripe unit migration required by previous techniques and therefore improves write amplification.

The second scenario is the case where the data being updated is a component of a full stripe across the array—the difference with the first scenario is that there is no free device to which updated parity can be relocated. In this scenario there are two possibilities: the first possibility is where parity distributions across the array is already balanced in terms of age distribution ratios and so parity must start migrating to the new device, the second possibility is where an existing device in the array needs additional parity located to it in order to maintain age ratios.

In the case of the first possibility, the updated data is written to the new hot spare device and the updated parity to the device on which the data element originally resided, and this can be performed in a single step. This eliminates the need for an additional write and swap operation required by previous techniques. In the case of the second possibility, two steps are necessary: firstly the updated data is written to the same device on which it originally resided; then the data stripe which resided on the second eldest device in the array is moved to the new hot spare device and the updated parity written to this next eldest device in the array. This improved write amplification over previous techniques due to fewer datums being moved.

6.2. Parity redistribution with cold data migration

Cold data migration refers to a period where the array is in an idle time (not currently servicing I/O requests). Actions taken in a cold data migration depend on the type of the piece of data being moved—determined using the dynamic strip mapping table. If the datum being moved is parity, then the same process as for hot data migration is used as this occurs no extra performance overheads. If the datum being moved is actual data then it is directly relocated to the new hot spare device.

This allows for parity data to be migrated to suitable target devices during idle time periods. This dynamic movement of data whilst the array is kept available to service I/O requests is the major enhancement over existing techniques—made possible because

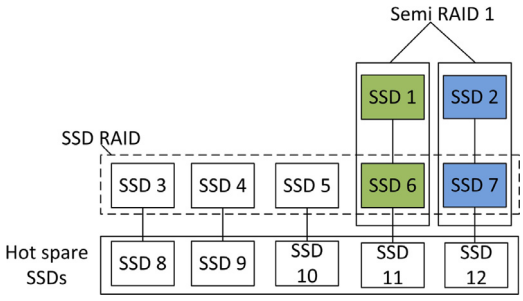

Fig. 3. Semi hybrid RAID after the second device replacement.

the old device is not immediately discarded upon needing replaced and is kept available for reading. This results in significant improvement in write amplification and I/O response times.

7. Semi-hybrid RAID

Proactive hot swapping initiates data migration from an ageing device to a new device before endurance limits become critical. Once the replacement process is complete there are two copies of valid pages: one in the old device, and one in the new device. This is a typical RAID 1 redundancy model, and may be exploited to improve read performance. In order to exploit this improved read performance further in this section we present a semi hybrid RAID architecture, where, when a device replacement is started the mechanism configures old and new devices in a manner similar to RAID 1. In doing so, the old device is not immediately discarded and is instead retained for a period in order to service some (non ageing) read requests. The architecture is referred to a semi hybrid RAID as it is effectively imposing a RAID 1 configuration over a device and its replacement, in a manner that is transparent to the SSD RAID architecture of the system.

This gives rise to a more complicated hardware and meta data architecture as the array is used through time as each device may be shadowed by a single older device that is used to service read requests; however the benefit is that a higher throughput of read requests can be serviced—particularly in the presence of the idle time detection techniques.

Fig. 3 illustrates an example of the semi hybrid RAID after the second device replacement process (we present the second replacement as it illustrates two instances of architecture emerging). At this point the system consists of devices 3, 4, 5, 6, and 7. The semi hybrid RAID 1 mechanism consists of the old devices 1 and 2 which have been replaced, shadowing the replacement devices 6 and 7. When further devices are removed from the array—for instance device 3 being replaced by device 8—these pairs would also form individual semi hybrid RAID configurations. New data is always written to the new device, with appropriate meta data updates. It is possible—depending on the age of the array—that each individual device in the array is shadowed by a RAID 1 partner old device.

Algorithm 4 shows informally how the semi hybrid raid works. The next queued access request is picked off the request queue and stored in the variable *request* (line 2). If the hybrid configuration is active and the request is a read (line 3) the mechanism then checks the location of the data by inspecting meta data(line 4)—this informs the device as to whether or not there is a second valid copy in the array, and incurs a very minimal performance overhead as the meta data status table is stored in local nvRAM. If the data is not backed up with a semi hybrid arrangement (such as device 3 in Fig. 3) *location.backup* will be null (line 5) and the data is read as normal (line 6). If the data is stored in a chip backup with semi hybrid raid but there is garbage collection or device replacement

Algorithm 4 Supporting read in semi hybrid RAID.

```
1:  while 1 do
2:      request ← RequestQueue()
3:      if request.type == read ∧ HR == 1 then
4:          location ← CheckMetaData(request.data)
5:          if location.backup == null then
6:              data ← Read(data, location.new)
7:          else
8:              if GC ∨ DR then
9:                  data ← Read(location.backup)
10:             else
11:                 data ← (Read(data, location.new)
12:                         ⊕ Read(data, location.backup))
13:             end if
14:         end if
15:     else
16:         We are not using a semi hybrid configuration
17:     end if
18: end while
```

taking place on the new chip (line 8) then data is read solely from the backup location (line 9), otherwise the data is read in parallel from both devices (line 11) which brings a slight performance enhancement. If the semi hybrid raid configuration is not active, or the original request was not a read (line 14) then all access requests are processed as normal. Any data writes are considered in an analogous manner: if the semi hybrid raid is not active then data is written as normal. If it is active, data is always written to the new device but meta data is also checked to ensure that any backup copy is invalidated.

8. Experimentation, results, and discussion

Experiments have been conducted using the Microsoft SSD simulator [25]. Each of the techniques presented have been implemented in the simulator, and experiments performed simulating device replacement. The experimental SSD array consists of five initial and five spare devices. The configuration parameters are as follows: reserved free blocks are set at 15%, minimum free blocks at 5%, and a chip contains a single SSD device with 1024 blocks per device, 64 pages per block, and a page size of 4 kB. Page read latency is set at 0.025 ms, write latency at 0.2 ms, block erase latency at 1.5 ms, and page stripe size at 4 kB. The primary reasons for selecting these parameters are consistency with previous reliability mechanism experiments. Experiments were conducted with synthetic traces (as published captured traces are not sufficient to age devices to desired levels) to analyse performance, device replacement time, and write amplification. Usage characteristics parameters required for the traces were set with a request size of 4 kB, an inter arrival time of 3 ms, and a probability of read access of 0.2 as default.

8.1. Performance evaluation

In this section two performance evaluations are given. Firstly, system response times are measured during the replacement process to evaluate efficiency of the proactive hot swapping and coordinated data migration. Secondly, read performance of the system is evaluated with the semi hybrid RAID once a device replacement is completed. Inter arrival time of requests is varied over a normal distribution with average times of 2, 3, and 4.2 ms (ms). Only traces dominated by random I/Os that cause frequent update operations on the parity device are considered as maximum reliability is achieved with a workload of small random writes. Basic idle

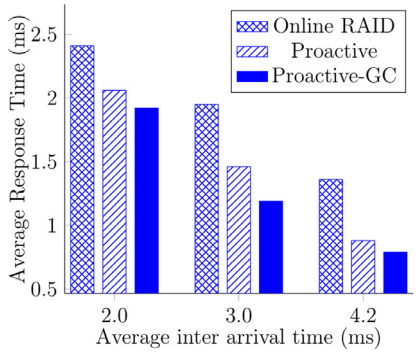

Fig. 4. Average response time varying inter-arrival time.

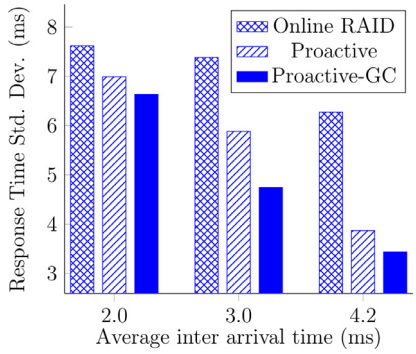

Fig. 5. Standard Deviations of response times varying inter-arrival time.

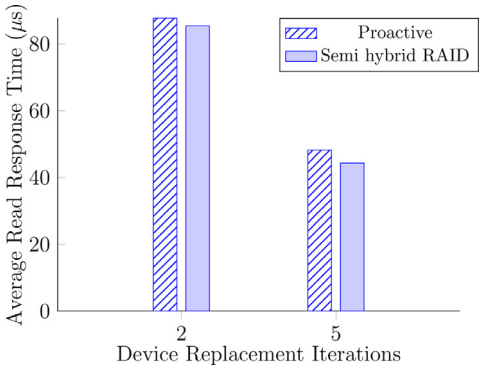

Fig. 6. Average read response time after device replacements.

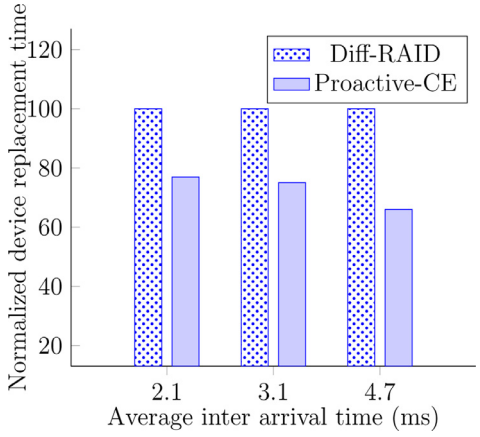

Fig. 7. Device replacement times for parity redistribution.

time detection approach for garbage collection and device replacement tasks is used.

Fig. 4 illustrates performance characteristics of three mechanisms—the online reconstruction of Section 5, the proactive hot swapping without garbage collection of Section 4, and the proactive hot swapping with garbage collection of Section 5. For a short inter arrival rate (2 ms), proactive hot swapping exhibits improvement in response time by 16% over basic online reconstruction; as inter arrival increases, proactive hot swapping exhibits further improvements. The system with co-ordinated garbage collection exhibits the most significant improvement across the range of inter arrival times—most clearly demonstrated by the results for an inter arrival time of 4.2 ms.

Standard deviations of system response times are given in Fig. 5, and they reflect similar positive results. Both the proactive method, and the proactive method with co-ordinated garbage collection exhibit smaller deviations across all three inter arrival times—with the co-ordinated garbage collection approach showing the smallest deviation in all three experiments.

Fig. 6 shows a comparison of read performances of the proactive hot swapping using the parity distribution of [1] and the semi hybrid RAID of Section 7 after two and five device replacements. Request size of read operations was fixed at 8 kB and probability of read access configured as 0.4. Results exhibit only marginally better performance in the semi hybrid architecture. However, the read performance of the semi hybrid RAID can be further improved with lesser parity percentage on the most aged device. Typically the most aged device holds 80% of total parity and 20% data and so the amount of migrated data is low when device replacement is triggered. As the percentage of data migrated to a new devices increases, the possibility of incoming read requests hitting rises, and so lesser parity percentages will improve the hit rate and consequently performance.

8.2. Device replacement time

Device replacement time refers to the wall clock time it takes from initiating a device replacement, to completing the process. This is important as the longer a replacement takes, the more likely it is that another device will tend towards an endurance threshold. Fig. 7 presents the results of comparing device replacement time of the on line cost effective parity distribution of Section 6 to the off line Diff-RAID mechanism of [2]. Default configuration parameters are used, and inter arrival times varied. Device replacement times are normalized to enable direct comparisons. The cost effective parity distribution mechanism exhibits a speed up of 34% over Diff-RAID—this is entirely due to proactive hot swapping. As inter arrival time increases so does the performance benefits due to co-ordination of idle time periods. This means that the total time taken to replace a device is approximately 35% shorter when proactive hot swapping is used.

8.3. Write amplification analyses

Fig. 8 shows the effect on write amplification under two different garbage collection approaches for Diff-RAID and the cost effective parity distribution with proactive hot swapping of Section 6. The first is idle time based, where cleaning takes place in the background. The second is threshold based where cleaning is triggered based on the amount of free space remaining. Both employ a greedy policy which selects the dirties blocks to clean. Under both garbage collection approaches, cost effective parity distribution mechanism exhibits improved write amplification with the idle time approach exhibiting the largest percentage improvement—Diff-RAID exhibiting an amplification factor of nearly 3, and cost effective parity distribution of only 2.

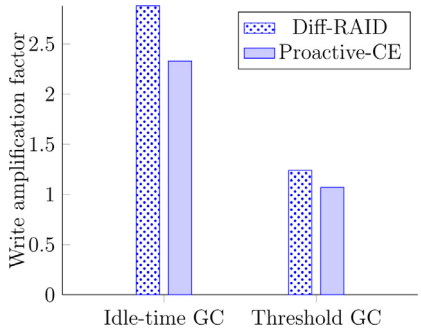

Fig. 8. Write amplification for random write workloads.

9. Conclusions and future work

In this paper we presented investigations into several techniques that enhance real time and performance capabilities of our SSD RAID array design such that they could be implemented in our FPGA based FTL. The proactive hot swapping mechanism predicts and initiates in advance when swapping will be required thereby maintaining system availability and reducing overheads of the on-line reconstruction. This eliminates a non-deterministic behaviour with respect to time. Garbage collection aware data migration further improves system performance during the replacement process. Simulation results demonstrate that the proactive garbage collection aware replacement mechanism significantly enhances I/O response time with low standard deviation during device swapping, which gives confidence in moving towards a system that provides real time guarantees. The results also indicate the on line parity redistribution technique improves write capability and replacement time compared to other existing works. Moreover, the semi hybrid RAID improves read performance after device replacement is complete.

As future work, we intend to investigate enhancing our mechanism with a real time garbage collector to determine the worst case execution time during replacement to address this potential area of non-determinism. A further important area of investigation is to measure the implementation costs of each of the techniques presented in a Verilog implementation as part of our system-on-chip design in order to determine real estate efficiency on different FPGA architectures upon which our FTL controller may be synthesised.

References

[1] I. Mir, A. McEwan, A reliability enhancement mechanism for high-assurance MLC flash-based storage systems, in: Embedded and Real-Time Computing Systems and Applications, IEEE, 2011, pp. 190–194, doi:10.1109/RTCSA.2011.58.
[2] M. Balakrishnan, A. Kadav, V. Prabhakaran, D. Malkhi, Differential RAID: rethinking RAID for SSD reliability, ACM Trans. Storage (TOS) 6 (2) (2010) 4:1–4:22, doi:10.1145/1807060.1807061.
[3] I. Mir, A. McEwan, A fast age distribution convergence mechanism in an SSD array for highly reliable flash-based storage systems, in: Communication Software and Networks, IEEE, 2011, pp. 521–525, doi:10.1109/ICCSN.2011.6014624.
[4] I. Mir, A. McEwan, A high performance reconfigurable flash management framework, in: Information Science, Electronics and Electrical Engineering, IEEE, 2014, pp. 1216–1220, doi:10.1109/InfoSEEE.2014.6947863.
[5] M. Komsul, A. McEwan, I. Mir, An FPGA-based development platform for real-time solid state devices, in: Information Science, Electronics and Electrical Engineering, 2, IEEE, 2014, pp. 1198–1203, doi:10.1109/InfoSEEE.2014.6947860.
[6] A. McEwan, M. Komsul, On-line device replacement techniques for SSD RAID, in: Digital System Design, IEEE, 2015, pp. 438–444, doi:10.1109/DSD.2015.69.
[7] A. McEwan, I. Mir, Age distribution convergence mechanisms for flash based file systems, J. Comput. 7 (4) (2012) 988–997, doi:10.4304/jcp.7.4.988-997.
[8] X.Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, R. Pletka, Write amplification analysis in flash-based solid state drives, in: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, ACM, 2009, pp. 10:1–10:9, doi:10.1145/1534530.1534544.
[9] M. Holland, G. Gibson, D. Siewiorek, Fast, on-line failure recovery in redundant disk arrays, in: The Twenty-Third International Symposium on Fault-Tolerant Computing, IEEE, 1993, pp. 422–431, doi:10.1109/FTCS.1993.627345.
[10] J.Y.B. Lee, J.C.S. Lui, Automatic recovery from disk failure in continuous-media servers, IEEE Trans. Parallel Distrib. Syst. 13 (5) (2002) 499–515, doi:10.1109/TPDS.2002.1003860.
[11] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, Z. Song, PRO: a popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems, in: Proceedings of the 5th USENIX Conference on File and Storage Technologies, USENIX Association, 2007, pp. 277–290.
[12] P. Chatterjee, R. Mishra, C. Biswas, B. Hallyal, Increased Data Availability in RAID Arrays Using Smart Drives, 2005, US Patent 6,892,276.
[13] B. Mao, H. Jiang, S. Wu, L. Tian, D. Feng, J. Chen, L. Zeng, HPDA: a hybrid parity-based disk array for enhanced performance and reliability, ACM Trans. Storage (TOS) 8 (1) (2012) 4.
[14] J.W. Hsieh, M.X. Liu, Configurable reliability framework for SSD-RAID, in: Non-Volatile Memory Systems and Applications Symposium, IEEE, 2014, pp. 1–6, doi:10.1109/NVMSA.2014.6927188.
[15] G. Zhang, W. Zheng, K. Li, Rethinking RAID-5 data layout for better scalability, IEEE Trans. Comput. 63 (11) (2014) 2816–2828, doi:10.1109/TC.2013.143.
[16] Y. Pan, Y. Li, Y. Xu, B. Shen, DCS: diagonal coding scheme for enhancing the endurance of SSD-based RAID arrays, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 35 (8) (2016) 1372–1385, doi:10.1109/TCAD.2015.2504333.
[17] T. Kim, S. Lee, J. Park, J. Kim, Efficient lifetime management of SSD-based RAIDs using dedup-assisted partial stripe writes, in: 2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2016, pp. 1–6, doi:10.1109/NVMSA.2016.7547184.
[18] S. Choudhuri, T. Givargis, Deterministic service guarantees for NAND flash using partial block cleaning, in: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, ACM, 2008, pp. 19–24, doi:10.1145/1450135.1450141.
[19] Z. Qin, Y. Wang, D. Liu, Z. Shao, Real-time flash translation layer for NAND flash memory storage systems, in: Real-Time and Embedded Technology and Applications Symposium, IEEE, 2012, pp. 35–44, doi:10.1109/RTAS.2012.27.
[20] L.-P. Chang, T.-W. Kuo, S.-W. Lo, Real-time garbage collection for flash-memory storage systems of real-time embedded systems, ACM Trans. Embedded Comput. Syst. 3 (4) (2004) 837–863.
[21] J. Lee, Y. Kim, G. Shipman, S. Oral, J. Kim, Preemptible I/O scheduling of garbage collection for solid state drives, IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 32 (2) (2013) 247–260, doi:10.1109/TCAD.2012.2227479.
[22] C. Lu, G.A. Alvarez, J. Wilkes, Aqueduct: online data migration with performance guarantees, in: Proceedings of the 1st USENIX Conference on File and Storage Technologies, USENIX Association, 2002, p. 21.
[23] R. Golding, P. Bosch, C. Staelin, T. Sullivan, J. Wilkes, Idleness is not sloth, in: Proceedings of the USENIX 1995 Technical Conference Proceedings, USENIX Association, 1995. 17–17
[24] E. Gal, S. Toledo, Algorithms and data structures for flash memories, ACM Comput. Surv. (CSUR) 37 (2) (2005) 138–163, doi:10.1145/1089733.1089735.
[25] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, R. Panigrahy, Design tradeoffs for SSD performance, in: USENIX 2008 Annual Technical Conference, USENIX Association, 2008, pp. 57–70.

Alistair A. McEwan is an Associate Professor in the Embedded Systems and Communications Group in the Department of Engineering at the University of Leicester. He holds a BSc from the University of Aberdeen, and a DPhil (PhD) from the University of Oxford. Prior to Leicester, he held several research fellowships at the Universities of Oxford, Kent, Surrey, and UCL. His research interests focus on the development and application of sound engineering principles to the design, development, and verification of dependable systems incorporating software - systems where high levels of assurance in the behaviours (and misbehaviours) are needed.

Muhammed Ziya Komsul received his B.Sc. and M.Sc. degrees from Trakya University, in Computer Engineering, Turkey, in 2010 and 2012, respectively. He is currently a Ph.D. student in the Embedded Systems and Communications group in the Department of Engineering at the University of Leicester. His research interests are flash based storage, real time systems, FPGA based architectures and embedded systems.