Age Aware Pre-emptive Garbage Collection for SSD RAID

Alistair A. McEwan and Muhammed Ziya Komsul

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

Abstract

Flash-based storage systems offer high performance, robustness, and reliability for embedded applications; however the physical nature of flash memory means that there are limitations to its usage in high reliability applications. In previous work, we have developed RAID architectures and associated controller hardware that increase the reliability and lifespan of these storage systems. However, flash memory needs regular garbage collection and this presents two issues in a high reliability context. The first issue concerns response times as when a garbage collector is active, the flash memory cannot be used by the application layer. This non-determinism in terms of response is problematic in high reliability systems that require real-time guarantees. The second issue concerns lifespan of flash chips. If the garbage collector is allowed free rein over erase operations while garbage collecting, this affects management of the lifespan of each SSD in the array.

In this paper we present an enhanced, dynamic, real-time garbage collection method for SSD RAID that does not ignore the strict age distribution management, while offering deterministic response times for access. Real-time efficiency is further improved by dynamically coordinating garbage collection across each device in the array. Our simulation results indicate that the dynamic garbage collection technique maintains the age distribution at a level that does not affect reliability of individual devices. This is evidences using various synthetic and realistic traces dominated by random I/O loads.

Keywords: , Solid state storage, Redundant array of independent disk, Real time storage, Garbage collection

1. Introduction

Flash memory is used as a primary storage medium for embedded systems because of a number of properties including high performance, low power consumption, shock resistance, and small physical size. The price of flash memory continues to decrease while density keeps growing with new technologies such as Multi-level cell (MLC) and Triple-level cell (TLC).

Although these technologies increase the density of flash memory, they suffer from lower erase endurance than Single Level Cell (SLC). Unlike magnetic disks, the lifespan of each flash cell depends upon the amount of erase operations as these slowly wear out cells. To mitigate this, Error Correction Codes (ECCs)—usually stored in meta data of each page— are used [1]. However, this technique is insufficient for MLC and TLC devices, and for component failures due to the limited size of the meta data area.

Conventionally, Redundant Array of Independent Disk (RAID) systems have been used to provide data protection against individual device failure, and to improve integrity of storage. However, RAID can not be directly applied to SSD arrays because of the risk of wearing out individual devices simultaneously [2]. [3] addresses this problem using a RAID architecture that prevents simultaneous wearing out of components by distributing parity data unevenly across all devices in the array. However this ignores the deterministic response time requirements of hard real time systems—primarily due to uncoordinated garbage collection.

Write and read operations are performed at page level (the smallest storage unit) but erase operations are performed at block level (a number of pages). A page can not be reused until the whole physical block is erased. An erase is the most time consuming operation of flash memory—nearly ten times longer than that of write. When applying an update-in-place policy (as used in magnetic devices), each update invokes an additional block erase—which slow downs system performance to an unacceptable level. To overcome this, flash employs out-of-place updates where the updated data is located to a new (free) place, invalidating the old data. After a period of time the memory suffers from a lack of free space due to these invalid pages. To reclaim them, garbage collection is performed.

These cleaning operations significantly degrade system performance and directly affect the lifetime of the device, and so there have been many techniques proposed to minimise these effects such as [4, 5, 6, 7]. Existing mechanisms are mostly triggered based on the amount of free space remaining

[8]. Once started, the garbage collector blocks all incoming requests for a non-deterministic time. This is problematic for real time system where guaranteed response times are required.

In [9] a pre-emptive mechanism is presented where incoming requests are served at predefined points. The algorithm employs two triggering threshold levels to distinguish the priority of garbage collection over I/O requests. However it is challenging to set suitable thresholds for pre-emption where the flash memory is used in a concurrent architecture such as our RAID array where strict management over ageing ratios is required. As each device has different workload weight due to (intentional) unevenly parity, constant thresholds do not increase the number of erasure for individual devices at the same rate. This causes the age distribution ratios to diverge.

In this paper we introduce a garbage collector into our RAID architecture that dynamically adjusts the thresholds of the pre-emptive method, taking into account age distribution ratios. We improve real time efficiency, deterministic access times, and system performance by globally coordinating the state of each garbage collector in the array.

The paper is organized as follows: in Section 2 we present motivation and related work. The architecture of the mechanisms is described in Section 3. Section 4 presents the dynamic threshold mechanism for garbage collection, with the dynamic thresholds calculations presented in Section 5. The mode controller mechanism for performance enhancement is presented in Section 6, and Section 7 presents our experimental results. Finally, in Section 8 we draw conclusions and describe future plans.

2. Motivation and Related Work

In studies such as [10] it is shown that the amount of garbage collection is significantly affected by workload type. For instance, small random write and update operations increase and slow down the amount of garbage collection, shown in [11]. As the reliability mechanism in [3] specially focuses on applications with high amounts of random and small sizes of write operations, garbage collection needs to be controlled.

One of the key points for efficient garbage collection is to determine an optimum threshold for triggering cleaning. This is particularly true of a pre-emptive mechanism such as [9] which provides deterministic response times without need for extra memory space. However, the limitation of [9] is that there is a tension between thresholds and our reliability concerns.

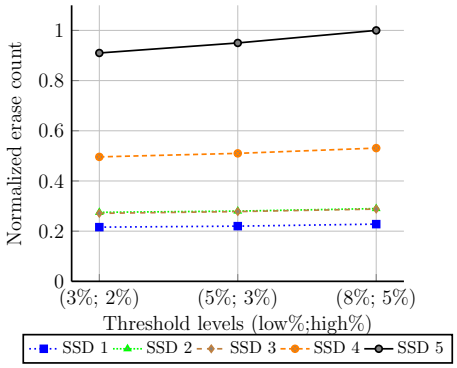

Figure 1: Normalized erase counts by varying triggering thresholds

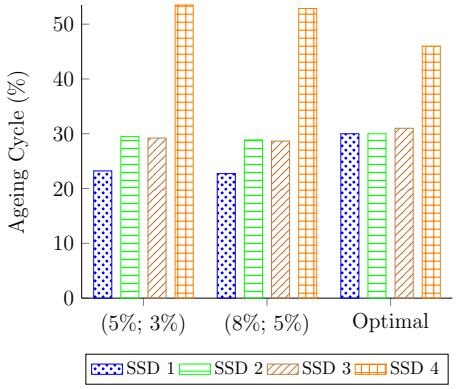

Figure 2: Comparison of age distributions varying triggering thresholds

Figure 1 presents experimental results conducted using the MSR SSD simulator[12], using the financial trace in [13], based on an array of five devices with uneven parity distribution. Threshold levels are described in terms of the percentage of free blocks remaining—the first of each pair indicates a low, and the second a high priority threshold. Results show that the normalized total erase count increases with thresholds, and significantly so as parity percentages increase (devices 4 and 5). Figure 2 presents these results in terms of ageing ratios of the first four devices relative to the fifth (most aged) device. These results show how ageing ratios deviate from the optimal levels (see [2]) for reliability (results for the first threshold level are omitted as they are similar to that for the second). For instance, device 4 wear outs

13% more than optimum, while device 1 wears out 21% less with thresholds of (5%; 3%)—firstly due to the fact that the ageing formula assumes a workload of pure random writes, and secondly due to the differing number of erase operations. The wear imbalancing technique is described in [14], and a known property of this mechanism is that there is not an optimum garbage collection threshold level, nor deterministic response times.

2.1. Related Work

Real time properties of flash garbage collection were first studied in [4], using a garbage collector thread for each real time task. [5] proposed a real time FTL that guarantees an upper bound for I/O operations of NAND flash using a partial block cleaning policy. [11] presents a real time FTL that employs a distributed garbage collection policy. A pre-emptive policy was proposed in order to suspend cleaning at pre-emption points in [9]. Most of the real-time solutions require additional buffer or memory to provide deterministic access guarantees but the pre-emptive solution does not—enhancing suitability in resource constrained systems. While these works offer guaranteed response times for I/O, they only consider non-determinism at a single chip level—not on storages with multiple chips. Additionally, these techniques do not consider the effect of real time algorithms on the life time of the chips—for instance those in a RAID configuration.

RAID technology has been applied to SSD storage in order to improve reliability, data integrity, and system performance [15, 16, 17]. In particular, RAID 0 redundancy has been studied to provide enhanced performance in [15]. A delayed partial parity update is presented in [16]. A configurable RAID mechanism that reduces parity overheads has been presented [17].

RAID 4 and RAID 5 hold parity to reconstruct original data in case of block errors (see [18]). However, the use of these techniques with SSD reveals the problem of simultaneously wearing out all devices. RAID based architectures were presented in [2, 3] that enhanced the reliability of SSD storage systems by mitigating this problem. These techniques employ uneven parity distribution to prevent simultaneous wearing out of the devices in the array. Maximum reliability is achieved only in the presence of random write workloads in [2], but [3] exhibits improved reliability for both sequential and random writes via a forced random write mechanism, as further described in [14]. Although those mechanisms enhance reliability, they ignore providing deterministic access times. The device replacement process with respect to

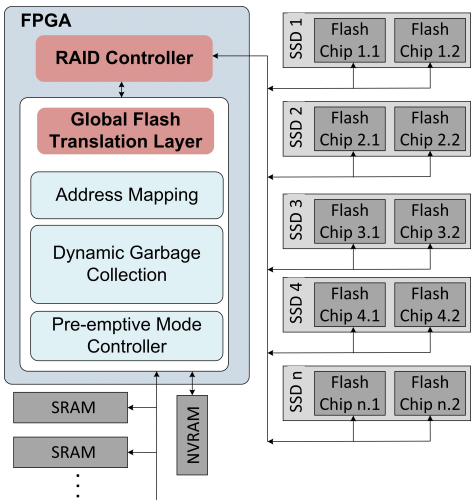

Figure 3: The architectural design

real time was discussed in [19] and enhanced with semi-hybrid RAID in [20]. However this technique does not address garbage collection.

An adaptive garbage collection threshold mechanism for a single device was presented in [21]. Global co-ordination to improve performance is proposed in [6]. A hybrid SSD architecture, using PRAM is presented in [22] and with combination of MLC and SLC flash memories in [23] to improve system throughput. However, these mechanisms only consider performance, and ignore deterministic access time.

The contribution of this paper is a dynamic garbage collection policy for an existing real time garbage collection mechanism over RAID to provide enhanced reliability and to improve guarantees for real time applications. We introduce a mechanism that co-ordinates garbage collection, using different modes of operation, by taking a holistic view of the system. This pre-emptive mode controller reduces the down time of the array during erase operations thereby causing fewer I/Os operations to be delayed.

3. Architecture and Design

Conventional FTL techniques usually allocate an independent and identical garbage collector for each device in an array, each of which has its own channel and internal register as in [24]. To maintain consistency with the

reliability mechanism, and to work towards real time performance, in this paper we adapt the architecture first presented in [25], as illustrated in Figure 3. This includes SSDs for storing data, memory components (SRAM, nvSRAM) for meta data, and an FPGA based storage management component. The FPGA based management component includes two main blocks.

Data Structure 1 The global view of the RAID array

```
struct RAID {
   RD_type raid_type
   integer length
   sequence metadata ssd_array
   stripe_map table
   integer period
}
```

The first main block is the RAID controller. This is responsible for partitioning the actual data into chunks using page-level striping where each incoming datum is divided into equal page sized parts (4 KB). The controller dynamically allocates a device for each actual data and parity data chunk as in [3], therefore a dynamic stripe mapping table is required. The RAID array is described by the pseudo code in Data Structure 1. It consists of the type of the array (RAID-4, RAID-5, Diff RAID, or flash-RAID; in this case flash-RAID), the number of devices in the array, a sequence of meta data structures describing the state of each device, the stripe mapping table which contains details of all stripes of data stored in the system, and a period to indicate how often dynamic garbage collection thresholds are calculated . This stripe mapping table is stored in SRAM memory. Only the relevant data structure elements are presented here for clarity.

The second main block is the Global FTL which takes a holistic view of the whole array, rather than a specific device. The global view of a single SSD is described by the pseudo code in Data Structure 2. For each device a record is kept of the current garbage collection mode (none, normal, or pre-emptive), the number of erasures performed, the number of free blocks remaining and total blocks, the ratio of low, medium, and high threshold priorities for changing status of pre-emptive garbage collection, the current and optimal ages for that device, the garbage collection efficiency factor for

Data Structure 2 The global metadata view of a single SSD

```
struct metadata  {
   GC_type mode
   integer erasures
   integer free, total
   priority low, medium, high
   integer current, optimal
   integer gce
   integer parity
   sequence block_metadata block
}
```

initiating/postponing cleaning, the parity percentage to be stored on the device, and a sequence of meta data structures describing the state of each block in the device. The Global FTL has three main components: address mapping, dynamic garbage collection, and the pre-emptive mode controller. The address mapping maintains a table of logical and physical addresses, stored in SRAM for each SSD, based on a page-level mapping table. The page status table is stored in NvSRAM to reduce the performance overhead of the meta data operations. The dynamic garbage collector manages all cleaning operations in the array. Finally, the pre-emptive mode controller manages real time aspects relating to garbage collection. Each of these components are described in detail in the following sections.

Data Structure 3 A single block

```
struct block_metadata {
   integer total
   integer free
   integer invalid
}
```

An entry in the block meta data of Data Structure 2 is described by the pseudo code in Data Structure 3 and consists of the total page number, the total free page number and the total invalid page number.

4. Dynamic Garbage Collection

The dynamic garbage collector adjusts triggering thresholds using meta data (number of erasures and free space) on each device. Two types of cleaning—high and low priority—are employed. The dynamic garbage collector adjusts triggering thresholds with the aim of maintaining age distribution ratios as close to optimum as possible. It also has the ability to postpone garbage collection if the cleaning process involves high levels of data migration (due to a large amount of valid pages in the victim block).

Algorithm 1 Dynamic adjustment of triggering thresholds

```
1: while true do
2:     if (metadata.erasures  mod  RAID.period) = 0 then
3:         for  each RAID.length  do
4:             fbp ← (ssd_array[i].free ÷ ssd_array[i].total)
5:             if ssd_array[i].low > fbp > ssd_array[i].high  then
6:                 Update thresholds
7:             else
8:                 Do not change thresholds
9:             end if
10:        end for
11:    else
12:        Do not change thresholds
13:    end if
14: end while
```

Algorithm 1 informally describes the periodic adjustment of triggering thresholds and age for each device, using current erase counts to determine a suitable period (line 2), and if a sufficent period has passed then the threshold evaluation and adjustment process begins and is performed sequentially on each device (line 3). The mechanism first stores the free block percentage for the given device in the variable `fbp` (line 4). If this free block percentage lies between the two thresholds (line 5) the thresholds require updating (line 6). This is done by comparing the current ageing ratio of the SSD in question (`metadata[i].current`) with its optimal level (`metadata[i].optimal`). If this is not optimum, it changes the ageing speed of the device by adjusting the threshold level. If the current ratio is greater than optimal the low threshold is decreased, and if the current ratio is greater than optimal it is increased.

However, the higher threshold may force an early cleaning on a block with a high number of valid pages. This causes several bottlenecks. Firstly, more valid pages in the victim block cause more data migration and thus cleaning is prolonged. Secondly, fewer invalid spaces will be reclaimed and cleaning efficiency may be quite low. Thirdly, if a number of incoming requests arrive at the same time then performance may suffer. To prevent this, garbage collection may be delayed if cost efficiency—in terms of time taken to completion—is relatively low.

Algorithm 2 Initiation of cleaning requests

```
1:  while true do
2:      fbp ← (ssd_array[x].free ÷ ssd_array[x].total)
3:      dirtiest ← find_dirtiest_block(ssd_array[x])
4:      ipp ← block[dirtiest].invalid ÷ block[dirtiest].total
5:      if fbp ≤ ssd_array[x].high then
6:          ssd_array[x].mode ← normal
7:          Initiate cleaning
8:      else if fbp ≤ ssd_array[x].medium then
9:          ssd_array[x].mode ← pre-emptive
10:         Initiate cleaning
11:     else if fbp ≤ ssd_array[x].low then
12:         if ipp ≤ ssd_array[x].gce then
13:             Do not initiate cleaning
14:         else
15:             ssd_array[x].mode ← pre-emptive
16:             Initiate cleaning
17:         end if
18:     else
19:         Do not initiate cleaning
20:     end if
21: end while
```

Algorithm 2 informally describes the decisions to initiate or postpone garbage collection for a given device `x`. The mechanism calculates the percentage of free space (line 2) and the invalid page percentage (line 4) of the dirtiest block (line 3). If memory is running out (line 5) than normal cleaning is initiated (line 6, line 7) otherwise the mechanism checks as to whether or not cleaning should be postponed. If the free block ratio is tending towards

its upper thresholds then postponing cleaning would result in long delays due to high priority cleaning being initiated—therefore a medium threshold level is checked (line 8). If it is lower than the medium threshold, then pre-emptive collection is triggered as the free block ratio is nearing high priority levels (line 9, line 10). If it is between the medium and low threshold, then efficiency of cleaning the victim block is checked (line 12). If the invalid page percentage of the dirtiest block is low, it is a high cost cleaning process and therefore cleaning is not started (line 13). In the case of a low cost clean—where the number of invalid pages is high—then it is initiated by generating low priority cleaning tasks and inserting them into the I/O queue of the device (line 15, line 16). If the device has enough free space then cleaning is not initiated at all (line 19).

As the performance overhead of high priority cleaning is higher and its completion time cannot be predicted in advance, high priority cleaning is avoided as much as possible, only being initiated when the system is starved due to a free space shortage. For this reason, the threshold for high priority cleaning is set as low as is practical and continually monitored.

5. Dynamic Thresholds

Low priority thresholds for a given device `i` are calculated using three parameters: existing low priority threshold, current age ratio, and optimum age ratio (`metadata[i].low`, `current`, and `optimal` respectively) where age ratios are relative to the most aged device indexed `n` and the first device is indexed 1. The calculation for the optimum age ratio of a device is given in Equation 1. The purpose of calculating low priority thresholds is to establish where it should lie in order to adjust the ageing speed of a device by increasing or decreasing the activation of cleaning.

$$metadata[i].optimal' = \frac{(metadata[i].parity \times (n-1)) + (100 - metadata[i].parity)}{(metadata[n].parity \times (n-1)) + (100 - metadata[n].parity)} \quad (1)$$

$$metadata[i].low' = metadata[i].low + metadata[i].optimal - metadata[i].current \quad (2)$$

The threshold calculation (Equation 2) for devices other than the oldest takes into account the difference between the current and optimum ages of the device. The difference between them is summed with the current threshold ratio to calculate the new threshold (`metadata[i].low'`). For example, if `metadata[x].current` is 0.465, and `metadata[x].optimal` is 0.461, device `x` has aged faster than is optimal. If the initial value of `metadata[x].low` is 0.1, `metadata[x].low'` will be 0.096. The new threshold is lower,and the ageing speed of device `x` is reduced when this new threshold is applied.

$$metadata[n].low' = metadata[n].low - (\sum_{i=1}^{n-1}(metadata[i].optimal - metadata[i].current)) \quad (3)$$

However, Equation 2 does not work for the most aged device (`n`), because `metadata[n].current` and `metadata[n].optimal` are always equal to each other as it is used as the reference point for the age ratios of all other devices in the array. Equation 3 shows the calculation which considers the overall ageing trends of the other devices in the array in order to calculate meaningful thresholds for `metadata[n].low'`. If the overall trend is high then `metadata[n].low'` is increased. For example, if the overall trend (the summation Σ))is -0.02, it shows that either the ageing speed of the most aged device is either slow, or the rest of the devices have increased ageing speeds. In this situation Equation 3 returns `metadata[n].low'` as 0.12—which will increase the triggering threshold and reduce the ageing speed of device `n`.

$$metadata[i].high' = priorityFactor \times metadata[i].low \quad (4)$$

$$0 < metadata[i].high < metadata[i].medium < metadata[i].low < 0.5 \quad (5)$$

High priority thresholds (`metadata[x].high`) are calculated as a multiple of the `metadata[x].low` and the (empirically determined) user defined variable `priorityFactor`, as shown in Equation 4. High priority thresholds should be updated at the same time as low priority thresholds. An invariant relationship between the thresholds and their bounds is given in Equation 5.

Initially the same threshold level is assigned to each device. When the

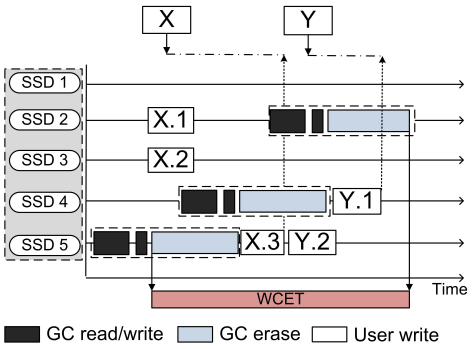

Figure 4: An example problem of pre-emption

most aged device, which retains the majority of the parity data, reaches a predefined age level then the current age ratio of all other devices in the array is calculated. The garbage collector periodically updates these calculations, and assigns updated threshold levels to each individual device until the most aged device reaches its endurance limit. This dynamic approach to calculating and moving threshold levels preserves the reliability enhancement mechanism but does not address performance issues. In order to address performance, a controlled form of pre-emption is introduced.

6. Pre-emptive mode controller

Pre-emptive garbage collection (PGC) provides more deterministic response times for a single device. However, its usage in a RAID array can result in an increased number of requests arriving during erase operations, and consequently the performance of the array suffers. This is because, in a RAID array, an individual garbage collector does not have a global view of the array, but read and write operations affect the whole array. For a RAID mechanism which uses uneven parity distribution it is normal that devices which retain more parity data will be garbage collected more in the case of random write dominant workloads. Therefore, if pre-emptive garbage collection is naively applied, different devices can be in different garbage collection states because of their varied workload rates. We use the term worst case execution time (WCET) to refer to the total time that the garbage collector consumes (across the array) during erase operations. In this section, we present a technique whereby the mode of the garbage collector is controlled in order to ameliorate this problem and improve real-time performance.

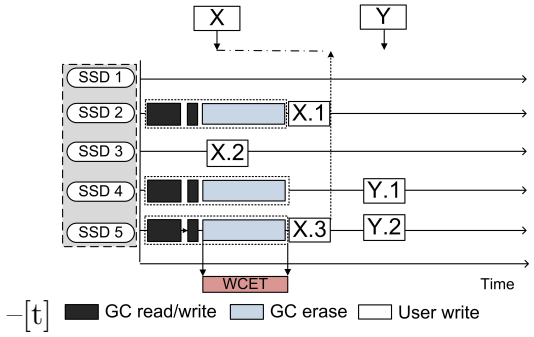

Figure 5: An example benefit of global pre-emptive mode control

An example of the problem stated above is presented in Figure 4. When a write operation for data `X` arrives from the host and is split into the three stripes `X.1`, `X.2`, and `X.3`, SSD 5 is in a state where the garbage collector has just completed read and write operations (partial GC tasks), and initiated an erase. As the FTL decision about where to physically place stripes is placed purely on age distribution, in this example `X.3` is destined for SSD 5. However the writing of stripe `X.3` overlaps the erase operation and so is delayed until the erase is complete—but the other stripes do not and so are written immediately meaning that the rate determining step of writing data `X` will be based on this erase time. A subsequent request to write data `Y` exhibits the same problem. This scenario is highly probable in a RAID architecture, particularly in the case of random write dominant traces. The key issue is the WCET—the total length of time that erase operations may be being performed anywhere in the array and therefore disrupting data writes. To eliminate the problem, the pre-emptive mode controller makes decisions about garbage collection by taking a view of the system as a whole. The mechanism considers all devices in the array when selecting a garbage collection state for an individual device.

The effects of the pre-emptive mode controller can be seen in Figure 5. Unlike naive pre-emptive cleaning, the mechanism considers all devices. In this example when space on a given device falls below the soft threshold garbage collection may be enabled for that device—but the mode controller also checks the other devices. This is what is shown for devices 2, 4, and 5—device 5 has reached a triggering threshold (as in the previous example). At the trigger, the pre-emptive mode controller has noted that devices 2 and 4 could also benefit from some cleaning and taken the decision to advance

cleaning on these devices also. The resultant parallel cleaning minimises the total time (WCET) spent in erase operations and the rate determining step for any delayed I/O operation reduces accordingly.

However, forced mode changes can result in overly aggressive early cleaning on a block with a small number of invalid pages. This can reduce the lifetime of a device more quickly and increase the cleaning cost as more valid pages are migrated. To overcome this, the mode controller checks the efficiency of cleaning a particular device before forcing the garbage collector to initiate a cleaning process, reducing the number of unnecessary erases.

Algorithm 3 Pre-emptive Mode Controller

```
1: ssd_array[all].mode ← none, RAID.length + 1
2: ipp ← block[dirtiest].invalid ÷ block[dirtiest].total
3: while true do
4:     if ssd_array[x].mode ≠ none then
5:         for each RAID.length do
6:             dirtiest ← find_dirtiest_block(ssd_array[i])
7:             if ipp > ssd_array[i].gce ∧ i≠x then
8:                 ssd_array[i].mode = ssd_array[x].mode
9:                 Initiate cleaning on the dirtiest block
10:            else
11:                Do not perform forced cleaning
12:            end if
13:        end for
14:    else
15:        Do not perform forced cleaning
16:    end if
17: end while
```

The decision process of the pre-emptive controller is informally described in Algorithm 3. Initially the states of all devices are set to no garbage collection as there is no need for cleaning (line 1). If any device (identified by the index `x`) changes garbage collection state to any other (line 4) the mechanism checks all other devices in turn for forced cleaning (line 5). The dirtiest block on the device under inspection is determined (line 6). If the invalid page number (`ipp`)of the dirtiest block is higher than the pre-determined `gce` of that device, garbage collection state is set to the same as that of device `x`, and cleaning commences (line 8, (line 9)). Forced cleaning is not

invoked if the number of invalid pages of the dirtiest block is low (line 11), or when there is no cleaning anywhere else in the array (line 15).

7. Experimental results

In this section the reliability and performance of the dynamic garbage collector and pre-emptive mode controller in a system employing parity distribution is evaluated under several experiments. We extend the simulator [12] with the requirements of Section 3 incorporating the RAID controller with uneven parity distribution and address mapping tables.

Configuration parameters are: reserved free blocks set at 15%, minimum free blocks at 5%. A device contains 2 flash chips, 1024 blocks per chip, 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, page stripe size at 4 kB, the ratio `h` for calculating high thresholds at 0.02 of the low threshold, and the medium threshold at 1% of the total number of blocks. The reason for selecting these parameters is to maintain consistency with previous experiments on reliability and to adopt generic SSD parameters. A number of synthetic and realistic traces are used to analyse age distribution variances. The default values of the synthetic traces used in our experiments are a request size of 4 kB, an inter arrival time of 3 ms, and a probability of read and sequential access of 0.2. We only consider traces which are dominated by random I/O that cause frequent update operations on the parity device, because maximum reliability of our RAID mechanism is achieved with a workload of small random writes. To create a variety of workload scenarios with synthetic traces an exponential and normal distribution is used for varying request sizes and inter-arrival times of requests. An array of four devices is used to enable direct comparisons with previous results.

7.1. Reliability Analyses

Two techniques are evaluated—pre-emptive garbage collection [9] with our reliability mechanism [3], and the dynamic garbage collection mechanism of Section 4, against optimal age distributions of [26].

Figure 6, Figure 7, and Figure 8 illustrate ageing characteristics with varying read access probabilities. Results show that the dynamic garbage collector exhibits improvement in ageing distributions over PGC-RAID in

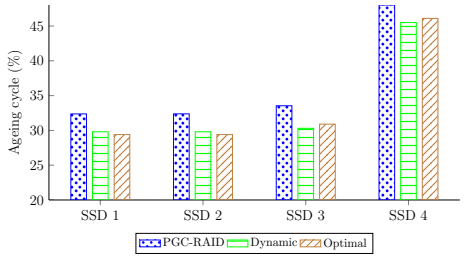

Figure 6: Age distributions with probability of read access 0.2

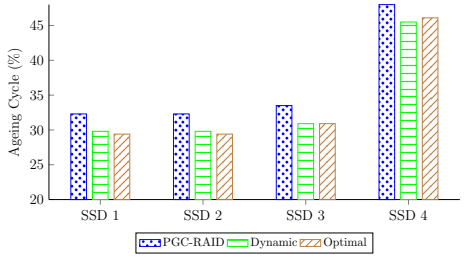

Figure 7: Age distributions with probability of read access 0.4

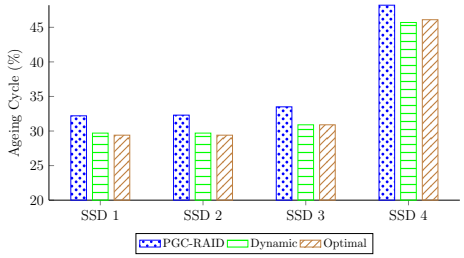

Figure 8: Age distributions with probability read access (0.6)

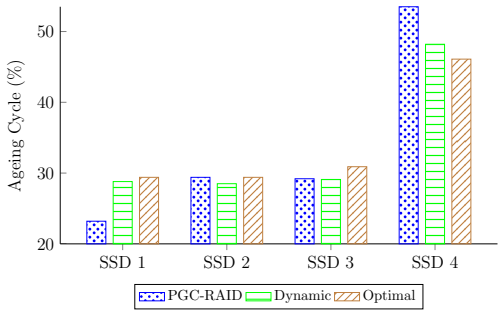

Figure 9: Age distributions with dynamic mechanism (Financial trace)

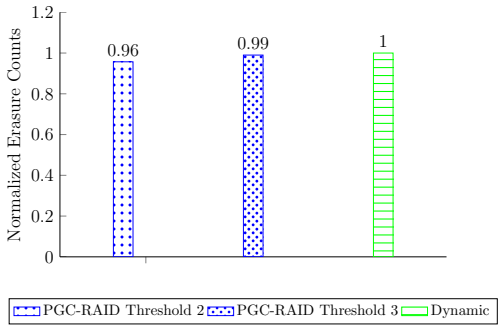

Figure 10: Normalized total number of erasures (Financial trace)

all cases. For instance, the age distribution of SSD 3 is 7% closer to its optimum level. As probability of read access increases the dynamic mechanism continues to perform better in terms of converging towards optimum levels.

Using the random write dominant realistic trace of [13], Figure 9 shows dynamic cleaning largely eliminates deviations in age distribution levels when coordinating low and high priority thresholds. For instance the ageing ratio of SSD 4 is much closer to optimum than PGC-RAID (by nearly 10%), and the ageing ratio of SSD 1 is 19% closer to its optimum level. This is because dynamic garbage collection has raised thresholds for SSD 1 while lowering them for SSD 4 when the deviation in age distribution is detected.

To observe the effect of garbage collection over the lifetime of the storage we compare it with existing techniques using constant thresholds for pre-emption. Threshold 2 is set with a soft level of 5% and a hard level of 3%, and threshold 3 is set with a soft level of 8% and a hard level of 5%. Figure 10 illustrates that the dynamic garbage collector has a slightly higher erase count than PGC-RAID—4% more erasure operations than PGC-RAID with

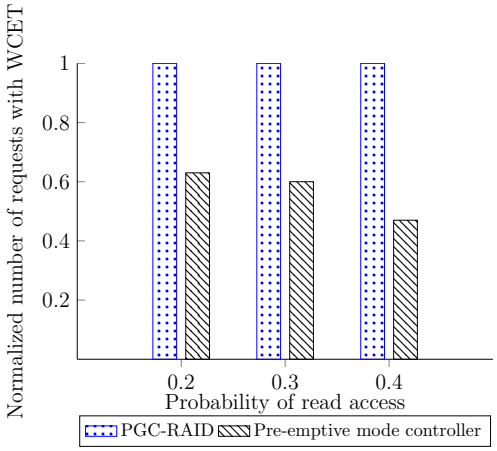

Figure 11: WCET comparisons

threshold 2, and no tangible difference with threshold 3. When threshold levels increase for PGC-RAID, erasure count continues to rise also. Results also indicate that reducing the period of the dynamic garbage collector for calculating thresholds helps ensure that age distribution converges towards optimum levels, but has a detrimental effect on device lifetime.

Overall, results demonstrate that the dynamic garbage collector improves the age distributions of devices and therefore reliability, along with offering upper bounds for I/O response, with minimal impact on device lifespan.

7.2. WCET and Performance Analysis

Real time efficiency of PGC-RAID [9] with the reliability mechanism [3], and the pre-emptive mode controller Section 6, is evaluated using a number of synthetic traces. Three different probabilities of read access were tested—0.2, 0.3, and 0.4. factor was set at 15% and average inter arrival time at 1.2ms. Figure 11 shows the amount of I/O requests arriving during active erases. For a read probability of 0.2, 40% more tasks are delayed by PGC-RAID over the pre-emptive mode controller. As read probabilities increase, so do improvements offered by the pre-emptive mode controller. Moreover, the pre-emptive mode controller shows enhanced average response times of around 6% over PGC RAID for all traces.

Standard deviation of response times are given in Figure 12 for the same experiments. For all cases the pre-emptive mode controller combined with dynamic garbage collection shows improvement over PGC-RAID. For a write

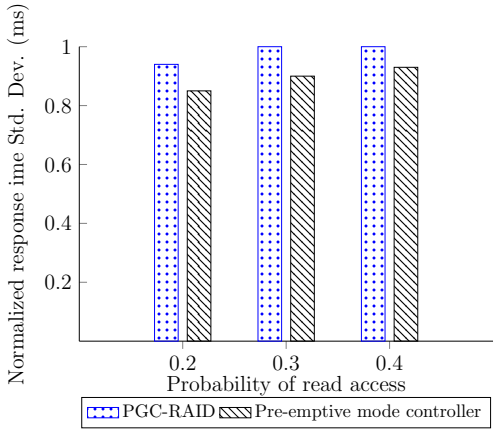

Figure 12: Standard deviations of response times varying read acccess probability

dominant workload of 0.2 read access probability this improvement is 15% over PGC-RAID. As probability of read access increases the improvements decrease—10% for 0.3, and 7% for 0.4, and with negligible differences above 0.4 due to garbage collection being invoked much less frequently. These results indicate that the pre-emptive mode controller combined with dynamic garbage collection provides more deterministic response times over PGC-RAID, due to the global co-ordination.

8. Conclusions and Future Directions

In this paper we presented enhanced real time garbage collection mechanisms for SSD RAID arrays. We identified that existing real-time garbage collection techniques have undesirable effects on our reliability mechanism. In order to ameliorate the impact, we introduced a dynamic garbage collection mechanism that dynamically adjusts thresholds levels for each SSD to achieve the optimum ageing distribution ratios for the reliability mechanism. Moreover, the pre-emptive mode controller improves real time efficiency by globally coordinating garbage collection states. Simulation results demonstrate that the dynamic garbage collector provides better real time access guarantees and also maintains the reliability mechanism. The pre-emptive mode controller reduces the down time of the array during erase operations, thereby causing fewer writes to be postponed and with low standard deviation compared to other published techniques.

As future work, we aim to improve WCET with an architecture that completely avoids I/O requests from being blocked by erase operations by exploiting the inherent concurrency of the architecture. We are also continuing progress towards a full Verilog FPGA implementation of garbage collector for integration with our existing implementation and test beds.

References

[1] S. A. Chamazcoti, S. G. Miremadi, On Designing Endurance Aware Erasure Code for SSD-based Storage Systems, Microprocessors and Microsystems 45 (PB) (2016) 283–296. doi:10.1016/j.micpro.2016.06.003.

[2] M. Balakrishnan, A. Kadav, V. Prabhakaran, D. Malkhi, Differential RAID: Rethinking RAID for SSD Reliability, ACM Transactions on Storage (TOS) 6 (2) (2010) 4:1–4:22. doi:10.1145/1807060.1807061.

[3] I. Mir, A. McEwan, A Reliability Enhancement Mechanism for High-Assurance MLC Flash-Based Storage Systems, in: 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Vol. 1, 2011, pp. 190–194.

[4] L.-P. Chang, T.-W. Kuo, S.-W. Lo, Real-time Garbage Collection for Flash-memory Storage Systems of Real-time Embedded Systems, ACM Transactions on Embedded Computing Systems 3 (4) (2004) 837–863.

[5] 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, CODES+ISSS '08, 2008, pp. 19–24.

[6] Y. Kim, J. Lee, S. Oral, D. Dillow, F. Wang, G. Shipman, Coordinating Garbage Collection for Arrays of Solid-State Drives, IEEE Transactions on Computers 63 (4) (2014) 888–901. doi:10.1109/TC.2012.256.

[7] S. J. Kwon, A. Ranjitkar, Y.-B. Ko, T.-S. Chung, FTL Algorithms for NAND-type Flash Memories, Design Automation for Embedded Systems 15 (3) (2011) 191–224. doi:10.1007/s10617-011-9071-9.

[8] E. Gal, S. Toledo, Algorithms and Data Structures for Flash Memories, ACM Computing Surveys (CSUR) 37 (2) (2005) 138–163. doi:10.1145/1089733.1089735.

[9] J. Lee, Y. Kim, G. Shipman, S. Oral, J. Kim, Preemptible I/O Scheduling of Garbage Collection for Solid State Drives, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32 (2) (2013) 247–260. doi:10.1109/TCAD.2012.2227479.

[10] F. Chen, D. A. Koufaty, X. Zhang, Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives, ACM SIGMETRICS Performance Evaluation Review - SIGMETRICS '09 37 (1) (2009) 181–192. doi:10.1145/2492101.1555371.

[11] Z. Qin, Y. Wang, D. Liu, Z. Shao, Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems, in: 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, 2012, pp. 35–44.

[12] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy, Design Tradeoffs for SSD Performance, in: USENIX 2008 Annual Technical Conference, ATC'08, USENIX, 2008, pp. 57–70.

[13] Storage - UMass Trace Repository (June, 2007), Available: `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[14] A. McEwan, I. Mir, An Embedded FTL for SSD RAID, in: 2015 Euromicro Conference on Digital System Design (DSD), IEEE, 2015, pp. 575–582. doi:10.1109/DSD.2015.39.

[15] S. Rizvi, T.-S. Chung, Data Storage Framework on Flash Memory Based SSD RAID 0 for Performance Oriented Applications, in: The 2nd International Conference on Computer and Automation Engineering (ICCAE), Vol. 1, 2010, pp. 126–128.

[16] S. Im, S. Im, S. Im, S. Im, D. Shin, D. Shin, D. Shin, D. Shin, Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD, IEEE Transactions on Computers 60 (1) (2011) 80–92.

[17] J.-W. Hsieh, M.-X. Liu, Configurable Reliability Framework for SSD-RAID, in: Non-Volatile Memory Systems and Applications Symposium (NVMSA), IEEE, 2014, pp. 1–6. doi:10.1109/NVMSA.2014.6927188.

[18] S. Chen, D. Towsley, The Design and Evaluation of RAID 5 and Parity Striping Disk Array Architectures , Journal of Parallel and Distributed Computing 17 (1-2) (1993) 58 – 74.

[19] A. McEwan, M. Komsul, On-Line Device Replacement Techniques for SSD RAID, in: 2015 Euromicro Conference on Digital System Design (DSD), IEEE, 2015, pp. 438–444. doi:10.1109/DSD.2015.69.

[20] A. A. McEwan, M. Z. Komsul, Reliability and Performance Enhancements for SSD RAID, Microprocessors and Microsystems (2016) – doi:http://dx.doi.org/10.1016/j.micpro.2016.11.012.

[21] Y. Qin, D. Feng, J. Liu, W. Tong, Z. Zhu, DT-GC: Adaptive Garbage Collection with Dynamic Thresholds for SSDs, in: 2014 International Conference on Cloud Computing and Big Data (CCBD), IEEE, 2014, pp. 182–188.

[22] G. S. Choi, I. Lee, M. Sung, C. Im, A Hybrid SSD with PRAM and NAND Flash Memory, Microprocessors and Microsystems 36 (3) (2012) 257–266. doi:10.1016/j.micpro.2011.11.001.

[23] J.-W. Park, S.-H. Park, C. C. Weems, S.-D. Kim, A Hybrid Flash Translation Layer Design for SLC-MLC Flash Memory Based Multibank Solid State Disk, Microprocessors and Microsystems 35 (1) (2011) 48–59. doi:10.1016/j.micpro.2010.08.001.

[24] A. Gupta, Y. Kim, B. Urgaonkar, DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings, ACM SIGPLAN Notices - ASPLOS 2009 44 (3) (2009) 229–240.

[25] I. Mir, A. McEwan, A High Performance Reconfigurable Flash Management Framework, in: 2014 International Conference on Information Science, Electronics and Electrical Engineering (ISEEE), Vol. 2, IEEE, 2014, pp. 1216–1220. doi:10.1109/InfoSEEE.2014.6947863.

[26] A. McEwan, I. Mir, Age Distribution Convergence Mechanisms for Flash Based File Systems, Journal of Computers (JCP), Academy Publisher 7 (4) (2012) 988–997. doi:10.4304/jcp.7.4.988-997.