IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 12, 2019, accepted September 23, 2019, date of publication September 30, 2019,
date of current version October 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2944764

A Real-Time Dependable Flash Storage System

ALISTAIR A. MCEWAN AND MUHAMMAD ZIYA KOMSUL
School of Engineering, The University of Leicester, Leicester LE1 7RH, U.K.
Corresponding author: Alistair A. McEwan (alistair.mcewan@le.ac.uk)

This work was supported by the University of Leicester, U.K.

ABSTRACT One of the limitations of flash memory in real-time and high dependability systems is its need for garbage collection, resulting in performance degradation due to non-deterministic response times. Recent work has presented RAID architectures for solid state storage systems. These RAID architectures increase the dependability from a data storage perspective but they do not provide application level dependability when real-time response times are required. In this study we present a garbage collection aware Flash Translation Layer that offers guaranteed access time to a solid state RAID array by managing incoming requests and preventing them from being blocked by ongoing garbage collection. We present a novel serial technique with a dynamic page allocation mechanism that eliminates non-deterministic behaviours of the garbage collectors in the array. The result is real-time access guarantees that maintain the data dependability enhancements using a run time parity migration technique. The mechanisms are evaluated using a trace driven simulator and a number of synthetic and realistic traces. Simulation results indicate that the garbage collection aware techniques offer improved upper bound response times for I/O requests of up to 73% compared to an existing mechanism, without disturbing the data dependability at the storage level. Traces dominated by random writes exhibit similarly significant enhancements.

INDEX TERMS Data storage systems, flash memories, fault tolerance, flash translation layer, garbage collection, real-time systems, SSD RAID.

I. INTRODUCTION

Flash memory (solid state memory, also referred to as SSD) is effectively ubiquitous in embedded devices, and even increasingly so in standard computing environments, because of the properties it enjoys including higher performance and access bandwidth, low power consumption, shock resistance, and increasingly smaller physical size. Storage capacity of solid state memory has increased massively over the past decade from a few megabits, to gigabits and to terabits, while the price per unit continues to decrease dramatically.

Despite these very positive properties, solid state memory also has some intrinsic drawbacks. There are two distinct types of flash memory commonly used—NOR, and NAND—the names refer directly to the microelectronic construction of the circuits that implement the memory. NOR memory benefits from fast random reads but suffers from very slow read and erase operations. These properties mean that it is typically used for holding relatively non-volatile data such as code for execution in an embedded system. On the other hand, NAND memory enjoys faster write operations and higher density of data storage. These properties mean that it is typically used for data storage in much the same way one would consider a traditional file system and traditional file storage. In this paper, we consider NAND flash.

Although the properties of NAND solid state storage make it suitable for traditional file and data storage applications, it also has some disadvantages that need to be taken into consideration. This includes a bounded lifetime—or specifically, a bounded number of times that areas of the memory can be erased (in preparation for rewriting) before it becomes unreliable. Unlike traditional magnetic storage devices (commonly referred to as hard disk drives, or HDD) the lifespan of each cell in a given chip depends on the number of erase/write operations that are carried out as the physical nature of this operation slowly wears out a cell until it reaches wear out state where it cannot be relied on to report data stored correctly. The number of erase/write operations that a given cell can take depends on a number of factors including manufacturing quality and usage patterns. While these will differ between manufacturers and devices, it is usually a figure known a priori for a given device thereby giving the user some understanding of the lifetime of the device before data may become unreliable or lost.

Error Correction Codes (ECC) [1] are checksum based techniques used to help maintain data integrity and improve lifespan of a given device. While these help in ensuring data is reliably reported up to a certain threshold (bit error limit)

The associate editor coordinating the review of this manuscript and approving it for publication was Hao Luo.

that can be determined by a checksum calculation, they offer no assistance in a device that is failing beyond the threshold of the checksum calculation, or has failed completely. Wear levelling algorithms are also employed to help prolong device lifespan. These algorithms typically ensure that data is written in an even pattern across a device—meaning that life-consuming erase operations also follow an evenly scattered pattern and the device will wear out evenly rather than suffer disproportionate performance degradation due to the overuse of one given section or area.

Advances in increasing storage density on devices include Multi Level Cell (MLC) and Triple Level Cell (TLC). While these are effective ways of fabricating chips that offer increased storage density and decreased device cost over de facto Single Level Cell (SLC) devices, they also come with the burden of reduced life expectancy and storage dependability. Moreover, ECC techniques are not in themselves sufficient to ameliorate the problems resulting from wear out of MLC devices. Consequently, caution is advised with use of these devices in systems or application domains require integrity of data and data updates.

Conventionally, Redundant Array of Independent Disk (RAID) systems have been widely used to provide data protection against chip-level failure and to improve integrity of storage such as in [2] using RAID–4 and RAID–5 parity based techniques—employing a reserved element of the array to store parity data. When a multibit error that can not be recovered using ECC occurs, correct data can be recalculated at chip, block, or page level.

However, RAID can not be directly applied to SSD arrays because of the risk of wearing out individual devices in the array simultaneously [3]. Reference [4] addresses this problem using a novel RAID architecture that enhances reliability by protecting against component failures using a load imbalancing technique. This technique prevents the simultaneous wearing out of components by distributing parity data unevenly across all devices in the array. This enhances reliability, but ignores the deterministic response time requirements of hard real-time systems—primarily due to uncoordinated garbage collection.

RAID offers significant improvement to storage level dependability but does not meet the (sometimes strict) application level dependability of real-time systems where guaranteed response times may be required—due to the erase-before-write requirement where exiting data cannot be overwritten before being erased. When storage is requested for new data, free locations are allocated. Any old data is marked as invalid, and the pages occupied by this invalid data are eventually reclaimed by a garbage collector so they can be freed. The act of garbage collection incurs a significant performance overhead because it involves moving valid pages in selected blocks into free blocks before erasure can happen. Moreover garbage collection is usually triggered when there is an idle period—however it is challenging to predict when these will happen in I/O workload streams and indeed some write workloads with significant burst characteristics do not generally have sufficient idle time for garbage collection.

In this paper, we introduce an efficient and dependable garbage collection mechanism into our RAID architecture. The garbage collector provides guaranteed response time for I/O by preventing blockage due to an ongoing garbage collection operation. We show that our mechanism achieves higher bandwidth and lower worst case execution time (WCET) either by dynamically reallocating the target chip or by using redundant data to service incoming requests. In summary, the contribution of this paper is:

- Garbage collection aware Flash Translation Layer (FTL) operations that are integrated into an FTL, enabling real-time response times even during active garbage collection processes;
- Garbage collection techniques for SSD RAID that explicitly consider the wear out problem in the context of real-time and dependability;
- A serialized garbage collection technique that ameliorates issues arising from multiple garbage collectors acting on multiple elements in an SSD array;
- An on-line parity migration mechanism that preserves ageing ratios, whilst maintaining dependability levels and deterministic response times;
- Experimental results, taken using the SSD simulator [5], with results demonstrating a significant improvement in average and maximum response times for synthetic and realistic workloads.

The paper is organized as follows: in Section II we present background information, the motivation for our research, and related work. Section III presents our garbage collection aware FTL and operations. Section IV presents the serialized garbage collection approach. The support for on-line parity migration is presented in Section V. Experimental results are presented and discussed in Section VI, and Section VII summarises our conclusions and areas of future investigations.

II. BACKGROUND

An area of flash memory consists of a set of blocks, where each block consists of a number of pages as shown in Fig. 1. A page contains two areas—a data area (for storing regular data) and an Out Of Band (OOB) area for metadata such as error correction codes (EEC), logical block address (LBA), logical page address (LPA), and page status (valid, invalid or free). The size of the data area varies between 512-2048 bytes while size for OOB vary between 16-64 bytes. The data stored in a page is referred to as a *stripe*. This general structure is captured informally in Data Structure 2. Data types used in this definition are given in Data Structure 1. A definition of the additional meta data in the device level register is not given as it is not used directly in this paper.

Unlike magnetic disks and volatile memories such as DRAM, flash memory requires erase-before-write. That is to say, blocks cannot be overwritten and must be erased before they can be re-used. Granularity of erase and read/write operations are different—erase operations are performed at block

Data structure 1 Data Types Supporting an SSD Definition

```
1: $Bit : 0|1$
2: $GarbageCollection : active|inactive$
3: $PageStatus : valid|invalid|free$
4: $LBA : \mathbb{N}$
5: $LPA : \mathbb{N}$
6: $ECC : Bit$
```

Data structure 2 Basic Structure of an SSD

```
1: $Register : GarbageCollection \times DeviceMetaData$
2: $Data : (\mathbb{N} \nrightarrow Bit)$
3: $OOB : (ECC \times LBA \times LPA \times PageStatus)$
4: $Page : (Data \times OOB)$
5: $Block : \mathbb{N} \nrightarrow Page$
6: $SSD\_Device : (\mathbb{N} \nrightarrow Block) \times Register$
```

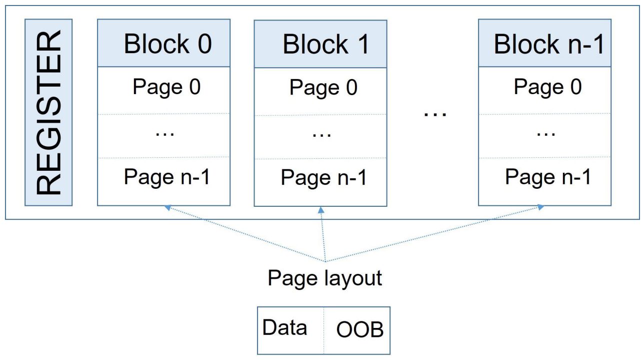

FIGURE 1. Internal structure of flash memory (SSD).

level but read and write operations at page level. A flash page can be in one of three different states: valid, invalid, or free. If there is no data written into a page it is marked free, a write operation to a free page changes its state to valid, and if that data subsequently needs updating it is written elsewhere in a new valid page and the original marked invalid.

To write data from an input register to a physical page takes typically 200–700μsec. However, erase operations are rather slower than a write operation and take in the order of 1.5–3$msec$. Read latency from a physical page location to a register is much faster than either, being in the order of 20–50μsec. These differences in timing are why flash memories do not typically employ an update-in-place operation and instead use the invalid page approach. To record changes and mapping between logical and physical addresses (as the physical address of a specific item of data can migrate during its lifetime due to invalid page migration) an address mapping table is kept—often stored in an SRAM memory.

The physical nature of the storage action means that each cell can tolerate a bounded number of erase operations before it wears out and reporting becomes unreliable—typically between ten and one hundred thousand erases. If a given cell is erased more regularly than neighbouring cells, it may become a rate determining step resulting in a whole page becoming unusable. Wear levelling techniques are employed to ensure that the writing—and consequently erasure—of data wears out the whole real estate in as even a manner as possible so as to avoid hot spots of unreliability [1].

A primary function of the FTL is to allow a device to be used as a normal block device by abstracting from internal structure. Part of this is the periodic reclamation of invalid pages for reuse via a process known as garbage collection.

Fig. 2 illustrates a basic garbage collection operation. The garbage collector first selects a victim block based on a most needy algorithm that selects the block that has the most number of dirty (invalid) pages. It then copies all valid pages from the victim block to an empty (free) block and updates indexing information in the address mapping table. Once all valid pages are copied, the entire victim block is erased and the pages in the victim block are free for re-use [6].

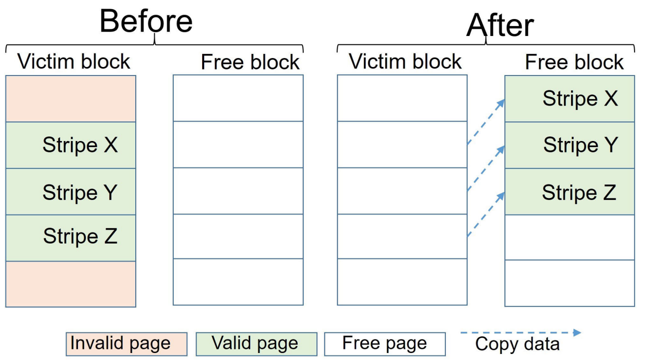

FIGURE 2. Moving data and erasing a victim block.

Garbage collection has a negative impact on the performance and lifetime of the memory. For instance if garbage collection overlaps with an instruction request from the host, the instruction must wait until garbage collection completes, causing unwanted latency and bottlenecks. Moreover because the erase operation consumes lifespan of the selected cell, blocks must be chosen carefully in order to minimize unnecessary erase operations but maximize available space.

Previous studies report that the number of garbage collections is highly affected by workload type. For instance, random writes and updates in small sizes increase the need significantly [7]. These types of workloads slow down cleaning because of the increased number of valid pages copied [8]. This study focusses on applications with high amount of random and small size write operations as this is the workload patterns that our dependability mechanisms target.

A. RAID AND DEPENDABILITY

RAID protects against data loss in the case of a single component failure. Blocks of data are spread across multiple devices, one of which is used to store a parity checksum calculation. When an individual device fails, lost data can be calculated and a replacement reconstructed. RAID-4 stores all parity data on a single device, while RAID-5 distributes parity across the array. These schemes work well for magnetic disks which suffer non-deterministic hardware failure rather than solid state memory which suffers wear out failure.

RAID-5, for instance, can be implemented with multiple SSD devices as shown in Fig. 3, with some of the FTL operations promoted to the RAID level. When the host system requests data storage, the data in question (for instance, X) is initially stored in a buffer internal to the RAID array. The RAID controller then breaks this down into $n-1$ stripes (X_1, X_2, X_3, X_4) where n is the number of devices in the array. An additional parity stripe (X_P) is generated by XOR-ing the data stripe units. Stripes may be either block level or page level depending on system configuration—in this paper we typically assume page level stripes. Each stripe (including the parity) is allocated a physical device to which it is written. Parity blocks are distributed evenly across different devices as shown by additional example stripes Y and Z. The reasons for this are twofold—as a change in any data will certainly require a rewrite of parity the load is distributed across the system; and the amount of data recalculations (as opposed to parity calculations) is distributed and thus speeds up device replacement. However these are only beneficial in a magnetic disk environment and neither of these reasons ameliorate the problems of wear out failure in solid state devices.

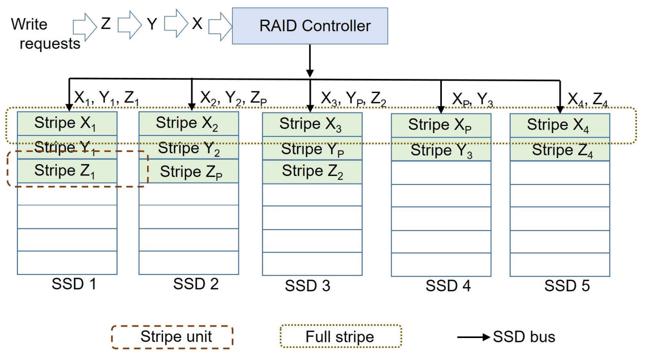

FIGURE 3. Example RAID-5 structure with data stripes.

A basic informal definition of the structure of an SSD array assumed in this paper is given in Data Structure 3. The array (*Devices*) is a sequence of individual devices. A request queue holds incoming requests to read from, or write to, the array. A mapping table maps physical block addresses (PBA) and physical page addresses (PPA) to their logical counterparts. A global array notes the garbage collection status—active or inactive—of each of the individual devices.

Data structure 3 Basic Structure of an SSD RAID Array

```
1: $Request : read|write$
2: $PBA : \mathbb{N}$
3: $PPA : \mathbb{N}$
4: $Devices : \mathbb{N} \nrightarrow SSD\_Device$
5: $RequestQueue : \mathbb{N} \nrightarrow (Request \times Data)$
6: $GC\_Array : \mathbb{N} \nrightarrow \mathbb{B}$
7: $MappingTable : (\mathbb{N} \nrightarrow PBA) \times (\mathbb{N} \nrightarrow LBA)$
```

In the interests of clarity, this definition does not include elements for RAID-level FTL features not covered directly in this paper such as wear levelling and dependability.

Two distinct types of write operations are used in RAID arrays. If data is written across all devices in the array then it is know as a *full stripe write* or *sequential write*; thus stripe X in Fig. 3 is a full stripe write. However, if the number of stripes does not span the array then it is known as a *small random write*, or *random write* for short; thus stripes Y and Z in Fig. 3 are random writes. Random writes may be used for complete data items, or when an update to data (that does not affect the whole array) takes place. For instance if an update to the data in stripe Y_1 was requested the RAID controller would execute the following steps:

1) Read Y_2 and Y_3;
2) Calculate the new parity Y_P using the new Y_1 and the stored Y_2 and Y_3;
3) Write the new data Y_1 to the same device, and write the new parity Y_P to the same device.

This process is described informally in Algorithm 1. This presentation, as with other algorithms given in this paper, is kept informal so as not to restrict the system to a given specification or architecture, but instead to broadly describe the algorithms in a form amenable to code development. The input to the algorithm is the data including the stripe to be updated Y_x(which will have been taken from the request queue), the mapping table to allow the physical locations to be found, and the device array itself. The output of the algorithm is the updated device array. This simple algorithm does not take into account wear levelling, as new data is written to the same devices as the original data resided on.

Algorithm 1 Simple Random Update for RAID Array

```
Require: $Y_X : (\mathbb{N} \times Page), MappingTable, Devices$
Ensure: $Devices$
1: Retrieve all other $Y$ stripes using $MappingTable$ from $Devices$
2: Calculate new parity $Y_P$ using new stripe $Y_X$ and all other stripes $Y$
3: Write the updated stripe element $Y_X$ to the original SSD and invalidate the original stripe
4: Write the parity stripe $Y_P$ to the original SSD and invalidate the original parity stripe
5: return
```

Performance of the array suffers in the presence of small random writes combined with garbage collection. RAID controllers normally spread data stripe units evenly across the array. However, if there is an ongoing garbage collection operation in an SSD, the garbage collector blocks the data stripe units to be written to that SSD until garbage collection has completed. For example, if there is an ongoing operation on SSD 1 while data Z arrives in Fig. 3, the writing of stripe Z_1 is delayed until garbage collection has completed while the rest of stripe units for Z can be written immediately.

B. RELATED WORK

RAID-5 typically spreads data and parity evenly across the array. In an SSD environment, when flash memory wears out it leads to the same increased bit error rates on multiple elements at the same time [9]. Diff-RAID [3] addressed this problem. Firstly parity stripes are distributed unevenly across the array and secondly, parity is redistributed when a device is replaced. Other examples considering the wear levelling problem include [10], where wear levelling is optimised to extend the lifespan of consumer products, and [11] which employs a block migration technique to protect blocks known to be at risk of high bit error rate. Reference [4] presents a RAID based SSD storage architecture which significantly improves dependability by using an uneven parity distribution and redistribution technique to manage the reliability levels of each element in the array, and this is further built on in [12], [13]. An approach to calculating reliability dynamics of an array with respect to parity distribution using continuous time Markov chains is given in [14], with results showing that approaches built on [3] exhibit improvements over RAID-5. Although these techniques enhance dependability, they do not meet requirements of real-time systems where guaranteed access time is a necessity. They are also limited in that device replacement is an off-line activity—the use of the array must be suspended while devices are replaced.

Several techniques have been proposed to exploit the internal parallelism of flash based storage, such as [5], [15]. The primary goal of these works is to increase I/O bandwidth with multi channel architectures, enabling the interleaving of data over multiple flash chips. Reference [16] augments this with a hot-cold data identification technique that improves endurance of the system. However, these techniques do not provide mechanisms to protect data in case of failure, or guarantee the upper bound of I/O access latency.

The implementation of a flash management framework in synthesizable Verilog is presented in [17]. It presents a number of techniques including dynamic scheduling, out-of-order execution, and multi chip parallelism to enhance the performance of flash management operations. An extended version of the framework is presented in [18] with further consideration of real-time issues at both hardware and software levels. This framework is used in [19], [20] to develop an on-line device replacement technique that uses hot swapping and thereby eliminates issues associated with taking the away off-line for device replacement.

A number of techniques have been proposed to address the parity update problem for write operations, such as [21]–[24]. Reference [22] uses non-volatile memory (NVRAM) to cache parity to reduce the frequency of parity update operations. Reference [21] presents a RAID-5 SSD architecture with a partial parity technique which reduces parity calculation overhead, and this is further built on in [23]. Elastic parity management techniques that reduce the amount of write traffic resulting in a consequential reduction in garbage collection, are presented in [25]. Reference [24] investigates the effect of the garbage collector over multiple devices in a RAID architecture. To reduce performance variability, the garbage collector is coordinated such that cleaning is triggered on each device simultaneously. However, none of these studies fully address the performance of non-determinism with respect to time when the garbage collector activates.

Various techniques and system architectures have been produced to provide predictable performance and thereby overcome non-deterministic response times. Real-time aspects are first studied in [26], which presents a garbage collector that aims at real-time performance by using a separate thread for each garbage collector task. Reference [27] presents a real-time FTL (called GFTL) that guarantees an upper bound for I/O operations. GFTL adopts a partial block cleaning policy that utilizes extra free blocks to reduce the upper bound of write operations. Reference [8] presents a real-time FTL (called RFTL) that employs a distributed garbage collection policy. Preemptible garbage collection is explored in [28] such that garbage collection can be suspended at certain pre-emption points, and further studied in [29], [30] although these studies focus on pre-emption at device level and do not abstract from individual devices at a global level. While they offer more bounded responses for I/O in the presence of active garbage collection, non-determinism with respect to time is only addressed at chip level, and not on a RAID configuration. Moreover, they only partly mitigate the overhead of garbage collection as the cost of the most expensive part (erase) is not eliminated. Some real-time aspects have been considered in, for instance, [31] at the FTL level, but these exploit application specific characteristics and are not intended for general purpose storage. Reference [32] considers this the context of the asymmetry of read and write operations. While the results show improvements in best case access times, they do not offer the guarantee needed for hard real-time systems, and are vulnerable to interference from garbage collection.

In summary, there is no single complete solution that combines the dependability requirements of SSD RAID, combined with real-time access guarantees, suitable for general purpose storage on enterprise scale. For such a solution to be general purpose and scalable, it needs to be incorporated at FTL level so as to not impose—or implicitly rely on—application level requirements or system configuration.

The contribution of this paper is the presentation of an FTL architecture for SSD RAID arrays, called Garbage Collection Aware Flash Translation Layer (GAFTL) that supports real-time access, improves worst case execution time and I/O performance, and maintains support for the wear levelling problem for dependability. The primary method by which this is achieved is a novel array level garbage collection management mechanism. However, garbage collection management is in itself shown to be insufficient for two reasons. Firstly, it requires consideration of read, write, and erase traffic in order to remove impact on real-time at a fine level of granularity. Secondly, it is shown to have a negative impact on reliability. In order to ameliorate these issues two further techniques are required—one which manages traffic access to the array more closely, and one which reverses the impact on reliability. It is the combination of these three techniques that enable a fully holistic general purpose solution at FTL level.

III. FTL OPERATIONS AND GARBAGE COLLECTION AWARENESS

In this section we introduce and example the write, update, and read operations of the Garbage Collection Aware Flash Translation Layer and explain how they interact both with the garbage collection control mechanism, and with ongoing garbage collection. Section III-A explores real-time random writes, random updates in Section III-B, the base case solution for sequential writes in Section III-C, and read operations in Section III-D.

A. RANDOM WRITES

The nature of a random write request is such that it is guaranteed that there will always be at least one array element (SSD) that will not be involved in the write operation. In this section we present a technique whereby this feature is exploited to eliminate the overhead of interrupting garbage collection processes when undertaking random writes.

The technique is based on page-level stripes. Given a write request from a host system, a RAID controller breaks the data into page stripes, generates parity, and determines the target SSDs. GAFTL then allocates a free physical page location for each stripe unit. Two levels of mapping to access physical location of the data are maintained. Firstly, the RAID controller keeps a stripe mapping table—the link between the logical address of the request and dedicated SSDs—retaining the SSD identifiers that store the data and parity units of a stripe. Secondly, GAFTL records an address mapping table between the logical address and physical location, using a page level address mapping table.

Stripe mapping tables–including SSD allocations–are dynamically created in a non-linear fashion, unlike existing RAID techniques where they are typically allocated in a linear, round robin way regardless of the internal status of the SSDs in the array. Allocations take into consideration any ongoing garbage collection in the array. Address mapping tables are stored in an SRAM memory (where a separate SRAM is used for each flash chip), and NVRAM is used for further metadata storage including the valid, invalid, or free status of all pages on a given SSD.

The generic algorithm for this is given in Algorithm 2. Input to this algorithm is the array of stripes to be written Y—it is assumed that the RAID controller has already created this from the full write request from the host. The invariant property that this array is of the correct size for a random write is also assumed. Furthermore, the invariant property of the garbage collection mechanism presented later in this paper that only one device may be engaged in garbage collection at a time is also assumed. Output of this algorithm is the updated *MappingTable*, and the updated *Devices*. Random writes are categorized into two main groups: new write operations, and

Algorithm 2 GAFTL Random Writes

```
Require: $Y : \mathbb{N} \nrightarrow PAGE, GC\_Array, MappingTable$
Ensure: $Devices, MappingTable$
 1: Generate $Y_P$ using $Y$
 2: Add $Y_P$ to $Y$
 3: for i=0 to #$Y$ do
 4:     Determine a target $SSD_T$ (using the age-aware technique in [4], [12]) for stripe $Y_i$
 5:     Determine a physical page location
 6: end for
 7: for i=0 to #$Y$ do
 8:     if $GC\_Array(SSD_T) = false$ then
 9:         Write $Y_i$ as intended
10:     else
11:         Re-allocate $Y_i$ to the unused SSD
12:         Determine a new physical page location
13:         Write $Y_i$
14:     end if
15: end for
16: Update $MappingTable$
17: return
```

update operations. A new random write refers to a write performed across a free stripe in the array, while an update operation is one which targets fully or partially filled strips with new partial data.

Fig. 4 examples how a normal random write operation for an FTL that is not garbage collection aware works, and Fig. 5 examples the same normal random write for GAFTL, both with respect to time. Two different random write scenarios are depicted. The first write request, data X, arrives where there is a single ongoing garbage collection. The second write request, data Y, arrives and overlaps with two ongoing garbage collection processes. For both of these, we abstract from which stripes are data and parity as this is not relevant.

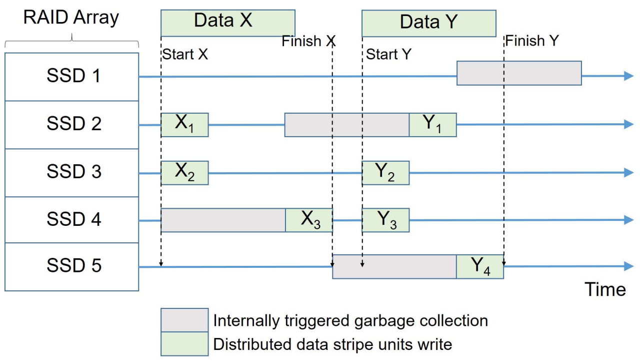

FIGURE 4. Garbage collection unaware mechanism.

In the first case in Fig. 4 the operation to write data X consists of 3 stripe units (X_1, X_2, and X_3) to be written to an array with 5 elements. The response time to complete this write is rather expensive because of the stripe targeting SSD 4 (X_3) as there is a current ongoing garbage collection process. This stripe unit—known as the overlapped stripe

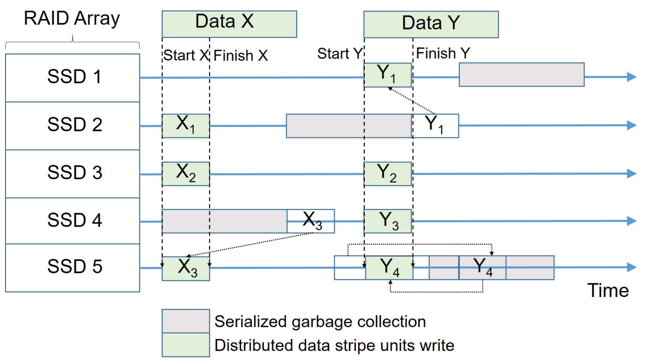

FIGURE 5. Garbage collection aware mechanism.

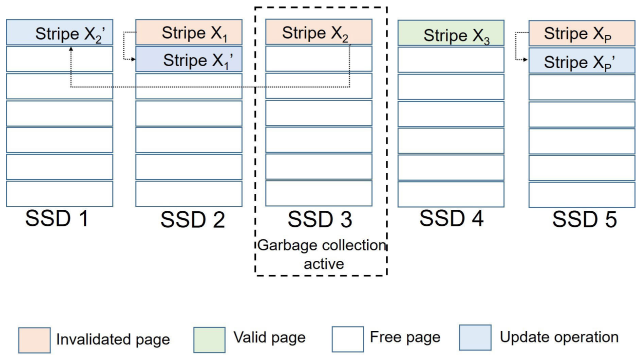

FIGURE 6. Random update for a partial stripe.

unit—is the rate determining step of the operation and the expense is a direct consequence of the lack of communication between the FTL and the RAID controller. This contrasts with the same operation executed on the garbage collection aware mechanism of Fig. 5—the overlapped stripe unit is dynamically reallocated to a free stripe on SSD 5, meaning that it may be written immediately.

Random writes by definition use up to n-1 stripe units in the array (where n represents the number of elements in the array) and consequently dynamic reallocation can only be guaranteed to speed up the writing process if there is a single active garbage collector at any given moment.

This is highlighted in the second case, where a request to write data Y is delayed in Fig. 4 by two garbage collection processes overlapping with stripes Y_1 and Y_4 on SSD 2 and 5 respectively. Clearly, only one of these can be reallocated to a free SSD—and this is shown with Y_1 being reallocated to SSD 1 in Fig. 5. Moreover, the garbage collection process on SSD 5 is forced to delay to an idle time in the future when the garbage collection on SSD 2 has completed—meaning that the writing of Y_4 can commence in parallel with the other stripes, and response time is significantly improved.

B. RANDOM UPDATES

Unlike new random writes, any random update operation on a stripe leads to a new parity recalculation. Random updates may target partially or fully filled logical stripes. GAFTL behaves differently for both cases.

Firstly, a real-time random update for a partial stripe operation is presented. Fig. 6 illustrates the process when the host sends a random update request to stripe unit X_1 and X_2. GAFTL first checks whether or not the target SSDs for the update operation have an active garbage collection operation. In this example, the target device for X_2 (SSD 3) does have an active process. Consequently the mechanism allocates a garbage collection inactive SSD for the update request X_2' instead of waiting for the garbage collection to finish. To assign an SSD for the overlapped stripe unit, the controller checks the stripe mapping table to find an appropriate SSD which does not contain any member of the stripe and the usual update operation is then started. The controller reads X_3 to calculate new parity X_P'. Then it writes the new data X_2' to SSD 1, the new data X_1' to SSD 2 and an updated parity X_P' to SSD 5 while invalidating X_1, X_2 and X_P. Finally, the stripe mapping table is updated accordingly.

To invalidate the overlapped stripe unit without accessing its physical page location, the page status table (stored in NvRAM) is used. Existing FTL techniques usually store status information of a page in its metadata area. To invalidate a page, physical access to the metadata area is required. However GAFTL stores the page status component of the metadata in NvSRAM—thus physical access to the invalidated page is not required.

In this architecture, NvSRAM is partitioned into n sections (where n is the number of SSDs). Two bits in NvSRAM are reserved for each page in the array. The least significant bit represents whether a particular page is a valid or invalid, and the most significant bit indicates if it is free to use or not. The status of X_2 is illustrated before and after the update operation in table Table 1. The update operation for a partial stripe can be employed without any performance overhead.

TABLE 1. Page status table update for random update.

LPN	Chip No	Status before	Status after
100	0	00	10
100	3	10	11

Secondly, a random update operation over a full stripe is exampled in Fig. 7. A scenario is considered where one of the target SSDs of the update request is engaged in garbage

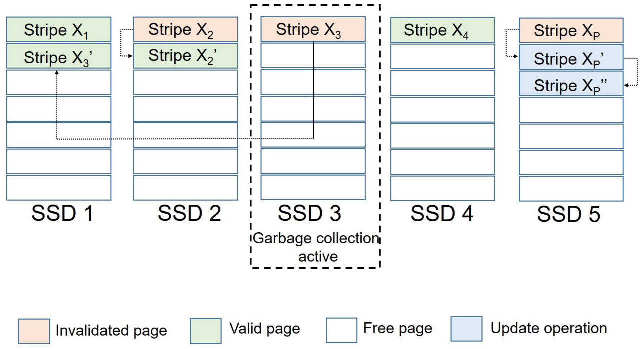

FIGURE 7. Random update for a full stripe.

collection. Moreover, there is no available chip to relocate the overlapped stripe unit as all SSDs in the array already store a member of the stripe. To overcome this, the overlapped stripe behaves as a new random write.

In Fig. 7 garbage collection is active on SSD 3 when the update request arrives. Stripe data X'_2 and the updated parity can be stored on SSD 2 and 5, respectively. As SSD 3 is blocked by ongoing garbage collection, the controller allocates a new SSD to X'_3. Moreover, the overlapped stripe unit (X'_3) is treated as a different random write because one SSD cannot retain more than a single stripe unit for a logical stripe. X'_3 is subsequently placed on SSD 1 and its associated parity is updated accordingly on a different stripe. This operation causes an extra write operation (X''_P) and results in the performance overhead of an update operation. However, compared to a long garbage collection process which includes an erase operation, this has significantly lower impact on system performance and WCET.

The general version of this algorithm is given in Algorithm 3. Input to this algorithm is the stripe to be updated Y_X, where X identifies the device on which that stripe resides—it is assumed in this presentation of the algorithm that this has already been read from the *MappingTable*. As with random writes, invariant properties about garbage collection are assumed. The output of this algorithm is the updated *MappingTable* and the updated array of *Devices*. In the interests of succinctness, this presentation of the algorithm assumes that it is not the device containing the parity that has active garbage collection, although the algorithm could be trivially extended to take this into consideration.

Algorithm 3 GAFTL Random Updates

```
Require: $Y_X$ : $PAGE$, $GC\_Array$, $MappingTable$
Ensure: $Devices$, $MappingTable$
 1: Calculate new parity $X_P$
 2: if $GC\_Array(SSD_X) = true$ then
 3:     if partial stripe then
 4:         Check page status table for a stripe free device $T$
 5:     else
 6:         Determine a target $SSD_T$ (using the age-aware tech-
            nique in [4], [12]) for stripe $Y_X$
 7:     end if
 8: end if
 9: Write $Y_T$ as intended, invalidating original data
10: Update parity $X_P$ and invalidate original
11: Update page status table
12: return
```

C. SEQUENTIAL WRITES

Until now, it has been assumed that a given workload only consists of random writes. A sequential write to an SSD array updates the whole stripe to prevent creating a wear imbalance and thus reducing the reliability of the system. Parity blocks wear at the same rate as data blocks in the case of sequential writes. However, random writes have the ability to imbalance write traffic across the SSD elements in the array. This provides an opportunity to differentiate the age of the SSDs in the array to prevent simultaneous wearing. It can be noted that maximum reliability is provided with a workload which only consists of random writes [3]. It is also known that many realistic workloads are dominated by random writes [33].

Section III-A presented the GAFTL technique for guaranteeing response times for random writes. In order to utilize the benefits of these, [34] presents a technique for converting a sequential write into a a random one. GAFTL expands on this technique by relocating partitioned random writes on to devices not currently being garbage collected.

A standard sequential write for X is exampled in Fig. 8b, a forced random write using the existing technique in [34] in Fig. 8c, and a new garbage collection aware forced random write in Fig. 8d. Each of these correspond to the array in Fig. 8a, and the keys for invalid stripes, stripe writes, and garbage collection pertain to each of the sub figures.

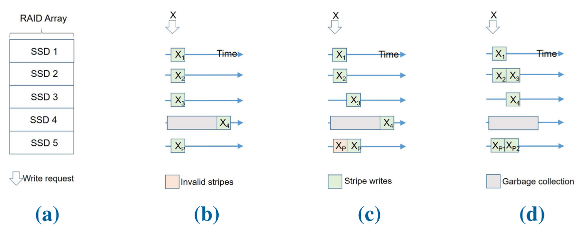

FIGURE 8. Example sequential and random stripe writes.

The aim of the forced random write technique in [34] is to create wear imbalance among the chips in the array for new and update data requests. As illustrated in Fig. 8c, there is an extra write operation to the parity SSD (SSD 5) compared to the standard sequential write in Fig. 8b. In this approach, it is assumed that the sequential write is divided into two random writes. First, two data stripes (X_1 and X_2) and a partial parity data (X_P) are written to the same stripe. Second, data X_3 and X_4 are written with full stripe parity (X_P) of X_1, X_2, X_3 and X_4 while invalidating the original partial parity.

Although this forced random write technique offers enhanced reliability across the RAID array compared to a sequential write, it still suffers from non-deterministic access latency due to ongoing garbage collection—such as that in SSD 4. Fig. 8d examples how GAFTL ameliorates this issue. As it has knowledge of the ongoing garbage collection process, the overlapping component of the stripe (X_4) is reallocated to a garbage collection free device.

In Fig. 8d, SSD 2 stores multiple data stripes belonging to the same stripe. In order to reconstruct X in the case of failure in SSD 2, the mechanism needs to keep two partial parities (X_{P1} and X_{P2}) for stripe X. Also, all indexing information is updated in the stripe mapping table. Thus the garbage collection aware forced random writes not only increases system reliability, but also provides guaranteed access times for sequential write workloads.

Algorithm 4 informally describes this. The algorithm exploits, and is presented in terms of, the previously described operations. Input to the algorithm is the full stripe to be written (Y), the garbage collection array, and the page mapping tables. It is first split into two random stripe writes. Each of these random stripes are then written using the random write technique of Algorithm 2, which takes into account any ongoing garbage collection. Once these have been written, the full stripe parity is written and updated for the whole stripe using the random update operation of Algorithm 3 and details committed to the page mapping table.

```
Algorithm 4 GAFTL Sequential Writes
Require: $Y : \mathbb{N} \nrightarrow PAGE, GC\_Array, MappingTable$
Ensure: $Devices, MappingTable$
1: Split $Y$ into random write stripes $Y_{R1}, Y_{R2}$ using [34]
2: Perform Algorithm 2 on $Y_{R1}$
3: Perform Algorithm 2 on $Y_{R2}$
4: Calculate full stripe parity $Y_P$
5: Perform Algorithm 3 on $Y_P$
6: Update page mapping table
7: return
```

D. READ OPERATIONS

Parity based RAID techniques have the ability to recover data in the instance of a single device failure. GAFTL also utilizes the redundant data in a RAID array to calculate overlapped stripe units. If a read operation needs to access a device where the garbage collector is active, the data is marked as a failed data access by the controller. When the host sends a read request to the SSD array, GAFTL follows Algorithm 5. When the host asks for data at a given logical address, physical locations are identified using the mapping table. If none of the physical locations have a target SSD currently involved in garbage collection, the data can be read and returned. If however one of the locations does have ongoing garbage collection, then all other data is read and the missing stripe calculated using parity. As with the other algorithms, the invariant property that only one SSD may undergo garbage collection at a time is assumed.

```
Algorithm 5 GAFTL Read
Require: $X : LogicalAddress, MappingTable, GC\_Array$
Ensure: $Data : \mathbb{N} \nrightarrow Bit,$
1: Read $MappingTable$ to identify SSDs to be read
2: if a target SSD has ongoing garbage collection then
3:     Read all available (non-GC) stripes
4:     Calculate missing stripe data using parity
5: else
6:     Read all stripes
7: end if
8: Return Data
9: return
```

For a real-time read operation, the mechanism simply avoids sending a command to the overlapped stripe unit thereby avoiding the time overhead in dealing with a failed read request until garbage collection completes. For instance, Fig. 9 examples the scenario where a read request for X is submitted whilst there is ongoing garbage collection on SSD 3. Normally, to read the data from SSD 3, the request has to wait until this garbage collection completes. However the mechanism is aware of this ongoing garbage collection and consequently does not send a read operation to this SSD. Instead, it reads the rest of the stripe units (X_1, X_2, X_4) including the parity data (X_P) and calculates the missing data X_3 using a standard XOR operation. This operation has only the calculation overhead, which is negligible. As seen from the example, the mechanism offers higher performance and guaranteed access time for a read operation even there is a single ongoing garbage collection in the array. This technique is applicable to both random and sequential read operations—although GAFTL ensures that all read operations will be random reads due to the random write enforcement.

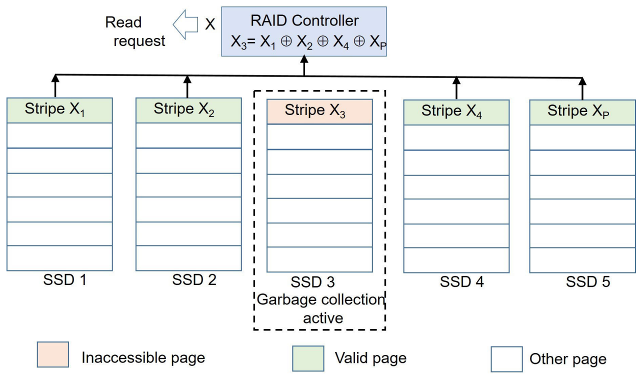

FIGURE 9. Real-time read operation.

All of the algorithms presented in this paper so far have assumed, and rely on, the invariant property that there is only one garbage collector active in the array at any one point in time. If this assumption is relaxed, the operations do not work in the general case—and this assumption is necessarily relaxed in the case of general, non-GAFTL RAID arrays. In the next section, we present a technique for serialized garbage collection whereby this invariant property can be relied on, thus preserving the integrity of the assumption.

IV. SERIALIZED GARBAGE COLLECTION

Unmanaged garbage collection has a significant impact on response time. In this section we present a serialized garbage collection technique that can be implemented as a RAID array FTL level (as in, for instance [34]) operation to achieve guaranteed performance. The technique ensures that there is only one garbage collector active in the array at any one time (under non-exceptional operating conditions), thereby ensuring the invariant assumptions of the read, write, and update operations presented in Section III. This serialisation of garbage collection is achieved by globally scheduling

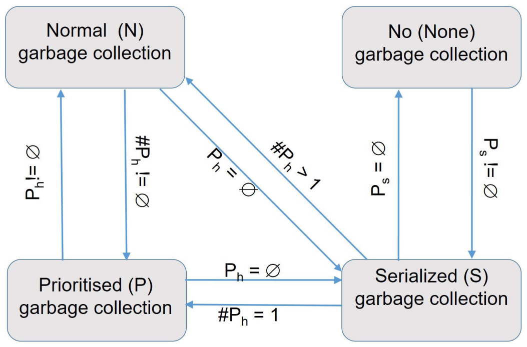

FIGURE 10. **Serialized garbage collector state diagram.**

the garbage collectors across the array. Whilst the combination of parity data and global scheduling eliminates non-deterministic access times, the consequential challenge is to ensure that no singe device fails due to a lack of free pages resulting from garbage collection delays. To solve this problem, the global garbage collection management is divided into several modes of operation, described by the state diagram in Fig. 10 and Algorithm 6 and as follows:

- No garbage collection (*None*): There is no active garbage collection enabled on any device.
- Serialized garbage collection (S): A single garbage collector may be activated on any one device.
- Prioritised garbage collection (P): Serialized garbage collection is enabled but another device has higher priority for cleaning.
- Normal (conventional) garbage collection (N): Any number of garbage collectors may be active in the array at the same time. This corresponds to an exceptional circumstance in GAFTL.

Conventional garbage collectors typically have two states: either there is no garbage collection going on, or unmanaged garbage collection across the array is permitted—this corresponds to the normal state (N) in Fig. 10. Each SSD monitors the number of free blocks it has available. If unmanaged garbage collection is permitted and the number of free blocks exceeds a pre-defined threshold then garbage collection commences. Alternatively, if an idle time period is detected, garbage collection may commence also.

A basic informal definition of the data structures used in the serialised garbage collector is given in Data Structure 4. In the serialised garbage collection model, thresholds are monitored and used to identify which devices may require urgent or non-urgent cleaning, and also which mode of cleaning is necessary. This is depicted in the `for` loop of Algorithm 6. The number of free blocks on each device is inspected in turn, and the device identifier placed either in the urgent (hard) pool P_h, the non-urgent (soft) pool P_s, or no pool depending on where they lie in relation to the parameterised pre-set system threshold hard and soft levels T_h and T_s.

Data structure 4 Data Types for Serialised Garbage Collection

```
1: $States : None|N|S|P$
2: $P_s, P_h : \{x : \mathbb{N} | x < \#Devices\}$
3: $T_h, T_s : \mathbb{N}$
4: $State : States$
```

Algorithm 6 GAFTL Serialised Garbage Collection

```
Require: $State, Devices, P_h, P_s, T_h, T_s$
 1: while true do
 2:     for i= 0 to # Devices-1 do
 3:         switch $free\_blocks(SSD_i)$
 4:         case $> T_s \wedge < T_h$: $P'_s = P_s \cup \{i\} \wedge P_h = P_h \setminus \{i\}$
 5:         case $> T_h$: $P'_h = P_h \cup \{i\} \wedge P_s = P_s \setminus \{i\}$
 6:         default: $P'_s = P_s \setminus \{i\} \wedge P_h = P_h \setminus \{i\}$
 7:         end switch
 8:     end for
 9:
10:     switch $State, P_h, P_s$
11:     case $None \wedge P_s! = \emptyset$: $State := S$
12:     case $N \wedge P_h \neq \emptyset$: $State := P$
13:     case $N \wedge P_h = \emptyset$: $State := S$
14:     case $P \wedge P_h \neq \emptyset$: $State := N$
15:     case $P \wedge P_h = \emptyset$: $State := S$
16:     case $S \wedge \#P_h > 1$: $State := N$
17:     case $S \wedge \#P_h = 1$: $State := P$
18:     case $S \wedge P_s = \emptyset$: $State := None$
19:     default: No action needs taken
20:     end switch
21: end while
22: return
```

The second `switch` statement in Algorithm 6 controls the state machine and consequently garbage collection activity. During idle periods when there is no garbage collection required, the system defaults to the *None* state—indicating that there is no active garbage collection underway. If, in this state, one or more devices are placed in the soft pool (clause $None \wedge P \neq \emptyset$), then the garbage collector may initiated on one of the devices chosen from the pool at random, and the system moves into the serialised state. In this state, it selects one of the garbage collectors in this pool at random and activates it. Upon garbage collection competing, it removes it from the pool. When there are no SSDs left in the soft pool (clause $S \wedge P = \emptyset$) the system reverts back into the no garbage collection state.

A common situation is that there may be more than one SSD in the pool requiring cleaning at any one time, and that one of the SSDs requiring cleaning is more urgent than the others. When a hard threshold is reached on a given SSD, this indicates that the level of free space is critically low and cleaning is urgent. This corresponds to the $> T_h$ clause in Algorithm 6—when a device exceeds this threshold it is placed into the urgent pool. It is necessary to monitor the possibility of urgency, as the selection of devices for cleaning from the soft pool is non-deterministic and so any given device may escape cleaning for prolonged periods.

If there is only SSD in the pool that has reached the hard threshold (clause $S \wedge \#P_h = 1$) the system moves into the prioritised garbage collection state. The chip is prioritised and cleaned, before the system removes it from the pool and returns to the serialized state. This mechanism is useful because it is common in an SSD RAID array for one chip to require more cleaning more frequently—for instance if it carries a higher parity data ratio due to age ratio management.

A final extraordinary situation which may arise is where multiple SSDs simultaneously exceed their hard thresholds (clause $S \wedge \#P_h > 1$). In this state the system moves into normal (conventional) garbage collection mode. In this mode multiple garbage collectors may run simultaneously as happens in conventional techniques. After sufficient free pages have been released, the system will revert either to the serialized, or the prioritised, mode dependant upon the number of SSDs exceeding soft and hard thresholds.

Clearly, in the normal (conventional) garbage collection mode real-time guarantees are not possible—however this mode is typically a backstop against pending system failure due to insufficient free space, rather than a mode employed in normal operation as the age ratio mechanism of [12], combined with an appropriate chose of hard threshold T_h, typically ensures that the most aged device is the only one that approaches its hard threshold. The modes show that, whilst the property of no more than one garbage collector being active is not truly invariant, it can be regarded as such in anything other than a failure recovery mode—in which real-time guarantees would not be expected to be upheld.

WCET ANALYSES

In Section II we presented how traditional architectures and garbage collection processes are poor at offering real-time guarantees due to uncertain latencies caused by unmanaged garbage collection. In this section we analyse the improved Worst Case Execution Time over existing solutions, and show how these prevent performance degradation. In this analysis, T_{er} refers to the time taken to erase a block, U_w the upper bound of a page write, and U_r the upper bound of a page read, terminology taken from existing literature [8], [27], [28]. Actual values for the upper bounds depend on various factors such as the type of address mapping table used, the method by which metadata is stored, and the existence or otherwise of buffering. Consequently our analysis considers relative, rather than absolute, bounds.

WCETs for existing techniques are compared in Table 2. Two modes require discussion—where GAFTL is in either serialized or prioritised modes, and where it may be in normal garbage collection mode. Firstly, if it moves into normal mode then WCET times would be increased—however as discussed in Section IV this serves only as a backstop mode to prevent system failure and so is not directly considered for GAFTL. The techniques of [8], [27], [28] do not take this into account as they have only one mode. Secondly it should be noted that previous FTL mechanisms only present solutions at single SSD level, not considering the whole RAID array.

Typically, erase operations (T_{er}) are around ten times more expensive than write operations (U_w), and write operations are of broadly equivalent cost to read operations (U_r). The underlying principle behind existing techniques is for the garbage collector to pre-empt instruction streams. As it can only pre-empt at the granularity of the operations (instructions), none of the existing solutions avoid the long latency of an erase operation. In the worst case scenario, a request may be received immediately after an erase operation has been initiated and thereby have to wait for it to finish.

GAFLT is seen to significantly reduce WCET by ensuring that no instructions are blocked by ongoing garbage collection. All requests can be directly serviced with a cost of (the maximum of) either U_w or U_r.

TABLE 2. **WCET comparison.**

Technique	WCET
GFTL [27]	$\mathrm{T}_{er} + max(\mathrm{U}_w, \mathrm{U}_r)$
RFTL [8]	$max\ (\mathrm{T}_{er} + (\mathrm{U}_w, \mathrm{U}_r))$
PGC [28]	$\mathrm{T}_{er} + max(\mathrm{U}_w, \mathrm{U}_r)$
GAFTL	$max(\mathrm{U}_w,\mathrm{U}_r)$

V. ON-LINE PARITY MIGRATION

While the serialized garbage collection mechanism presented in Section IV offers performance guarantees for incoming requests, it is less well suited to existing reliability mechanisms where SSD ages are strictly controlled by pre-assigned parity distribution ratios. This is due to the differing behaviour of the mechanism in the case where garbage collection is inactive or active. In the case where it is inactive SSD destinations for writes do not change. However in the case where is it active, different destinations are selected. Over time, this results in a degeneration of the parity data distribution from the pre-assigned ratio. As the most aged SSD performs more garbage collection than other devices in the array it's default parity data percentage decreases whilst the others increases.

This has a negative impact on reliability. In this section, we present a new mechanism that controls parity data distribution in this scenario. The mechanism enables the on-line proactive migration of parity data while the system is running so that desired ratios can remain adhered to (or be intentionally altered in special cases).

Data structure 5 Data Structures for On-Line Parity Migration

```
1: $Percentage : \{x : \mathbb{N} | 0 \leq x \leq 100\}$
2: $Migration : \mathbb{B}$
3: $Assign : \mathbb{N} \nrightarrow Percentage$
4: $Dest : \mathbb{N} \nrightarrow SSD\_Device$
5: $Mode : 1..4$
6: $OPM : GC\_array \times Migration \times Assign \times Dest$
```

An invariant property of *GC_array*—that garbage collection may be active on at most one device at a time—is assumed. Data structures used for on-line parity migration are presented informally in Data Structure 5, and are as follows:

- *Migration*: a boolean variable indicating whether or not parity migration is currently active in the array.
- *Assign*: a sequence of percentages, indicating the share of parity currently assigned to each device in the array. This would normally carry the invariant property that the sum of the percentages should be 100.
- *Dest*: a sequence of devices, indicating the destinations for incoming parity write requests. The device at position n in this sequence indicates the device that incoming parity write requests for $Devices(n)$ (Data Structure 3) should actually be written (migrated) to.
- *OPM*: a tuple consisting of *GC_array* (Data Structure 3), *Migration*, *Assign*, and *Dest* used to manage the on-line parity migration and the different states.

A simple state diagram for the parity migration is presented in Fig. 11, where each state (mode) is characterised by various different states of the tuple *OPM*. Each mode may be summarised as follows:

- Mode 1: Both garbage collection and parity migration are inactive. Parity distribution ratios are not changed until an SSD replacement is required—implemented using the on-line device replacement techniques of [20]. This mode corresponds to a value of *OPM* where the range of *GC_array* is *false*, *Migration* is *false*, the range of *Assign* is the initially chosen values (using the technique of [12]) and no identifier in *Dest* points to any other device (that is, $Dest(n) = Devices(n)$).
- Mode 2: Garbage collection is active on the most aged device in the array ($GC\_array(0) = true$), but migration remains inactive ($Migration = false$). Any parity writes to this device are redirected to the second most aged device in the array ($Dest(1) = SSD\_Device(2)$). Parity assignment ratios (*Assign*) remain unchanged.
- Mode 3: Garbage collection is active on a device in the array other than the most aged device ($\exists_1\ n : 1..\#GC\_array | GC\_array(n) = true$), but migration remains inactive ($Migration = false$). Any parity writes directed towards this SSD are redirected to SSD 1 ($Dest(n) = Devices(0)$). Parity distribution ratios remain unchanged.
- Mode 4 On-line parity migration: Garbage collection is disabled across the array ($GC\_array = false$), and migration is active ($Migration = true$). Initial parity distribution ratios (*Assign*) may be reassigned if on-line device replacement requires it. Existing (pre-written) parity data is moved around the array to rebalance parity distribution according to the ratios in *Assign*.

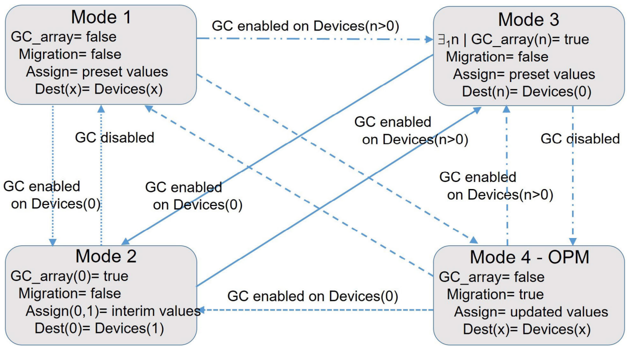

FIGURE 11. Parity migration states.

The state transitions are described informally in Algorithm 7. On entering mode 1, which is effectively an idle mode, *Assign* is reset to reflect the original (or recalculated if through mode 4) parity distribution levels, and the destinations for incoming parity writes is reset to the expected destinations. These have been left out of the description in Algorithm 7 in the interests of succinctness, but are reflected in Fig. 11. Exit transitions depend on the state of the garbage collector. If garbage collection is enabled on the most aged device (SSD 1) then the exit transition is to mode 2, if it is enabled on any other device the exit transition is to mode 3. If there is no enabled garbage collection, then the pro-active parity migration of mode 4 may be entered.

Algorithm 7 On-Line Parity Migration

```
Require: $Mode$, $OPM$
while true do
    switch (Mode)
    case 1:
        $OPM$ := preset values
        if $GC\_array(0) = true$ then $Mode := 2$
        else if $GC\_array(n > 0) = true$ then $Mode := 3$
        else $Mode := 4$ endif
    case 2:
        $Assign(0, 1)$ := interim values
        $Dest(0) := Devices(1)$
        if $GC\_array(n > 0) = true$ then $Mode := 3$
        else $Mode := 1$ endif
    case 3:
        $Dest(n) := Devices(0)$
        if $GC\_array(0) = true$ then $Mode := 2$
        else $Mode := 4$ endif
    case 4:
        $Migration := true$
        while $GC\_array = false \wedge$ migration needed do
            if idle time then migrate a parity block endif
        end while
        if $GC\_array(0) = true$ then $Mode := 2$
        else if $GC\_array(n > 0) = true$ then $Mode := 3$
        else $Mode := 1$ endif
    end switch
end while
return
```

In mode 2, incoming parity writes to SSD 1 are redirected to SSD 2. This is achieved by assigning interim values to both *Assign* and *Dest* on entry. SSD 2 is assigned the interim value of its parity distribution and that of SSD 1, while SSD 1 is assigned an interim distribution of 0%. The destination for parity writes intended for SSD 1 is redirected to SSD 2. When garbage collection completes, the mechanism leaves this mode. If there is a pending enabled garbage collection for another device in the array the transition is to mode 3, otherwise it returns to mode 1.

In mode 3, all incoming parity data targeting the device being cleaned is redirected to the most aged device (SSD 1) using an interim value for $Dest(n)$. Parity distribution ratios for other devices remain unchanged from their preset values. As this may have caused an imbalance in parity distribution that affects ageing characteristics, an exit transition directly back to mode 1 is not permitted, and exit may go through the proactive migration of mode 4 if there is no pending garbage collection on the most aged device.

On entering mode 4, the mechanism raises the *Migration* flag to indicate that pro-active parity migration is underway in order to allow interaction with idle time detection mechanisms. The amount of parity data needing migrated, the devices from which it needs to be migrated, and the devices to which it can be migrated, is calculated. Parity data is then moved from the over-subscribed devices to the under subscribed devices when idle times in the data stream are detected. This move may also involve a stripe swap operation between the two devices if the destination contains a member of the same logical stripe.

VI. EXPERIMENTATION AND RESULTS

In this section we present the results of a number of experiments performed on the garbage collection aware FTL, including comparisons with a non garbage collection FTL, and the pre-emptive garbage collection mechanism. Performance evaluation is performed using the Microsoft SSD simulator [5], which is in turn derived from Disksim [35]. The parity distribution mechanism is implemented in the simulator, and the RAID controller is configured as the garbage collection aware FTL. NAND flash SSD is simulated, with the performance characteristics given in Table 3. Source code for the amended (GAFTL enabled) version of the SSD simulator and FTL may be made available upon request.

TABLE 3. Array parameters.

Parameter	Value
Reserved free blocks	15%
Minimum free block	5%
Flash chip elements	1
Blocks per plane	2048
Pages per block	64
Page size	4kB
Page read latency	4kB
Page write latency	0.200 ms
Block erase latency	1.5 ms
Redundancy Schema	Taken from [4]
Stripe unit size	4kB

Under normal workload—where there are not simultaneous garbage collection requests—the serialized mode (shift operations) is not enabled. When multiple garbage collections are simultaneously requested, serialized mode is enabled and the controller checks the garbage collection pool for page consuming requests. If there are requests for multiple chips in the pool the controller schedules all incoming garbage collection requests according to priority levels. This is repeated over all chips in the request pool until a chip has enough free blocks. When the pool is empty, the system goes back to a non garbage collecting state.

Results show an evaluation and comparison with several different approaches: a non garbage collection aware FTL, the garbage collection aware mechanism of Section III, and the pre-emptive mechanism of [28]. By default, the simulator uses the basic normal, uncoordinated approach where garbage collection is initiated based on the internal status of a flash chip with no global coordination. Experiments report results of a mixture synthetic workloads (Section VI-A) and realistic workloads taken from actual traces (Section VI-B). For synthetic workloads the probability of sequential access, inter-arrival time of requests, and size of requests are all varied. Memory is initially filled with valid data. The initial parameters for synthetic traces are given in Table 4.

TABLE 4. Synthetic trace initial parameters.

Parameter	Value
Request Size	4kB
Inter-arrival Time	3ms
Probability of read access	0.2
Probability of sequential access	0.2

A. SYNTHETIC WORKLOADS

Each of the graphs in this section (Fig. 12, Fig. 13, Fig. 14, and Fig. 15) show results taken both with the serialized garbage collection FTL of Section IV and of an FTL unaware of garbage collection. Workloads are synthetic: that is to say, we use workload traces where we vary several parameters across a normal operating range. Under the serialized garbage collection FTL, the FTL dynamically reallocates requests across available elements of the array subject to active garbage collection; in the garbage collection unaware FTL they are naively allocated as the FTL.

Fig. 12 shows the results of varying the request size across 4, 8, and 10kB. These values are used as the reliability enhancement mechanism achieves maximum efficiency using partial stripe writes that enable age imbalancing. For a small request size (4kB), GAFTL response time is approximately three times faster than the garbage collection unaware equivalent. As the request size increases, the response performance of the garbage collection unaware mechanism drops off very significantly—response times for 10kB requests are approximately four times slower than than for 4kB requests. however GAFTL does not drop off rapidly and so the

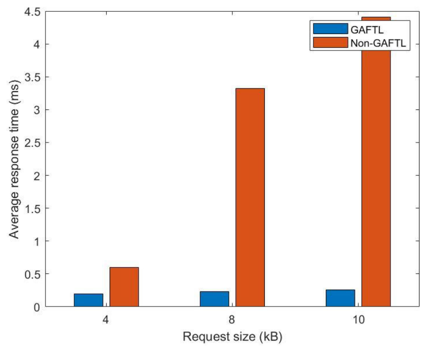

FIGURE 12. Varying request size (synthetic traces).

performance improvement for a 10kB request is approximately 10 fold. This is because the mechanism is not delayed by the number of cleaning processes active during the request.

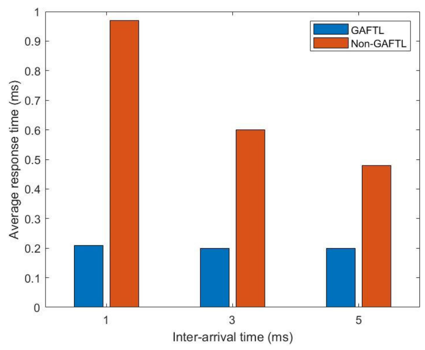

FIGURE 13. Varying inter-arrival time (synthetic traces).

Fig. 13 shows similar improvements with respect to varying the arrival rate of I/O requests over 1, 3, and 5ms. For the non garbage collection aware FTL the response time is rather slow for a rapid inter-arrival time of 1ms, but it speeds up as arrival time intervals increase. This is because with rapid inter-arrival times the probability of conflict with garbage collection is high, and as arrival times slow the likelihood of a conflict with an active garbage collection process decreases. However they do not decrease to a level where the effect is negligible. For GAFTL, response times remain effectively constant across the range—and notably remain significantly below that of the conventional mechanism.

Fig. 14 shows the results of varying the ratio of read requests to write requests. This is an important parameter for two reasons. Firstly, the cost in terms of time for a read is significantly less than for that of a write operation. Secondly, as the number of write operations increases, so does the likelihood of need for garbage collection. For the non garbage collection aware FTL, access times increase as the probability of any given access being a read access increases, almost linearly. The same relative (linear) improvement in GAFTL may also be observed—but the response times are significantly faster. For instance, when the probability of any given access being a read access is 0.3, GAFTL responds three times faster.

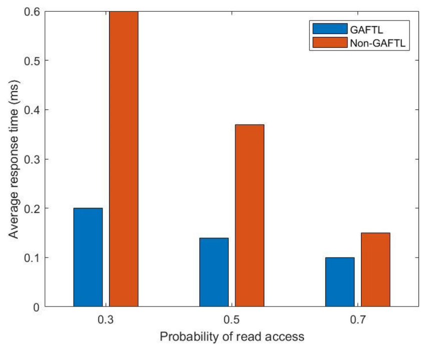

FIGURE 14. Varying read ratio (synthetic traces).

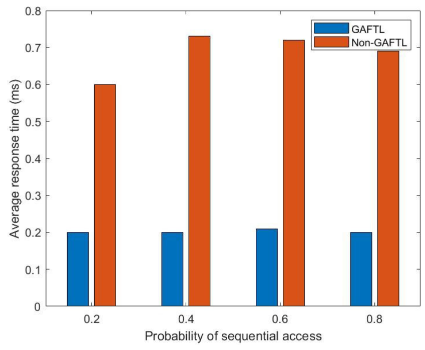

FIGURE 15. Varying sequential access probability (synthetic traces).

Fig. 15 shows the results of varying the probability of a write access being a sequential write access. This is an important measure as the reliability enhancement mechanism in GAFTL converts all sequential write accesses to random write accesses—random write accesses generally lead to more parity update operations, and therefore more invalid pages, and therefore more garbage collection. The probability of an access being sequential is varied at 20% intervals. The average response time for the non-garbage collection remains high throughout, with a peak around 50%. The average response time for GAFTL remains both constant, and significantly lower. The consistency may be expected due to the fact that all accesses in GAFTL are converted to random write accesses. The significantly lower repose times can be attributed to the successful management of interactions with garbage collection requests.

Overall, these results show significant performance improvements for GAFTL under varying synthetic workloads, regardless of workload characteristics. Results are both more consistent than for a non garbage collection aware mechanism, and also more performant, generally being significantly faster—both are very important in a real-time context. Moreover, the benefits of GAFTL become more pronounced for characteristics that are more likely to lead to cleaning operations (smaller request sizes, faster inter-arrival times, more random accesses. and fewer read accesses).

B. REALISTIC WORKLOADS

In this section, the same set of experiments are reported, but with realistic (real captured) workloads. Traces were captured using the Flashmoon tool [36]. Characteristic features of the traces (both read and write dominant) are given in Table 5. The Postmark benchmark represents the behaviour of a general purpose file system, is write dominant, fast inter-arrival times, and typically generates small request sizes. The Boot benchmark represents the behaviour exhibited during a kernel boot process. It is read dominant, and instructions have a reasonably spaced inter-arrival rate.

TABLE 5. Characteristics of realistic traces.

Workload	Read %	Inter-arrival time	Required size
Postmark	30	0.92	4 kB
Boot	92	3.32	4 kB

In the keys for the graphs that follow, *A* represents the garbage collection aware mechanism presented in this paper, *B* represents the preemptible mechanism of [28], and *C* represents a standard non garbage collection aware mechanism.

Fig. 16 shows average response times for Postmark and Boot workloads. Average response time is improved by nearly 13 times compared to a standard non garbage collection aware mechanism in the Postmark workload. A small performance improvement is achieved over the pre-emptive garbage control mechanism. For the Boot workload, the garbage collection aware mechanism exhibits a small (2.32%) improvement over the pre-emptive mechanism, and a more significant (26%) improvement over the non garbage collection aware mechanism.

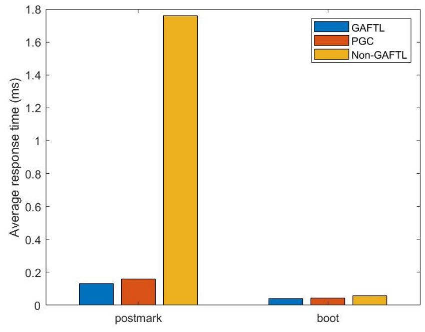

FIGURE 16. Average response times (realistic traces).

Maximum response times are presented in Fig. 17, with results normalized to the slowest response. The garbage collection aware mechanism exhibits 62% and 73% faster maximum response times over the pre-emptive mechanism for both workload types. Improvement over over the traditional mechanism is several orders of magnitude for both workload types—because GAFTL eliminates long cleaning processes by dynamically disturbing data stripe units.

C. RELIABILITY ANALYSES

In this section we investigate the effects of the serialized garbage collection mechanism, and the on-line parity migration, on the reliability mechanisms. As presented in [4], the reliability mechanism manages ageing of devices by controlling the percentages of total parity data written to each device. Individual device age is calculated using the ageing formula presented in [3]. Experiments were conducted using a write dominant synthetic trace in the MSR SSD simulator [5]. SSD model parameters are given in Table 3, and the synthetic trace characteristics in Table 4.

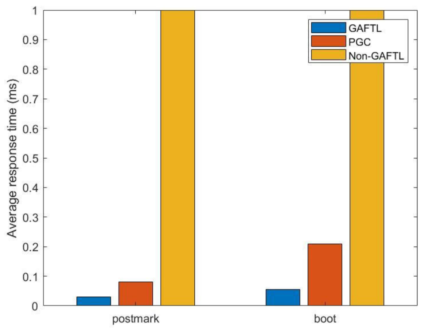

FIGURE 17. Maximum response times (realistic traces).

The reliability mechanism is not dependant upon read or write dominant traces and so in these experiments I/O request sizes were kept under a 16kB stripe size. Fig. 18 shows results of device ageing over the fist ten device replacements. In this experiment, when the most aged device reaches its normalized endurance limit (100% of cycles) for the first device replacement point it contains 26% less parity data than expected. As the device is to be replaced, this parity data is migrated to the second most aged device in the array which then contains approximately 40% more parity than its optimum level before optimum levels are shifted.

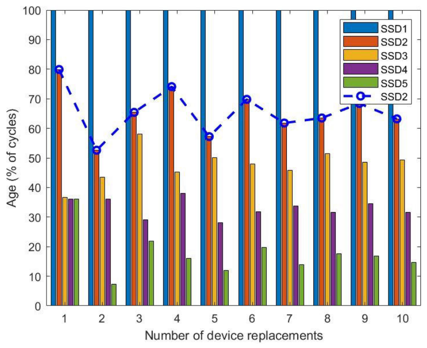

FIGURE 18. Age distributions under GAFTL.

During the first replacement cycle the (normalised) age of the second most aged device is 80%—meaning probability of data loss is raised. The age of the second most aged device continues to fluctuate through subsequent replacement cycles—exceeding critical ageing ratios at replacement points 1, 3, 4, 6, and 9. This is in contrast to the basic mechanism presented in [4], where the age of the second device stabilises after the third replacement cycle.

Fig. 19 shows the results of the same experiment, using the same synthetic trace, with on-line parity migration enabled. These results show that on-line parity migration has significantly reduced the negative effects on ageing stability caused by the garbage collection aware mechanism. Parity distribu-

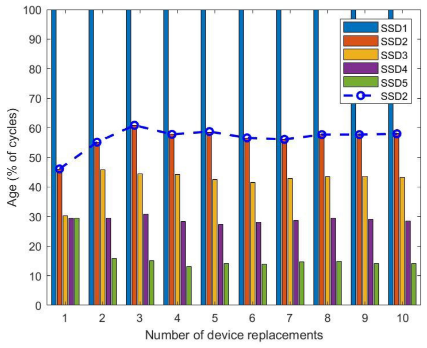

FIGURE 19. Age distributions under GAFTL and OPM.

tion is kept at optimum levels and stabilises from replacement cycle 3, thus preserving the reliability offered by the basic mechanism of [4].

VII. CONCLUSION AND DISCUSSION

Limitations of SSD RAID are well understood both in published scientific literature, and in application environments such as manufacturers, end users, and internet forums. The increasing use of large scale SSD RAID in enterprise architectures has motivated the requirement for these limitations to be addressed. Whilst there has been advances in reliability mechanisms in recent years, the challenge of highly deterministic real-time support has remained.

Consumer demand for large, enterprise server storage facilities that offer measurable service levels has grown massively in recent years, and there are a number of products that serve this market. Whilst these products offer configurable service level guarantees at the applications (or file system) level, the guarantees are typically not at the level of real-time support at FTL or device level. Consequently, as the size of enterprise storage facilities grows, and application demands grow, it becomes increasingly challenging to maintain these guarantees. Moreover, the lack of support for real-time guarantees at the device level has meant that the use of these architectures in high-assurance and safety-critical applications has presented significant barriers.

In this study new techniques for a high reliability SSD RAID array that ameliorates these challenges by addressing them at the FTL and RAID level have been presented. The approach taken in this study was firstly to raise physical and logical aspects of read, write, and erase operations globally to the FTL. Secondly, it was to raise the oversight of garbage collection globally. The combination of these meant that it was possible to manage read, write, and erase operations to take garbage collection into consideration—resulting in deterministic response times.

A drawback of this approach is the adverse effect on the data level reliability—specifically, the wear-levelling that is required for SSD RAID reliability. This drawback was addressed by introducing an on-line parity migration system that redressed unintended wear imbalance by moving parity when required, during idle periods.

By considering the challenge in the context of unmanaged garbage collection across the array, the study illustrated that results in terms of real-time performance, and storage reliability, are viable, performant, deterministic, and an enhancement on state-of-the-art. It also illustrated that this can be achieved without need for changes at a physical device level or at a high level file system level, and that it can be achieved with software enhancements at FTL and RAID level.

Techniques were investigated, and experiments performed, using an industry standard trace driven simulator. Experimental results validate the hypothesis that the combination of these techniques result in deterministic performance with respect to time, without deterioration in terms of performance in terms of reliability. Response times exhibit near-order of magnitude improvement over unmanaged techniques, as well as consistency under varying request size, read ratio, inter-arrival time, and sequential access probability. Results are consistent across both standard synthetic and realistic workload experimental data.

A property of these techniques is that they all rely on the assumption (invariant) property that only one garbage collector is active at any one time. This was enforced in three modes of operation. Not only do the results evidence that this is not a hindrance, they demonstrate that it is exactly enforcing this property that unlock the performance improvements. A fourth mode relaxes this assumption. In this fourth mode (equivalent to current state-of-the-art) garbage collection is unmanaged. The experimental data did not cause this mode to be entered, leading to the conclusion that it may be useful as a recovery mode in the case of extraordinary circumstances, and the further hypothesis that it may be completely unnecessary in the general case.

A future goal for this work is to incorporate it into our FPGA-based Verilog hardware-in-the-loop RAID controller and FTL of [18], such that results may be correlated with physical devices. This would enable further insight into real-time properties—including the standard deviation of read access times—and scalability to enterprise level.

An additional goal for this work is to produce a fully functional, real-time, verified SSD RAID file system suitable for safety-critical environments. In order to achieve this, the informal descriptions presented in this paper could be captured accurately in a full formal, timed model of the FTL RAID implementation using timed automata or timed process algebra such as [37]–[40]. This would mean safety-critical and real-time properties, investigated empirically through experimentation, could be formally verified also—both at the file system level, and at the environment (application) level.

REFERENCES

[1] S. J. Kwon, A. Ranjitkar, Y.-B. Ko, and T.-S. Chung, "FTL algorithms for NAND-type flash memories," *Des. Automat. Embedded Syst.*, vol. 15, nos. 3–4, pp. 191–224, Dec. 2011. doi: 10.1007/s10617-011-9071-9.

[2] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Comput. Surv.*, vol. 26, no. 2, pp. 145–185, Jun. 1994. doi: 10.1145/176979.176981.

[3] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential RAID: Rethinking RAID for SSD reliability," *ACM Trans. Storage*, vol. 6, no. 2, p. 4, Jul. 2010.

[4] I. F. Mir and A. A. McEwan, "A reliability enhancement mechanism for high-assurance MLC flash-based storage systems," in *Proc. IEEE 17th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, vol. 1, Aug. 2011, pp. 190–194.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2008, pp. 57–70.

[6] Intel Corporation, "Understanding the flash translation layer (FTL) specification," Intel Appl. Note AP-684, 1998.

[7] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 181–192, 2009. doi: 10.1145/1555349.1555371.

[8] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Real-time flash translation layer for NAND flash memory storage systems," in *Proc. Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2012, pp. 35–44. doi: 10.1109/RTAS.2012.27.

[9] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill, "Bit error rate in NAND flash memories," in *Proc. IEEE Int. Rel. Phys. Symp.*, Apr. 2008, pp. 9–19.

[10] L. Fan, J. Luo, Y. Mei, T. Rutt, and Z. Wang, "Lifespan analysis for redundant array of independent module based solid state drives," *IEEE Trans. Consum. Electron.*, vol. 64, no. 3, pp. 328–333, Aug. 2018.

[11] S. Wang, F. Wu, Z. Lu, J. Zhou, and C. Xie, "Ward: Wear aware raid design within ssds," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2918–2928, Nov. 2018.

[12] A. McEwan and I. Mir, "Age distribution convergence mechanisms for flash based file systems," *JCP*, vol. 7, no. 4, pp. 988–997, Apr. 2012.

[13] A. A. McEwan and M. Z. Komsul, "Reliability and performance enhancements for SSD RAID," *Microprocess. Microsyst.*, vol. 52, pp. 461–469, Jul. 2017. doi: 10.1016/j.micpro.2016.11.012.

[14] Y. Li, P. P. C. Lee, and J. C. S. Lui, "Analysis of reliability dynamics of SSD RAID," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1131–1144, Apr. 2016.

[15] S.-Y. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based SSDs," *IEEE Comput. Archit. Lett.*, vol. 9, no. 1, pp. 9–12, Jan. 2010.

[16] C. H. Wu, P. H. Wu, K. L. Chen, W. Y. Chang, and K. C. Lai, "A hotness filter of files for reliable non-volatile memory systems," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 4, pp. 375–386, Jul. 2015.

[17] I. Mir and A. McEwan, "A high performance reconfigurable flash management framework," in *Proc. Int. Conf. Inf. Sci., Electron. Elect. Eng.*, vol. 2, Apr. 2014, pp. 1216–1220.

[18] M. Z. Komsul, A. McEwan, and I. Mir, "An FPGA-based development platform for real-time solid state devices," in *Proc. Int. Conf. Inf. Sci., Electron. Elect. Eng.*, vol. 2, Apr. 2014, pp. 1198–1203.

[19] M. Z. Komsul, A. A. McEwan, and I. Mir, "A real-time hot swapping technique for SSD RAID systems," in *Proc. Int. Conf. Appl. Syst. Innov. (ICASI)*, May 2016, pp. 1–4.

[20] A. A. McEwan and M. Z. Komsul, "On-line device replacement techniques for SSD RAID," in *Proc. Euromicro Conf. Digit. Syst. Design*, Aug. 2015, pp. 438–444.

[21] S. Im and D. Shin, "Delayed partial parity scheme for reliable and high-performance flash memory SSD," in *Proc. 26th Symp. Mass Storage Syst. Technol. (MSST)*, 2010, pp. 1–6.

[22] K. M. Greenan, D. D. Long, E. L. Miller, T. Schwarz, and A. Wildani, "Building flexible, fault-tolerant flash-based storage systems," in *Proc. 5th Workshop Hot Topics Syst. Dependability (HotDep)*, Jun. 2009, pp. 1–6.

[23] S. Im and D. Shin, "Flash-aware RAID techniques for dependable and high-performance flash memory SSD," *IEEE Trans. Comput.*, vol. 60, no. 1, pp. 80–92, Jan. 2011.

[24] Y. Kim, J. Lee, S. Oral, D. Dillow, F. Wang, and G. Shipman, "Coordinating garbage collectionfor arrays of solid-state drives," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 888–901, Apr. 2014.

[25] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "Elastic parity logging for SSD RAID arrays: Design, analysis, and implementation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 10, pp. 2241–2253, Oct. 2018.

[26] L.-P. Chang and T.-W. Kuo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 3, pp. 837–863, Nov. 2004. doi: 10.1145/1027794.1027801.

[27] S. Choudhuri and T. Givargis, "Deterministic service guarantees for NAND flash using partial block cleaning," in *Proc. 6th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Oct. 2008, pp. 19–24.

[28] J. Lee, Y. Kim, G. Shipman, S. Oral, and J. Kim, "Preemptible I/O scheduling of garbage collection for solid state drives," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 2, pp. 247–260, Feb. 2013.

[29] A. A. McEwan and M. Z. Komsul, "Pre-emptive garbage collection for SSD RAID," in *Proc. Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2016, pp. 356–363.

[30] A. A. McEwan and M. Z. Komsul, "Age aware pre-emptive garbage collection for SSD RAID," *Microprocess. Microsyst.*, vol. 56, pp. 13–21, Feb. 2018. doi: 10.1016/j.micpro.2017.10.008.

[31] N. A. Rodríguez-Olivares, A. Gómez-Hernández, L. Nava-Balanzar, H. Jiménez-Hernández, and J. A. Soto-Cajiga, "FPGA-based data storage system on NAND flash memory in RAID 6 architecture for in-line pipeline inspection gauges," *IEEE Trans. Comput.*, vol. 67, no. 7, pp. 1046–1053, Jul. 2018.

[32] Y. Lu, C. Wu, and J. Li, "EGS: An effective global I/O scheduler to improve the load balancing of SSD-based RAID-5 arrays," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl., IEEE Int. Conf. Ubiquitous Comput. Commun. (ISPA/IUCC)*, Dec. 2017, pp. 298–305.

[33] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Proc. IEEE Int. Symp. Workload Characterization*, Sep. 2008, pp. 119–128.

[34] A. A. McEwan and I. Mir, "An embedded FTL for SSD RAID," in *Proc. Euromicro Conf. Digit. Syst. Design*, Aug. 2015, pp. 575–582.

[35] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The Disksim simulation environment version 4.0 reference manual (CMU-PDL-08-101)," Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-08-101, 2008, p. 26.

[36] P. Olivier, J. Boukhobza, and E. Senn, "Flashmon V2: Monitoring raw NAND flash memory I/O requests on Embedded Linux," *SIGBED Rev. Rev.*, vol. 11, no. 1, pp. 38–43, Feb. 2014. doi: 10.1145/2597457.2597462.

[37] S. Schneider, *Concurrent and Real-time Systems: The CSP Approach*, 1st ed. New York, NY, USA: Wiley, 1999.

[38] A. Sherif and H. Jifeng, "Towards a time model for circus," in *Formal Methods and Software Engineering*, C. George and H. Miao, Eds. Berlin, Germany: Springer, 2002, pp. 613–624.

[39] K. Wei, J. Woodcock, and A. Burns, "Timed circus: Timed CSP with the miracle," in *Proc. 16th IEEE Int. Conf. Eng. Complex Comput. Syst.*, Apr. 2011, pp. 55–64.

[40] M. Khattri, J. Ouaknine, and A. Roscoe, "Translating timed automata to tock-CSP," *Proc. 10th IASTED Int. Conf. Softw. Eng.*, Apr. 2011. doi: 10.2316/P.2011.720-047.

ALISTAIR A. MCEWAN received the D.Phil. degree in computer science from the University of Oxford, in 2006.

He joined the University of Leicester, U.K., in 2007, where he is currently an Associate Professor of real-time systems and software engineering with the Advanced Computational Engineering Group. He has worked on a number of projects, where he was involved in theoretical approaches to concurrent systems, systems specification and verification, modeling of safety-critical systems, and hardware/software co-design. His current research interests include application of formal software engineering techniques to the development, and safety-modeling of large systems involving software and bespoke hardware.

MUHAMMAD ZIYA KOMSUL received the B.Sc. and M.Sc. degrees in computer engineering from Trakya University, Turkey, in 2010 and 2012, respectively, and the Ph.D. degree from the School of Engineering, University of Leicester. His research interests include flash-based storage, real-time systems, FPGA-based architectures, and embedded systems.