Performance impacts with Reliable Parallel File Systems at Exascale level

Ramon Nou[1], Alberto Miranda[1], and Toni Cortes[1,2]

[1] Barcelona Supercomputing Center
[2] Technical University of Catalonia

Abstract. The introduction of Exascale storage into production systems will lead to an increase on the number of storage servers needed by parallel file systems. In this scenario, parallel file system designers should move from the current replication configurations to the more space and energy efficient erasure-coded configurations between storage servers. Unfortunately, the current trends on energy efficiency are directed to creating less powerful clients, but a larger number of them (light-weight Exascale nodes), increasing the frequency of write requests and therefore creating more parity update requests. In this paper, we investigate RAID-5 and RAID-6 parity-based reliability organizations in Exascale storage systems. We propose two software mechanisms to improve the performance of write requests. The first mechanism reduces the number of operations to update a parity block, improving the performance of writes up to 200%. The second mechanism allows applications to notify when reliability is needed by the data, delaying the parity calculation and improving the performance up to a 300%. Using our proposals, traditional replication schemes can be replaced by reliability models like RAID-5 or RAID-6 without the expected performance loss.

1 Introduction

Nowadays, it seems clear that changes will need to be made to reliability planes in parallel file systems, once storage requirements reach into the Exascale orders of magnitude. Given the strong constraints on energy efficiency imposed on Exascale clusters [1–3], the current replication techniques used by parallel file systems (PFSs) to increase reliability and availability (where several copies of each datum are kept in independent storage nodes) can represent a huge penalty, since they multiply the investment and energy costs in the storage layer.

Parity based-reliability, where mathematical checksums are computed and stored to recover failed data, is a more suitable method in this scenario since it uses less storage resources than replication. As such, there is an increasing interest to support node-wide RAID-5/6 reliability schemes in current PFSs. For instance, Lustre [4] is planning to support file-level replication, and this technique can already be found in Gluster [5]. On the other hand, Panasas (with PanFS [6]) supports object/file level RAID configurations using triple parity data [7], and GPFS [8] also supports a similar configuration with the *declustered array* technique that can visualize all the JBOD disks individually [9].

In addition, there has been some interest on shifting the HPC cluster paradigm to include more energy efficient computing nodes. Currently, some projects (e.g. Montblanc [10], or Euroserver [11]) are attempting to create the next-generation supercomputers using mobile technology, moving away from the more standard and powerful nodes (i.e., x86 or PowerPC-based) to the more energy efficient ARM-based nodes, which usually have reduced performance capabilities. Thus, to account for this reduction in performance, the number of nodes needs to be increased to provide enough computing power and, as a result, there will be an undesired congestion on all I/O levels as more requests go through the I/O layer. Unfortunately, an increase in the number of I/O requests will also affect traditional parity-based reliability techniques. Increasing the number of data writes will accentuate the *partial stripes* and *small writes* problems [12] that typically affect these strategies: a small change to a datum will force the parity checksum to be recomputed and stored, which requires additional I/O operations as well as computation. Thus, introducing these node-wide reliability strategies into Exascale storage can cause a performance impact: updating a datum in RAID-5 requires four I/O requests (reading the original datum and the old parity and writing the new datum and the new parity) and six I/O requests in RAID-6 (due it uses two parity checksums). As we will show later, even though these additional requests can be distributed between storage servers to be processed in parallel, they can represent a loss of performance of up to 85% for update operations when compared to storing raw data. In this situation, it seems clear that optimizations over RAID parity calculations are needed to remove or alleviate this performance penalty and provide Exascale storage systems with alternatives for reliability. In this paper we propose a transparent (from the user perspective) I/O layer called the Write Cache Layer that reduces the number of parity updates for arbitrary reliability configurations. Finally, we also propose a novel method (Delayed Parity) that takes advantage of the collaboration between the PFS and the clients and, using this layer, allows applications to delay parity computations. The analysis and design is evaluated with a simulator, using a write-only workload to focus on the issue that we are solving (read operations are not affected negatively).

The rest of the paper is organized as follows: In Section 2, we present the design of the Write Cache Layer. In Section 3, we describe the simulator and we evaluate our proposal. Section 4 describes related work. Finally, Section 5 states our conclusions and future work.

2 Partial Stripe Avoidance

In this section, we discuss the two strategies proposed to reduce the overhead of parity update operations. The first one (basic avoidance) actively reduces the number of read operations done on each write request, whereas the second one (delayed parity) delays the parity calculation until the application decides that it actually needs the reliability. These strategies are implemented within a new I/O layer called the Write Cache Layer (WCL) that sits on top the PFS' Object

Storage Servers (OSSs) and directly communicates with them. More specifically, the WCL keeps all the metadata on parity blocks necessary to implement the proposed strategies, communicates with the PFS servers in order to allocate enough space to write full stripes and, whenever possible, transforms the different write access patterns into a series of non-overwriting I/O operations, so that our optimization techniques can be applied. Note that this transformation of access patterns can be done with already existing techniques (e.g. log file systems [13] or versioning [14]) that fall out of the scope of this paper. Also notice that the WCL only needs to take into account write operations to implement our proposed techniques and therefore read operations can follow through normally.

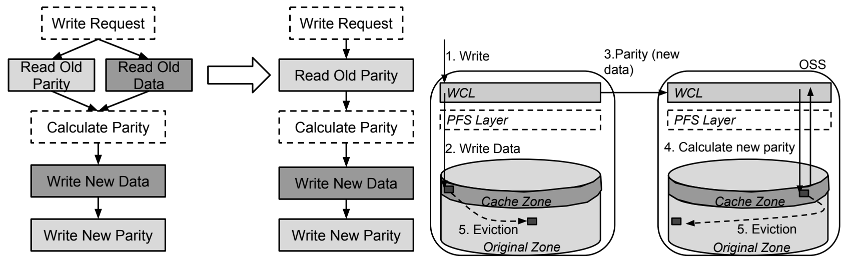
2.1 Basic Avoidance

Whenever stripes are rewritten, the PFS needs to read the original data blocks in order to be able to compute the new parity checksum. Our first proposal, the basic avoidance technique, reduces the number of data reads needed to write full stripes by avoiding these read operations of old data blocks. This is ensured because, as we have mentioned, the WCL can either detect a workload of non-overwriting writes (which should be the case of a big majority of HPC applications), or transform a mixed workload into a sequence of non-overwriting operations. To implement this mechanism, the WCL creates a *cache zone* or working zone using the available space in the storage devices connected to each OSS, and redirects all write operations to it (see Figure 1.b). This effectively allows the WCL to treat all data writes as new data writes instead of updates, which allows to compute the parities based only on the new data written (i.e., avoiding the unnecessary reads of old data blocks). Once a stripe has been completely written and its parity checksums computed, the WCL simply moves (i.e., rewrites) the stripe back to its original placement into the OSS. With this technique the reads of old data are removed from the critical path of write operations, and are replaced with full-stripe reads (probably cached in the OSS) and writes that can happen out of the critical path. The WCL layer keeps the same reliability level as in the original system (e.g, RAID-5 or RAID-6) as it is built in top of the OSSs. This is a basic difference of this technique with buffering techniques.

In order to be able to support both sequential and non-sequential access patterns, the WCL keeps an in-memory data structure (a bitmap or interval tree) to manage the used space of the cache zone. This structure is used to be able to identify which blocks need to have their parity calculated and be able to send large chunks of full-stripe writes to the disk. The metadata included in this structure is made persistent in the OSSs storage, in order to guarantee that new versions of data blocks can be safely recovered in case of unexpected shutdowns.

Figure 1.a shows the parity update workflow of the basic avoidance technique when compared to vanilla RAID-5. The technique reduces the congestion in Data OSSs by removing one (or two in RAID-6) data reads, hence reducing the number of operations when writing new data by 25% in RAID-5 and 16% in RAID-6.

Note that even though the WCL has been implemented without modifying the original PFS, better performance gains can be achieved and less control over



(a) RAID-5 vs optimized RAID-5 (shadings mean different OSSs)

(b) Workflow

Fig. 1: Basic Avoidance Technique

errors (failures, concurrency) will need to be taken if the implementation is done directly inside the PFS (e.g. the in-memory bitmap would no longer be needed). We will assume in this paper that the WCL layer is persistent in the disk, to show the worst performance. We will also consider that the available storage space is enough to avoid forced evictions, to avoid interferences in the evaluation. These forced evictions and the space required depend on the randomness, in block position terms, of the workload.

2.2 Advanced Avoidance: Delayed Parity

One of the most used fault tolerance mechanism in HPC applications is checkpointing, an operation that stores the current status of all processes creating an opportunity to restore the application in case of failure. Due to the continuous writes needed to save the state, this particular operation issues many parity update requests to maintain the reliability. However, is such reliability really needed? Consider for instance that the system fails in the middle of the checkpointing: the application could recover using a previous checkpoint and delete the partial-checkpoint file, rendering the parity computations for the partial checkpoint useless.This also applies to long computations like matrix multiplications (e.g., MADCAP, on MADBench2 [15]). A failure in the middle of the computation would require it to be restarted again from the beginning, hence the partially stored data would be discarded and the parity calculations would become an avoidable overhead. We can envision a lot of HPC applications that could make use of such functionality, since reprocessing a chunk would be less costly than the cost over all the system to store *all the data* in a reliable way compared to doing it when the process completes.

Using this idea, we propose the `START_DELAYED` and `END_DELAYED` hints to mark this kind of candidate operations, delaying the parity calculations until all the writes are completed. When the WCL receives a `START_DELAYED` hint, redirects all write operations to the caching zone and disables the parity computations for this data. Once the corresponding `END_DELAYED` hint is received, the WCL computes the parities of all the full stripes affected by the hints, and

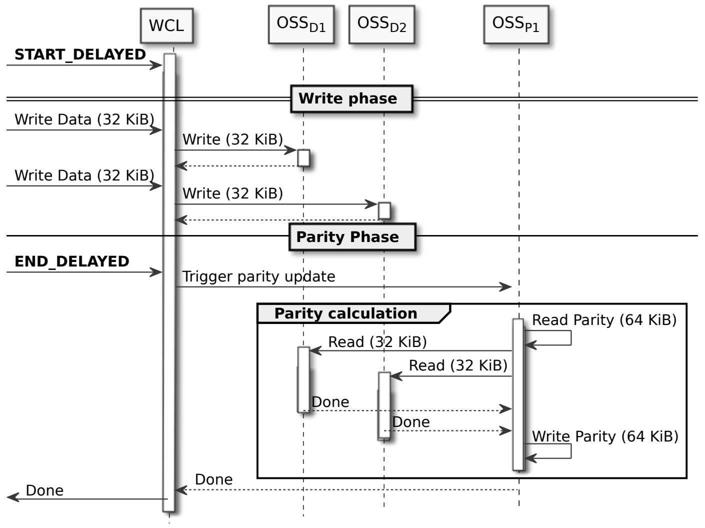


Fig. 2: Delayed Parity: Clients issue two writes and then the parity is calculated. The writes are aligned so they end at the same Parity OSS but at sequential positions. Due to this alignment, the parity calculation can be consolidated.

moves (rewrites) the data to its original location in the OSS. As a result, the PFS can avoid using the Parity OSSs during the creation of the data and, when parities are calculated, it only needs to send them the consolidated writes.

Figure 2 shows a simplified sequence diagram for the delayed parity technique. In that example, 2 clients issue a 32 KiB write and then their parity is calculated. The writes are directed to different data OSS, but the same Parity OSS. Parity blocks are sequential, and hence parity updates can be consolidated.

The WCL supports synchronous parity calculations, but can also advance the calculation in the background to reduce time. However, advancing the calculation may generate extra work if the calculation is not needed (i.e., application cancellation or error in the client). In short, the delayed parity technique can be represented as a collective operation between all clients, but without increasing intermediate memory requirements since partial data will be written to disk. Failure in acquiring the `END` signal can be detected with timeouts.

3 Evaluation

Due to the large number of clients, servers and requests involved in Exascale computing clusters, we decided to evaluate the feasability of the proposed techniques using simulations. The simulation environment is created using the OMNet++ framework [16], a C++-based event simulation package targeted at simulating computer networks and other distributed systems. In order to create a network simulation as real as possible, the devised simulation implements Lustre message sizes and payloads, including the actual sizes of Lustre RPC message headers.

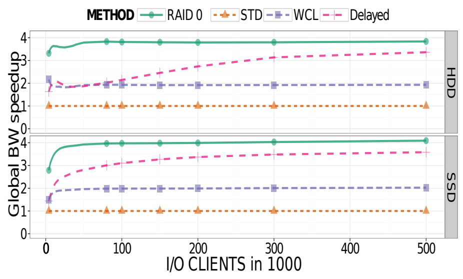
For the simulation of the storage devices, we decided to capture the individual I/O requests received by each of the devices configured in each experiment and replay them on a real device using the FIO benchmark, taking special consideration to include the interference between different requests. This allowed us to create a complete statistical model for each device, that can be used to approximate the behaviour of the real device under the conditions imposed by the network simulation. The decision to create this statistical model came after attempting to use DiskSim [17] with the SSD addition from Microsoft Research and finding severe limitations when using more than 64 combined devices at once. The devices modelled in our simulation are Western Digital Black Hard Disks (750GB capacity, 7200 RPM) and Intel SSD 320 Series (160 GB capacity).

Regarding client behaviour, the simulated application uses a set of clients issuing writes to the I/O layer. During a single simulation run, the datum size of write operations is fixed for all the clients and the writes are distributed along a different file per application to avoid overwrites. Each client writes enough data to produce a statistically representative number of parity calculations. The behaviours of the applications simulated mimic that of the FLASH application [18], a computational tool for simulating and studying thermonuclear reactions, that periodically outputs large checkpoint files and smaller plot files. For the Delayed Parity evaluation, one process of each application acts as master issuing the new hints and all the clients wait until the parity is calculated.

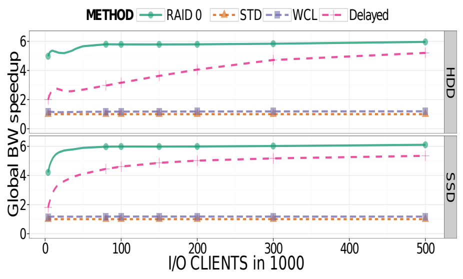
Workloads with mixed block sizes were only tested with a low number of devices, as the cost of generating the statistical device model was too high (it is necessary to attempt all the possible combinations of request sizes, which grows exponentially). This is an important drawback of modelling the storage devices without a simulator. Nevertheless, we did not find significant differences between these simulated workloads in systems with the large number of clients targeted in the paper. The same also applies to different stripe sizes.

We decided to use this workload in order to concentrate on the effect of typical HPC writes over the proposed reliability techniques. Nevertheless, we also checked other workloads, which showed similar results. In particular, our preliminary results using mixed workloads (with reads and writes), showed improvements in the performance on read operations since the proposed techniques favoured a reduction of the overhead on the storage devices. Due to space limitations, we will not discuss these results further.

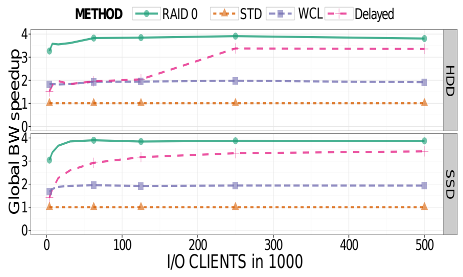
To assess the effectiveness of the novel strategies, we repeat each measurement with different seeds, that control how requests are mapped into each storage device. Each simulation run stops when we have a minimum of 1000 seconds of simulated time, other variables (BW, latency, etc.) are tracked to assess that the results are representative. In the following experimental results, RAID-0 represents the performance obtained from the OSSs when there are no parity calculations. STD-RAID represents a standard striped RAID with parity and without optimizations. Finally, our two proposals are WCL and Delayed Par-



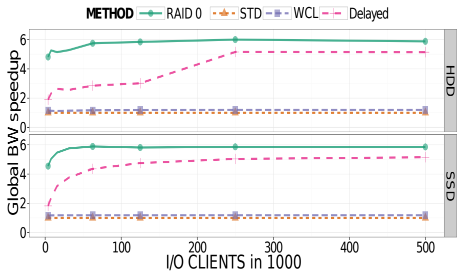
(a) Weak-Scaling



(b) Weak-Scaling



(c) Strong-Scaling



(d) Strong-Scaling

Fig. 3: RAID-5 (left) and RAID-6 (right) write bandwidth w.r.t. STD-RAID.

ity. If not specified, the results are scaled relative to STD-RAID for each x-axis point. We use 512 OSSs in all the experiments with a stripe unit of 1MiB and a width of 512 OSSs.

We measure the results in two different ways: using Weak-Scaling and using Strong-Scaling. Weak-Scaling means that the problem size increases with the number of clients and thus, regardless of the number of clients, each one will process the same amount of data (128 KiB per data write). Conversely, in Strong-Scaling the number of clients is increased, but the problem size is kept. Consequently, this increases the network and storage congestion due to I/Os when the number of clients increases, due to the higher number and smaller size of requests. In this scenario, the data writes grow from 8 KiB to 1MiB according to the number of clients in the simulation. For all the scenarios, the data is partitioned to avoid overwrites according to their block size. We analyse the outcome of the new techniques using RAID-5 in Subsection 3.1 and RAID-6 in Subsection 3.2.

3.1 RAID-5 Results

This section describes the measured results of our simulations when reliability is implemented using RAID-5 (i.e., one parity checksum per stripe).

Weak-Scaling results. Each client writes 128 KiB per write distributed along a file avoiding overwrites with other clients. As we can see on the simulation results in Figure 3.a, the performance of writes is four times slower on STD-RAID w.r.t. RAID-0 both in HDDs and SSDs. Since we go from one operation to four operations per write to two different OSSs, the performance degradation is expected. Moreover, the WCL scenario results in a speedup of 2x w.r.t. the STD-RAID, achieved simply by eliminating the read operation of old data blocks.

Regarding the Delayed Parity proposal, the strategy reduces the number of writes on the corresponding Parity OSS for each row of the file ($m-1$ vs 1, where m are the OSSs involved) w.r.t. the WCL strategy, but we increase the reads in the data OSSs (0 vs $m-1$) as we need to transfer the data to the parity node to calculate the parity (see Figure 2). If we take a look at the number of accesses per Data OSS and Parity OSS, we observe that only one access is needed to the Data OSS with WCL (to write) and two with the Delayed Parity approach (to write and to read in order to calculate the parity). On the other hand, using the WCL approach requires a parity for each write (a read and a write to the Parity OSS), whereas with the Delayed Parity approach the parity only needs to be generated when all operations have completed (i.e., a maximum of two accesses to the OSSs).

As we can see in Figure 3.a, the benefit of the Delayed Parity strategy depends on the cost of the operations in the device. In that Figure, with less than 80,000 clients, the required time to complete the described operations on the HDD devices surpasses the reduction on the number of operations. As a result, the performance with Delayed Parity with a low number of clients is comparable to the WCL strategy, but without the reliability of RAID-5 (as the parity is calculated at the end). Despite this result, the general behaviour is for performance to grow up close to RAID-0, as we remove a big number of parity updates producing a higher throughput.

Apart from the results presented with a 128 KiB write size, if we fix the number of clients to 250K and we check different block sizes, we can observe how the relative performance is stable using HDDs. However, using 4KiB blocks with SSDs STD-RAID achieves a better performance, which means that the benefit of the other schemes is lower. The reason is that SSDs offer a better performance of parallel operations, and thus a reduction of them on 4 KiB block sizes does not produce the same improvement w.r.t. the STD-RAID.

Strong-Scaling Results. The experiments done with Strong-Scaling (see Figure 3.c.) are similar to the Weak-Scaling ones. Specifically, we obtain the same results for RAID-0, STD-RAID and WCL, and thus the difference on block size (i.e., from 128 KiB for Weak-Scaling to 8 KiB for Strong-Scaling in the 500,000 client scenario) is not important at this scale.

Using the Delayed Parity technique, we observe performance improvements due to the fact that the number of parity updates is reduced greatly as the number of clients increases. For instance, with 250K clients, we do not issue a parity update until all the clients of an application have stored their 16KiB to the devices, which means that we go from 250K parity updates (clients·appl. iterations) to 1.7K parity updates ($\frac{\text{clients}}{\text{clients per appl.}} \cdot$ appl. iterations). Actually, the performance obtained differs when using HDD or SSD technology (as in Weak-Scaling). On HDDs, the Delayed Parity achieves less performance, thus it may be preferable to avoid using this strategy for a small number of clients.

3.2 RAID-6 results

The complexity and variability of RAID-6 deployments make it more difficult to simulate than RAID-5. Using RAID-6 with two or more parity devices, where location and modification rules depend on the erasure codes used, adds too many variables into the evaluation. The position of the parity devices can modify the number of operations of each OSS and the network communication patterns, but in any situation, our proposal will always optimize them by removing the need to read the old data (because it is a new write). For this reason, in this experiments we have selected two horizontal parity devices per stripe for RAID-6.

Weak-Scaling results. As we can observe in Figure 3.b, we found that the STD-RAID performance of RAID-6 configurations is lower than for RAID-5 configurations and the improvement on performance using WCL is a bit lower (1.18x speedup). This happens because the strategy can only successfully avoid 16% of the operations instead of the 25% for RAID-5.

The Delayed Parity option offers a bigger performance boost on RAID-6 since we now have two parity devices, and we move from two parity updates per write to two parity updates per application iteration (`START-END` hint).

Strong-Scaling results. For Strong-Scaling results with RAID-6 we obtain similar results to RAID-5 (see Figure 3.d). Like with RAID-5 results, the block size is not important on the simulated scenarios for all the experiments except Delayed Parity. In that particular experiment, we can see the same performance loss found on RAID-5 with HDD devices. However, the performance improvement compared to the WCL proposal is bigger even with a lower number of clients since parity updates are more expensive in RAID-6 than in RAID-5. Thus, removing them (more precisely, grouping and delaying them) produces bigger improvements. In general, since the performance loss of STD-RAID w.r.t. RAID-0 is much higher for RAID-6, the potential gain of the Delayed Parity strategy is higher.

4 Related work

RAID-5 and RAID-6 are redundant systems that provide a way to recover from a data loss using the remaining disks, so for RAID-5 we can recover using $n-1$ disks and with RAID-6 we can recover using only $n-2$ disks. RAID-6 can use a huge range of erasure codes offering different performance and recoverability values. One such example is *Reed-Solomon*, which maps to a polynomial equation so the missing data recovers using interpolation, therefore we can use extra devices to extend the recovery capabilities (for example 12 data disks using 4 redundant disks, will be able to recover any failed disk from 12 correct disks of the 15 available that are still working).

Node-local Redundancy. Inside RAID-6 research at controller layer, we can find optimizations of the erasure codes as in *P-Code* [19] and *HDP-Code* [20], and optimizations of the erasure codes using specialized hardware (FPGAs and GPUs [21]) as in Gilroy [22] and Curry [23]. Moreover, we can find improvements with SSD RAIDs [24] taking into account the wear-levelling of the parity stripe. Directly related to the partial write on stripes problem, we found H-Code [25] with a performance improvement of 15.54% and 22.17% compared to *RDP* and *EVEN-ODD* erasure codes. Finally, there are also patents that solve the problem in the hardware level as Lyons [26] and Baylor [27], reducing the reads and writes done at the controller level. Historically we also have parity logging to solve small writes problem as Stodolsky [12] proposes, we study the problem in distributed systems, RAID-6, and newer devices as SSDs. Our first proposal maintains the reliability of the original system using a small fraction of space of the devices, depending on the randomness of the workload and the cost of evictions, and reduces the number of operations issued to the devices.

Those proposals may not be fully usable or become inefficient at the storage server layer, as it involves network communication and bigger latencies. Despite of this, some of them may improve the performance due to the different parity layouts or calculations. Our work is transparent to the reliability configuration used and will improve their performance.

Distributed Redundancy. Inside distributed redundancy, with can find TickerTAIP [28] building a RAID system using the network as transport method. It can be seen as a preliminary approach to support RAID parity schemes over PFS, finally RAID-x [29] presents how to optimize it using a mirroring mechanism reducing the number of operations in small writes. On the PFS plane, GlusterFS supports mirroring schemes and some requests have been done to support RAID-5/6 schemes. Hadoop (HDFS [30]) supports file replication. Finally, PanFS supports file level RAID configurations using triple parity data [7].

Delayed Parity Calculation. About our delayed parity proposal, a similar approach is found in NetApp [31] where writes are buffered to issue an improved write operation. Also, at AFRAID [32] they move the parity calculation to idle periods to obtain a performance boost. The main difference of our proposal with the previous mentioned works is that the lower reliability mode is selected by the user (via hints) when he decides that the data is not useful until it is completed (i.e., check-pointing or partial results that will need to be recalculated). All writes are persisted to the disk, so it may recover from failures, at the same rate than the used PFS.

5 Conclusions and Future Work

Under Exascale constraints, reliability will be needed on the PFS layer if we want to keep the storage costs and the energy used under control. Especially, when we use a high number of clients the number of parity updates will increase.

We propose a transparent cache layer that is able to reduce the number of operations needed to update the parity on such environments. To do that, we ensure that the writes are not overwriting so we can drop the read of old data from the parity update workflow. This proposal improves the write performance of the standard workflow by a 1.18x to a 2x depending on the RAID level (6 or 5, respectively). Moreover, we show that applications gain substantial performance controlling the parity calculation as in the Delayed Parity Proposal. Using reliability oriented application hints, we can improve the write performance up to levels near a RAID-0. This behaviour is useful when partial data does not need to be reliable until all the data writing is finished, e.g. big partial matrices.

In this paper, we used simulation to predict the impact of these strategies. The implementation in existing file systems is non-trivial and out of scope of this paper; nevertheless this theoretical consideration steers the direction of a beneficial implementation in the future.

Acknowledgements

The research leading to these results has received funding from the European Community [FP7 / 2007-2013] Montblanc project under grant agreement 288777 and 610402, by the Spanish Ministry of Economy and Competitiveness under the TIN2012-34557 grant and by the Catalan Government under the 2014-SGR-1051 grant. Thanks to Julian Kunkel and André Brinkmann for their comments and corrections of this paper.

References

1. Villa, O., Johnson, D.R., O'Connor, M., Bolotin, E., Nellans, D., Luitjens, J., Sakharnykh, N., Wang, P., Micikevicius, P., Scudiero, A., et al.: Scaling the power wall: a path to exascale. In: SC'14
2. Rajovic, N., Vilanova, L., Villavieja, C., Puzovic, N., Ramirez, A.: The low power architecture approach towards exascale computing. Journal of Computational Science **4**(6) (2013) 439 – 443
3. Bergman, K., et al.: Exascale computing study: Technology challenges in achieving exascale systems. Technical report, Technical report, DARPA (2008)
4. Braam, P.J., Zahir, R.: Lustre: A scalable, high performance file system. Cluster File Systems, Inc (2002)
5. Gluster: Glusterfs web page. `http://www.gluster.org/` (2014)
6. Nagle, D., Serenyi, D., Matthews, A.: The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In: SC'04
7. Panasas: PanFS RAID. `https://www.panasas.com/products/panfs/PanFS_RAID`
8. Schmuck, F.B., Haskin, R.L.: GPFS: A shared-disk file system for large computing clusters. In: FAST'02. Volume 2. 19
9. Deenadhayalan, V.: GPFS Native RAID slides, Invited talk at LISA'11 (2011)

10. Rajovic, N., Carpenter, P.M., Gelado, I., Puzovic, N., Ramirez, A., Valero, M.: Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? SC '13 (2013) 40:1–40:12
11. Durand, Y., Carpenter, P., Adami, S., Bilas, A., Dutoit, D., Farcy, A., et al.: Euroserver: Energy efficient node for european micro-servers. In: DSD'2014
12. Stodolsky, D., Gibson, G., Holland, M.: Parity logging overcoming the small write problem in redundant disk arrays. In: ACM SIGARCH Computer Architecture News. Volume 21. (1993) 64–75
13. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM TOCS **10**(1) (1992) 26–52
14. Muniswamy-Reddy, K.K., Wright, C.P., Himmer, A., Zadok, E.: A versatile and user-oriented versioning file system. In: FAST'04
15. Carter, J., Borrill, J., Oliker, L.: Performance characteristics of a cosmology package on leading hpc architectures. In: HiPC 2004. Springer 176–188
16. Varga, A., et al.: The OMNeT++ discrete event simulation system. In: ESM 2001. Volume 9. (2001) 185
17. Bucy, J.S., Schindler, J., Schlosser, S.W., Ganger, G.R.: The disksim simulation environment version 4.0 reference manual. Parallel Data Laboratory (2008)
18. Latham, R., Daley, C., Liao, W.k., Gao, K., Ross, R., Dubey, A., Choudhary, A.: A case study for scientific I/O: Improving the FLASH astrophysics code. Computational Science & Discovery **5**(1) (2012) 015001
19. Jin, C., Jiang, H., Feng, D., Tian, L.: P-Code: A new RAID-6 code with optimal properties. In: 23rd international conference on Supercomputing. (2009) 360–369
20. Wu, C., He, X., Wu, G., Wan, S., Liu, X., Cao, Q., Xie, C.: HDP code: A Horizontal-Diagonal parity code to optimize I/O load balancing in RAID-6. In: Dependable Systems & Networks (DSN). (2011) 209–220
21. Brinkmann, A., Eschweiler, D.: A microdriver architecture for error correcting codes inside the linux kernel. In: SC'09. (2009)
22. Gilroy, M., Irvine, J.: RAID 6 hardware acceleration. In: FPL'06. (2006) 1–6
23. Curry, M.L., Skjellum, A., Ward, H.L., Brightwell, R.: Accelerating reed-solomon coding in RAID systems with GPUS. In: IPDPS 2008. (2008) 1–6
24. Park, K., Lee, D.H., Woo, Y., Lee, G., Lee, J.H., Kim, D.H.: Reliability and performance enhancement technique for SSD array storage system using RAID mechanism. In: ISCIT 2009. (2009) 140–145
25. Wu, C., Wan, S., He, X., Cao, Q., Xie, C.: H-Code: A hybrid MDS array code to optimize partial stripe writes in RAID-6. In: IPDPS. (2011) 782–793
26. Lyons, G.: Method and apparatus for improving sequential writes to RAID-6 devices (2000) US Patent 6,101,615.
27. Baylor, S., Corbett, P., Park, C.: Efficient method for providing fault tolerance against double device failures in multiple device systems (1999) US Patent 5,862,158.
28. Cao, P., Lin, S.B., Venkataraman, S., Wilkes, J.: The tickertaip parallel raid architecture. ACM TOCS **12**(3) (1994) 236–269
29. Hwang, K., Jin, H., Ho, R.: Raid-x: A new distributed disk array for i/o-centric cluster computing. In: High-Performance Distributed Computing, 2000. 279–286
30. Borthakur, D.: Hadoop - HDFS Design (2014) Apache.
31. Kleiman, S., Sundaram, R., Doucette, D., Strange, S., Viswanathan, S.: Method for writing contiguous arrays of stripes in a RAID storage system using mapped block writes (2007) US Patent 7,200,715.
32. Savage, S., Wilkes, J.: AFRAID: a frequently redundant array of independent disks. In: USENIX ATC'96