Comparing archival policies for BLUE WATERS

Franck Cappello[1], Mathias Jacquelin[2], Loris Marchal[2], Yves Robert[2] and Marc Snir[1]

1:INRIA-UIUC joint laboratory for Petascale Computing
2: LIP laboratory, CNRS, INRIA, ENS-Lyon & University of Lyon , France
fci@lri.fr, {Mathias.Jacquelin|Loris.Marchal|Yves.Robert}@ens-lyon.fr, snir@illinois.edu

***Abstract*—This paper introduces two new tape archival policies that can improve tape archive performance in certain regimes, compared to the classical RAIT (*Redundant Array of Independent Tapes*) policy. The first policy, PARALLEL, still requires as many parallel tape drives as RAIT but pre-computes large data stripes that are written contiguously on tapes to increase write/read performance. The second policy, VERTICAL, writes contiguous data into a single tape, while updating error correcting information on the fly and delaying its archival until enough data has been archived. This second approach reduces the number of tape drives used for every user request to one. The performance of the three RAIT, PARALLEL and VERTICAL policies is assessed through extensive simulations, using a hardware configuration and a distribution of I/O requests similar to these expected on the BLUE WATERS system. These simulations show that VERTICAL is the most suitable policy for small files, whereas PARALLEL must be used for files larger than 1 GB. We also demonstrate that RAIT never outperforms both proposed policies, and that a heterogeneous policies mixing VERTICAL and PARALLEL performs 10 times better than any other policy.**

I. INTRODUCTION

The new generation of petascale supercomputers will need exabyte-scale archival storage. For example, the BLUE WATERS petascale system that is being installed at the University of Illinois, Urbana-Champaign [1] will have a peak performance of over 10 petaflop/s, over a petabyte of DRAM, and over 18 petabytes of disk storage; yet most of the storage – up to half an exabyte – will be on tapes. This is consistent with a recent report [2], based on eight year experience at several major High Performance Computing (HPC) centers, that shows a need to archive about 35 TB of new data each year for every TB of memory, not counting archived data that are deleted (20-50%). In a such a system, there is a crucial need for efficient archival policies for writing to and reading from the tape system.

In addition, archive systems need to be reliable. Experience shows that a significant fraction of jobs in HPC centers fail because of some errors in the archive system: inability to load the tape, metadata errors on the tape, tensioning errors, breaking tape, etc. It is estimated in large centers that 1 out of 100 tape handling events leads to a job failure [3]. Moreover, tapes may face unrecoverable data errors leading to permanent data loss. The current solution to prevent data loss and avoid the propagation of a failed tape handling event is RAIT, which stands for *Redundant Array of Independent Tapes* [4] – in analogy to RAID.

On BLUE WATERS, the disk space will be managed by GPFS [5], while the archival tape system will be managed by HPSS [6], [7]. The GPFS-HPSS Interface (GHI) integrates the tape archive into the GPFS namespace, so that the disk storage essentially is a cache for the tape storage, and data migrates transparently between disk and tape.

HPSS supports RAIT Level 0 (mirroring, for reliability) and RAIT Level 1 (stripping, for increased transfer speed). Mirroring doubles the amount of tape storage needed – an expensive proposition. There is ongoing work on HPSS support for schemes similar to RAID 6 [3]. Such a RAIT architecture requires $X + Y$ tape drives for archiving a single file. The file is split into blocks; for each X consecutive blocks one computes Y *Erasure Code* (EC) blocks. The $X + Y$ blocks are written on $X + Y$ distinct tapes, with EC blocks rotated across tapes. Such a design can recover from the failure of any Y tapes, and speed up transfer rate, by a factor of X. (However, start-up time increases, as $X + Y$ tapes need to be loaded.) Values being considered include $X+Y = 4+1$, $4+2$ or $8 + 2$.

The main drawbacks of such a RAIT architecture are the following:

- Because X consecutive blocks of the (currently written) file are stored in parallel, files are scattered across many tapes.
- Each archival request monopolizes $X + Y$ tape drives, which considerably reduces the number of user requests that can be simultaneously processed by the system.

The first problem (file fragmentation) is expected to have a dramatic impact on performance. Contiguous access is faster when reading from tapes, just as it is for disks, but the speedup ratio is much larger. Suppose that, in order to use a tape efficiently, one needs to access a contiguous block of size at least S; then, with RAIT, one needs to access at least $X \times S$ contiguous data to use the tapes efficiently. Many files might be shorter than this threshold. It is possible to solve this issue for writes, by concatenating multiple short files into one larger "container file". However, subsequent reads will have low performance unless all the files concatenated in one container are accessed together – something that is not always true, and cannot be guaranteed, especially in a system such as Blue Waters where archival is initiated by the system, not by a user.

The second problem (several tape drives per request) will drastically limit the access concurrency of the system, by increasing the response time when many users aim at archiving

their data. If, say, 500 tape drives are available, and if the archival policy requires 10 tape drives per request, then at most 50 requests can be served simultaneously. This may well prove a severe limitation for some usage scenarios of the target supercomputer platform. Furthermore, the average start-up time for file transfers can increase significantly, since the number of robotic arms to move tapes is often lower than the number of tape drives, and the transfer can start only after the last tape was mounted.

Note that, unlike for RAID, it is not always necessary to read the redundancy blocks when a file is accessed: Since tape blocks are long, one can compute and store longitudinal codes to ensure that data read is valid. Also, one can leverage the fact that disk storage (unlike main memory) is persistent to delay the storage of error correcting information to tape, thus enabling more asynchrony. These differences allow for new policies, different from classical RAID.

To overcome the shortcomings of RAIT, we have designed two new archival policies. The first of them, PARALLEL, still uses the same number $X + Y$ of tape drives, and hence suffers from the same problem that it reduces the servicing capacity of the system and increases start-up time. But it does reduce the fragmentation of files, by pre-partitioning such files into X stripes that will be written in contiguous mode on the tapes. Subsequent reads will be able to access larger segments from each tape.

The second policy, VERTICAL, is more drastic and solves both problems, at the price of lower transfer rates. The idea is to write data contiguously and sequentially on X tapes, filling up the tapes one by one, and to delay the archival of the redundancy data on Y tapes after X tapes have been actually written. This requires to update the contents of the Y redundancy tapes on-the-fly. This scheme allows for serving as many requests as the number of available tape drives. However, each request is processed without any parallelism in writing, hence transfer rate decreases. (The problem can be avoided, for very long files, by simultaneously writing or reading tape-sized segments; it is not an issue for very short files, where tape load and seek time dominates access time; it affects files in a range in between these extremes.)

The main goal of this paper is to evaluate the three RAIT, PARALLEL and VERTICAL policies within an event-driven simulator, and to compare their performances through extensive simulations. The simulation setting corresponds to realistic execution scenarios (in terms of both hardware platform parameters and I/O request rates) for the future exploitation of BLUE WATERS. After studying their relative performance on different file sizes, we propose a last strategy, which mixes the best two candidates (PARALLEL and VERTICAL) to outperform them.

The paper is organized as follows. We first briefly review related work in Section II, and we outline the framework in Section III. Then we detail the three archival policies in Section IV. The main scheduler and load balancer are described in Section V. The simulation setting is provided in Section VI, as well as the results of the comprehensive simulations. Finally, we state some concluding remarks and hints for future work in Section VII.

II. RELATED WORK

We classify related work into two main categories, those dealing with resilient storage policies, and those discussing tape request scheduling strategies.

a) Resilient storage policies: The first fault-tolerant policy proposed for tapes was inspired from disks. It adapts the classical disk RAID policy for tapes, and thus was called RAIT [4]. An important difference between RAID and RAIT is that the erasure code is computed from disk blocks on RAID 5 and RAID 6, while RAIT compute the erasure code from file stripes. Notes that this approach of encoding the data has also been proposed recently to overcome the issues related to RAID 5 and RAID 6.

Jonhson and Prabhakar proposed to decouple the stripes used to write files to the tapes from the ones used to compute parity, called *regions* [8]. Their basic idea is to group a number of regions from different tapes into a parity group and to compute and store the parity of these regions on another tape. The proposed framework allows for a wide variety of policies, such as the ones developed in this paper.

b) Scheduling tape requests: Together with designing tape storage policies, we also need to schedule I/O requests. Some specific problems to I/O on tapes have been considered in the literature. In [9], Hillyer et al. consider the problem of scheduling retrieval requests to data stored on tapes. Using a precise model for the performance of the tape drives, they proved the problem of minimizing the completion time for a set of request NP-hard, and proposed a complex heuristic to solve it.

In [10], Prabhakar et al. consider the problem of scheduling a set of storage requests using a simple model of tape storage, with the objective of minimizing the average waiting time. An optimal scheduling policy is provided for the one tape drive case, and the problem is proven NP-complete for multiple drives.

To the best of our knowledge, this work is the first aimed at designing innovative tape archival policies for petascale computers like BLUE WATERS, and assessing their performance.

III. FRAMEWORK

In this section, we first describe the platform model, and then we state the optimization problem under consideration.

A. Platform model

We derive a model that is representative of a system such as Blue Waters, but does not match exactly its (currently confidential) configuration. Our goal is to simulate the maximum capacity of the BLUE WATERS archival storage (0.5 Exabytes), that is much higher than its initial capacity. Here is a list of key parameters describing the platform:

Tapes The archival system counts 5,000,000 serpentine tapes. Each tape stores up to 1 TB of uncompressed data.

Tape Drives There are 500 tape drives to perform read/write operations on these tapes.

Tape Libraries Tapes are gathered into 3 tape libraries, with passthrough to transfer tapes between different libraries

Mover Nodes 50 mover nodes are dedicated to process I/O requests and compute redundancy blocks. Each mover node has 24 cores and 96 GB of RAM; a mover mode is connected to 10 tape drives. We assume that a mover node has access to 10 TB of local disk storage.

Additional computing resources are used by HPSS, e.g., to schedule transfers, and by GPFS to run file system code.

B. Problem statement

The focus of this study is to handle I/O requests in an efficient way. An I/O request can be sent in the system in response to an explicit user command (to archive or delete data, or move it to disk or off-site); by the automatic disk management system (to migrate from disk data not touched recently); or by the job scheduler (to load to disk files needed by scheduled batch jobs).

An I/O request is characterized by the file that it is accessing, and therefore by the size of this file. The request is also associated with a resiliency scheme $X+Y$, where X denotes the number of data blocks corresponding to Y Erasure Code (EC) blocks. The Y EC blocks are computed by using any EC algorithm over X blocks of data. Finally, an I/O request is defined by the I/O policy which it is using, i.e., the way data and EC blocks are organized onto tapes. As already stated, three I/O policies will be considered in this study: RAIT, VERTICAL and PARALLEL.

The most natural objective function is the *average response time* for a request, which measures the time between the arrival of a request in the system and the completion of its processing. However, this objective is known to unduly favor large request over smaller ones, and the objective of choice is rather the *average weighted response time*, where the response time for a request is divided by its size. The weighted response time is close to the *stretch*, which is a widely used fairness objective [11]. The stretch is the slowdown experienced by the request, i.e., its response time in the actual system divided by its response time if it were alone in the system (this later quantity being roughly proportional to the file size if we neglect all latencies). Another important, platform-oriented objective, is the aggregate throughput, or aggregated bandwidth of I/O operations, achieved by the system.

IV. Tape archival policies

Writing data to tapes is a challenging task, and particular care is required to design and implement efficient archival policies. In this section, we first review the well-known RAIT policy. Then we introduce two novel I/O policies, VERTICAL and PARALLEL.

A. RAIT

The first policy is called RAIT, for "Redundant Array of Independent Tapes", and is the counterpart of RAID for tapes [4]. In order to overcome the performance and reliability limitations of each individual tape drives, RAIT writes data in parallel while keeping the usage of all tapes balanced. In addition to the resiliency scheme $X+Y$, which requires $X+Y$ tape drives, RAIT is characterized by a block size B which defines the transfer unit. In order to ensure resiliency, RAIT computes Y EC blocks from X data blocks. These EC blocks are computed in memory before data blocks are actually written to tapes. This implies a memory footprint of $(X+Y)\times B$ for each RAIT request running concurrently.

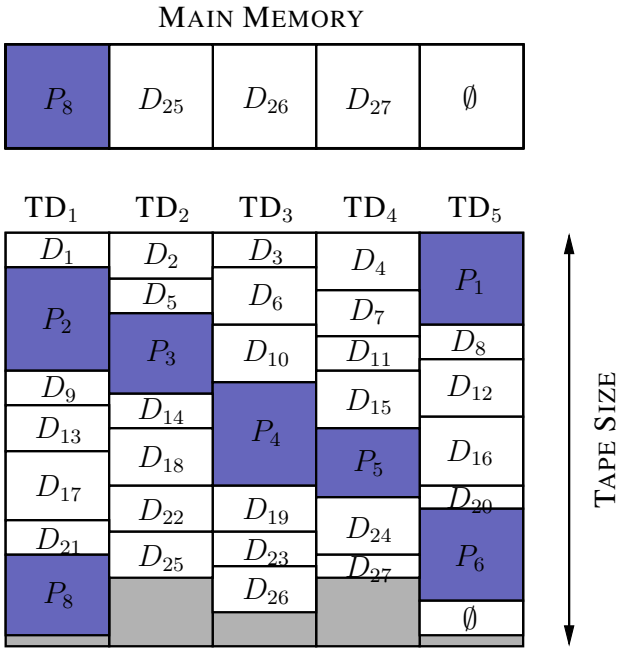


Figure 1. Writing data with RAIT policy for $X=4$ and $Y=1$.

The behavior of RAIT is depicted on Figure 1: if all tapes are empty, the first X blocks are written on tape drives TD_1 to TD_X, while the Y EC blocks are written on TD_{X+1} to TD_{X+Y}. The following sets of X data and Y EC blocks are then periodically shifted (see Figure 1). Note that when several consecutive requests of the same policy and resiliency scheme are served using the same tapes, they are processed with consecutive unit shifts as if we had a single large request.

Note that tape drives generally offer a hardware compression mechanism. Hence, although every block has a size of B, the space occupied on tape may differ from block to block. In particular, EC blocks are expected to be much less compressible than data blocks. RAIT balances tape occupation through its periodical shifting mechanism.

The use of $X+Y$ tapes for each file transfer reduces the number of concurrent transfers possible, and increases start-up time. Moreover, as data and EC blocks are periodically shifted

across all tape drives, it is not possible to bypass EC blocks on reads.

B. PARALLEL

One of the main drawbacks of RAIT is that data on tapes is fragmented. In order to solve this issue, we introduce a novel policy called PARALLEL. This policy keeps the parallel I/O operations offered by RAIT but rather writes data as much contiguously as possible.

Just like RAIT, PARALLEL is characterized by a block size B. For a given resiliency scheme $X+Y$, it also requires $X+Y$ tape drives for writing data along with Y tape drives for writing EC blocks. These EC blocks are computed in memory from X data blocks before being actually written to tapes. The memory footprint is therefore $(X+Y)\times B$ for each PARALLEL request running concurrently. However, unlike RAIT, (i) all EC blocks are written on Y separate tapes; and (ii) the system writes on each of the X data tapes the longest possible sequence of consecutive data blocks. Thus, if the file has S blocks, then each of the X data tapes will store either $\lceil S/X \rceil$ or $\lfloor S/X \rfloor$ consecutive data blocks. In the simpler case where $S = W \times X$ then, at step i, the system transfers to X data tapes the X file blocks $D_i, D_{i+W}, \ldots, D_{i+(X-1)W}$, and transfers to the Y EC tapes the Y EC blocks computed from the X data blocks. The scheme is depicted on Figure 2: first, blocks D_1, D_7, D_{13} and D_{19} are held in memory and EC block P_1 is computed. Everything is then written on distinct tape drives. Then, next sets of blocks are processed the same way until a tape dedicated to data gets filled. Whenever this happens, every tapes are replaced by new empty tapes. However, it may happen that some (or all) of the tapes dedicated to store EC blocks get filled before data tapes (as depicted on Figure 2(a)), EC blocks being generally less compressible. In such a case, these tapes are ejected and replaced by new ones. Overflowing EC blocks are then written onto those new tapes. This is what happens on Figure 2(b).

Altogether, PARALLEL maintains the same level of concurrency in the transfer of files to/from tape, but stores contiguously as much data as possible. Because of the hardware compression mechanism embedded in tape drives, and given that EC blocks are generally less compressible than data blocks, tape occupation is slightly unbalanced. This can lead in some cases to the use of extra tapes to hold the Y EC blocks, which could have an effect on performance. Like RAIT, PARALLEL also has a significant impact on the level of parallelism of the system, although lower than RAIT, since $X+Y$ tape drives are required for write operations, but only X tape drives are needed for read operations.

We point out that adapting RAID-3/RAID-4 to tapes would lead to a quite inefficient RAIT-4 policy. Indeed, RAIT-4 is similar to the RAIT policy presented above, but with EC blocks being not shifted among tapes, and instead being written on Y dedicated tapes. There is a major difference between RAIT-4 and PARALLEL: in RAIT-4, the blocks written on a given (data) tape are $D_i, D_{i+X}, D_{i+2X}, \ldots$, where are in PARALLEL, we have consecutive blocks $D_i, D_{i+1}, D_{i+2}, \ldots$. Thus, RAIT-4 combines all drawbacks: (i) it uses non-consecutive blocks, which is expected to slow down the I/O operations on tapes; and (ii) it suffers from imbalance in tape occupation, since it uses dedicated tapes for EC blocks. This is why we have discarded RAIT-4 from our policy evaluation.

C. VERTICAL

With both previous policies, parallelism and resiliency are tightly coupled, which tends to decrease the overall level of service provided by the system. In order to break this coupling, we propose the VERTICAL policy, which also keeps data entirely contiguous on tapes.

For a given $X+Y$ resiliency scheme, VERTICAL writes X data tapes before writing Y EC tapes (or more, according to the compression rate). All writes are performed serially, on one tape drive. In order to do so, VERTICAL requires enough local storage space to store Y uncompressed tapes. Each time a data block is written, the corresponding Y EC blocks are updated; the computation of these EC blocks is completed after X data blocks have been written. The scheme is depicted on Figure 3. Several areas are allocated on disk dedicated to hold the Y ECs. As data is received, ECs areas are updated and data is written onto tape. When the tape (holding data) is filled, it is replaced (as shown on Figures 3(a), 3(b) and 3(c)).

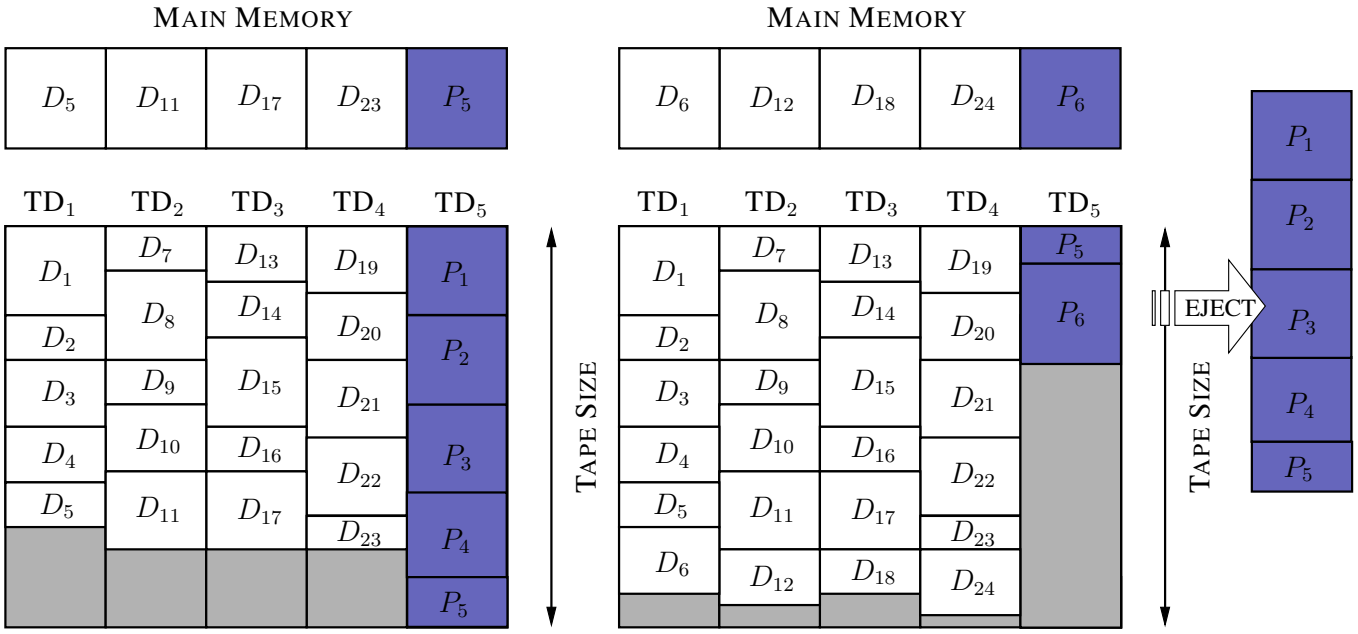
When the X^{th} data tape has been filled or there is no more data to be written (Figure 3(d)), the tape is replaced by the first tape dedicated to ECs. EC blocks are then written onto tape (as depicted on Figure 3(e)). However, similarly to the PARALLEL policy, it may happen that some EC tape gets filled before its entire EC block has been written. Whenever this happens, the tape is ejected and replaced by a new tape which will hold the remaining part of the EC block. This case is depicted on Figure 3(f). This step is then repeated for each remaining EC block.

It is clear that VERTICAL has a minimal impact on the level of parallelism of the system since it requires only one tape drive, regardless of the resiliency scheme used. Data is contiguous on tape and EC blocks need not be read when data is read.

However, this approach has several limitations: (i) the erasure code cannot be computed in mover memory because at least $Y+1$ entire tapes should fit in this memory, which is not possible. Therefore, these tapes must instead be stored on disks. Like the PARALLEL policy, EC blocks are written onto dedicated tapes, meaning that tape occupancy may be less balanced than with RAIT. (ii) although the entire system may be able to handle more requests concurrently, each request will take more time to complete since there is no data parallelism with VERTICAL. (iii) the tape drive is unavailable for application I/O when writing the EC blocks.

V. SCHEDULING ARCHIVAL REQUESTS

Scheduling I/O requests on a petascale platform is a hard task. Indeed, the tremendous number of parameters that need to be taken into account makes it challenging.



(a) Data is written until EC tape gets filled.

(b) Filled EC tape has been ejected, and replaced by a new one.

Figure 2. Writing data with PARALLEL policy with $X = 4$ and $Y = 1$.

We introduce an online scheduler which basically maps tape I/O requests submitted to the system onto a mover node. This scheduling process, denoted as MAIN-SCHEDULER in the following, works hand in hand with a load balancing process, denoted as LOAD-BALANCER, responsible for handling requests which were impossible to schedule by MAIN-SCHEDULER at the time of their arrival.

We choose a "dynamic" approach, where processes corresponding to different I/O policies are created on-the-fly onto the mover nodes, rather than being statically allocated. This creation process is done either by MAIN-SCHEDULER or LOAD-BALANCER whenever a new process is required.

The MAIN-SCHEDULER process works as follows: as soon as a request R is submitted to the system, it is handled by MAIN-SCHEDULER. MAIN-SCHEDULER first checks whether R can be served now, i.e., if there is no previous request(s) regarding the same file or, if R is a read request, if the tapes containing the concerned file are not currently in use. If R is in use, it is delayed and placed in the waiting list.

Requests are identified by their *type*, which is defined as their archival policy together with their resiliency scheme. For instance $(\text{PARALLEL}, 4+1)$ or $(\text{RAIT}, 8+2)$ are possible request types. If the request R can be scheduled, MAIN-SCHEDULER tries the following actions:

- first, MAIN-SCHEDULER tries to find a currently running process which matches the type of R. If such a process P exists, and if no more than MAXLIGHTLOAD requests are already scheduled onto this process, then R is mapped on process P;
- otherwise, MAIN-SCHEDULER tries to find a mover node having enough idle tape drives to host a new process for R, and it creates this process;
- then, if MAIN-SCHEDULER is unable to create a new process for handling R, it tries to schedule it on a currently running process P matching the type of R, but this time regardless of the number of requests already mapped onto P.

The rationale is to allocate requests to already running processes, provided that their load remains reasonable, otherwise it might be better to create new processes. The role of the system parameter MAXLIGHTLOAD is to tune the load threshold of the processes. Finally, if MAIN-SCHEDULER is still not able to schedule R, the request is delayed and placed in the waiting list.

In order to schedule the requests in the waiting list, as well as to keep the load balanced across the system, the LOAD-BALANCER is periodically executed every MINLBINTERVAL units of time. However, in order to keep the number of interventions of LOAD-BALANCER within a reasonable amount, one of the following conditions must be met:

- the oldest request has been delayed for more than MAXWAITINGTIME, and its file is not currently in use;
- the number of non-scheduled pending requests in the waiting list exceeds MAXREQCOUNT;
- the maximum *imbalance* of the system exceeds MAXIMBALANCE. Here, the *imbalance* is defined as the difference between the most and the least loaded types, were the load of a given type is the ratio between the number of requests and the number of processes of that type.

Whenever LOAD-BALANCER is triggered, it resets all pending requests and marks them as unscheduled. Only those

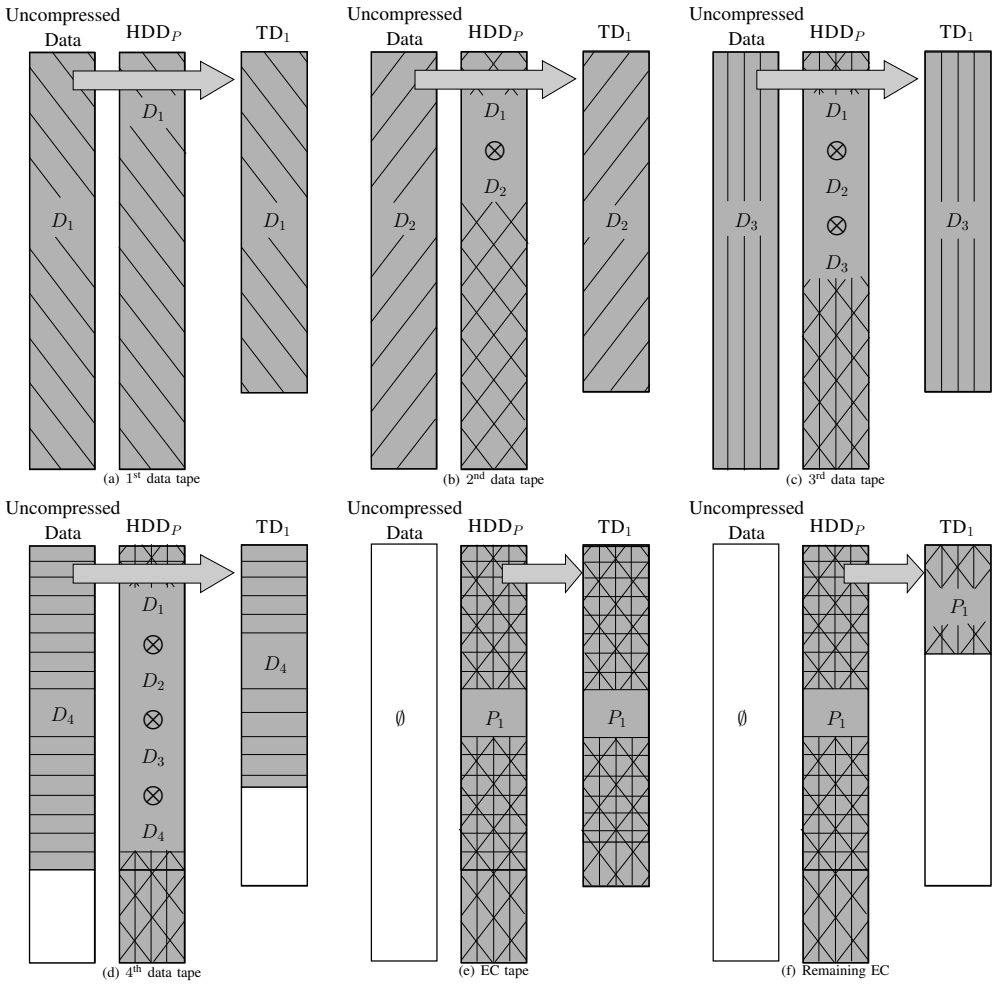


Figure 3. Writing data with VERTICAL policy with $X = 4$ and $Y = 1$.

requests that are currently executed are not modified (and continue their execution), but their processes are terminated, while all other existing processes are canceled. Then LOAD-BALANCER analyzes which process types are required by the set of unscheduled requests. For each required type, a new process is created on a mover node. Then, if idle tape drives able to host a process still remain, a new process is created, matching the type of the most loaded type. This action is then repeated until no new process can be created.

Once LOAD-BALANCER has created new processes, it tries to map each unscheduled request R. If the file concerned by R is not currently used, or, if R is a read request, if the tapes containing the concerned file are not currently in use, R is mapped on the least loaded process matching its type. Otherwise, R is delayed and placed in the new waiting list.

VI. PERFORMANCE EVALUATION

In order to assess the performance of each I/O policy, and the behavior of our I/O request scheduling algorithm, we have simulated an entire platform resembling that of a current petascale supercomputer. We first describe the environmental framework. We then conduct experiments where all requests obey the same archival policy (RAIT, PARALLEL or VERTICAL), and we discuss the influence of file sizes on the performance. Based upon the results of these experiments, we evaluate a scenario mixing policies, which associates the best-suited policy to each file size category.

A. Experimental framework

We have developed our own simulator using SimGrid [12], [13], a discrete event simulator framework, in its 3.5 version. The platform is simulated using distributed processes running in parallel on multiple virtual hosts. Each component of the model is represented by such processes. For instance, I/O policies running concurrently on a single mover are simulated by parallel processes on a single host, whereas each tape drive is represented by a host and a dedicated process. The same holds for the main scheduling and the load balancing processes, which are running concurrently on a single host.

The simulated platform is depicted on Figure 4. The user process simulates the arrival of the requests in the system. Those requests are handled by the MAIN-SCHEDULER process, which may create new I/O processes on the mover nodes

and assign tape drives to them. Unscheduled requests are handled by the LOAD-BALANCER process. Finally, the tape library controls multiple robotic arms dedicated to move the tapes back and forth from the library to the tape drives.

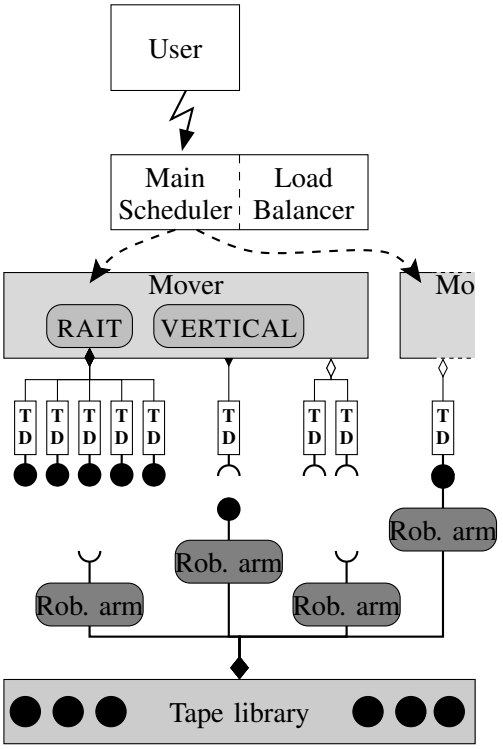


Figure 4. Model of the simulated platform

In the experiments, the platform is instantiated using one tape library managing 20 robotic arms and 500 tape drives. These tape drives are connected 10-by-10 to 50 mover nodes, responsible for handling I/O operations. These parameters match the size of today's petascale supercomputers.

$X+Y$	$1+0$	$2+1$	$3+1$	$3+2$
p_{X+Y}	0.025	0.025	0.05	0.1

$X+Y$	$4+1$	$4+2$	$6+2$	$8+2$
p_{X+Y}	0.1	0.3	0.2	0.2

Table I
RESILIENCY SCHEMES USED IN THE EXPERIMENTS.

The major challenge that we face for the evaluation is the lack of real traces of a storage system comparable with the one of BLUE WATERS. The BLUE WATERS machine is still being designed, and its use of tape storage significantly differs from that of existing supercomputers. Contrarily to such systems, tapes will be the main storage for BLUE WATERS, and discs will only be used as a cache. Hence, tapes will store large archival data as well as all data present in the file system. Thus, it is important to test the storage policies not only with large file sizes, but also for small and medium sizes. This is why we generated random workloads following a Poisson process with an arrival rate λ. File sizes are chosen according to a random log-uniform distribution, which is a simple approximation of the log-normal distribution of file sizes observed in file systems [14] for files larger than a few KB. The type of the I/O operation is chosen uniformly between read and write. A new file is created in 90% of the cases if the request is a write operation, and an existing file is written again otherwise. Each request is provided with a resiliency scheme $X+Y$, which is randomly chosen among a set of representative schemes. These schemes and their respective probability are given in Table I. Finally, for each file, the compression rate of data blocks C_D is chosen within $[1,3]$ while the EC blocks compression rate C_P belongs to $[1, C_D]$.

B. Results with a single policy

In a first step, only homogeneous scenarios are considered: requests may have different resiliency schemes but use only one I/O policy, either RAIT, PARALLEL or VERTICAL.

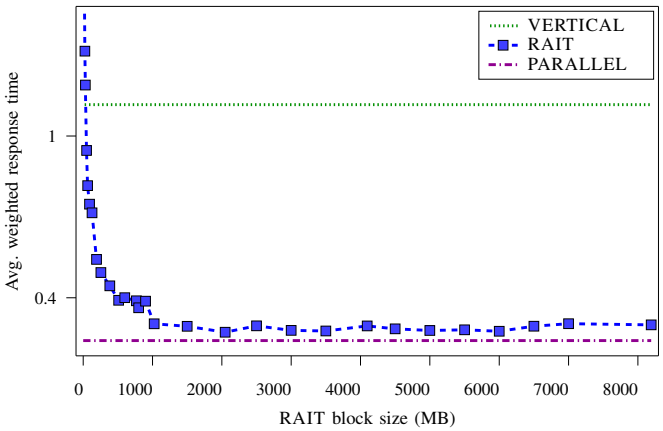


Figure 5. Impact of RAIT block size B on average weighted response time.

The first experiment aims at analyzing the impact of the block size B for the RAIT policy (contrarily to PARALLEL and VERTICAL, RAIT requires a block size to be tuned, since it impacts how data is written onto tape). The performance of RAIT is computed in terms of the average weighted response time. Request arrival rate is set to 180 requests per hour, and file sizes range from 1 GB to 1 TB, while B varies between 1 MB and 8 GB.

Results presented on Figure 5 show that B has a significant impact on the average weighted response time of RAIT. For the smallest values, RAIT performs worse than VERTICAL whereas it almost ties PARALLEL for larger values. With $B = 1$ MB, RAIT is about 80 times slower than with $B = 192$ MB, while the performance is constant between 192 MB and 8 GB. Altogether, this experiment outlines the importance of B value for RAIT policy, which clearly benefits from large enough blocks in order to offer competitive performance. In all the following experiments, B will be chosen according to these results.

The next experiment intends to compare the performance of all I/O policies for various arrival rates. The objective is

twofold : assessing the average performance of each policy, and in particular, determining the maximum arrival rate which can be handled by each policy. File sizes are chosen among three subsets: *small* file sizes range from 10 MB to 1 GB, *medium* file sizes from 1 GB to 100 GB, and *large* file sizes from 1 TB to 100 TB. Arrival rates are chosen with respect of these files sizes between a few requests per hour to hundreds of requests per hour.

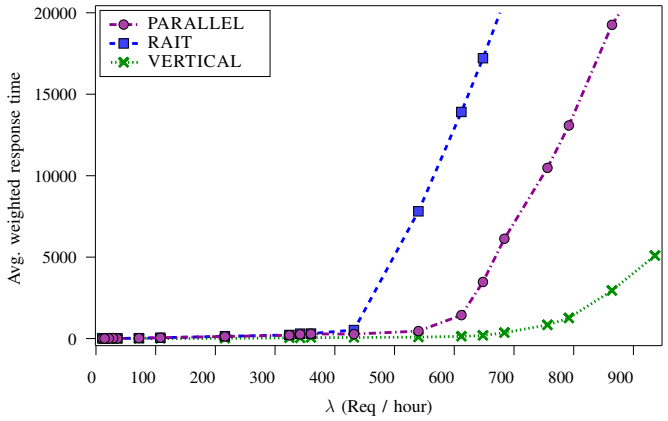


Figure 6. Impact of arrival rates on the average weighted response time for small files ($B = 16$ MB for RAIT).

For small files, as depicted in Figure 6, the best average weighted response time is offered by VERTICAL, which can sustain higher arrival rates than the other policies. RAIT is able to serve requests with reasonable response time for rates lower than 420 requests per hour. PARALLEL performs better than RAIT since it can keep up with rates lower than 600 requests per hour. The best policy in that case is VERTICAL, which can tolerate rates up to 800 requests per hour. This is due to the fact that with small files, extra latencies paid by data parallel policies (RAIT and PARALLEL) are not negligible. Also, recall that more files can be written concurrently throughout the entire system with VERTICAL. Contrarily to its contenders, VERTICAL can pipeline a large number of files before having to write EC tapes, thereby increasing average performance.

For medium sizes, results presented on Figure 7 show that, as expected, the system benefits more from data parallelism, latencies being now negligible. PARALLEL dominates other policies in this case, being able to handle up to 150 requests per hour while RAIT gets overloaded with arrival rates higher than 95 requests per hour. The extra tape drives used by RAIT as well as extra latencies when reading data have a significant impact on the average weighted response time. In that case, VERTICAL is the worst policy since it can only sustain up to 80 requests per hour.

Finally for large files, results presented on Figure 8 outline a behavior similar to that observed for medium sized files. PARALLEL represents the best policy in terms of average weighted response time. Using this policy, the system is able to cope with 1.3 requests per hour while the second competitor, RAIT, can only serve up to 1 request per hour without being

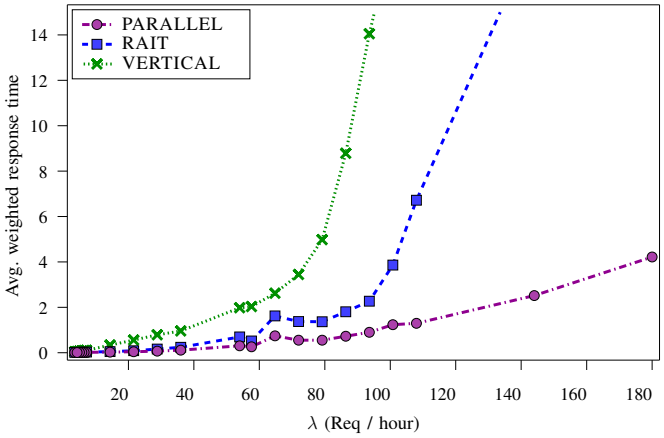


Figure 7. Impact of arrival rates on the average weighted response time for medium-size files ($B = 256$ MB for RAIT).

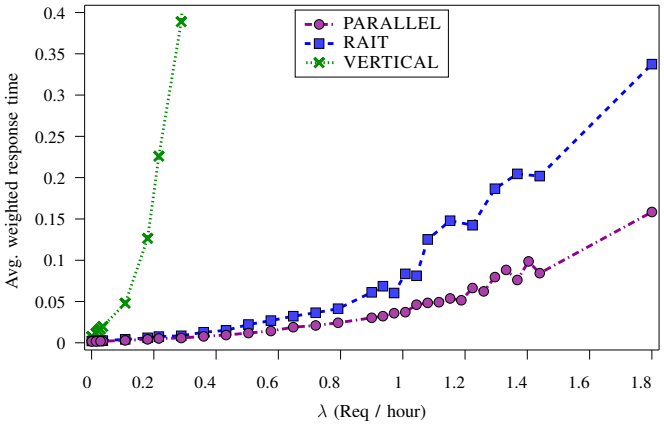


Figure 8. Impact of arrival rates on the average weighted response time for large files ($B = 8$ GB for RAIT).

overloaded. Unsurprisingly, VERTICAL suffers from the lack of parallelism with these huge files, and can only bear 0.1 request per hour.

As a conclusion, these experiments show that PARALLEL offers the best results overall. RAIT outperforms VERTICAL whenever files are large enough. The fact that PARALLEL dominates the other solutions means not only that data parallelism is crucial, but also (and less expectedly) that balancing parity across tapes (as in RAIT) has a negligible impact compared to that of enforcing data sequentiality.

The next experiment aims at measuring how the difference of compressibility between data and EC blocks affects the overall performance of each I/O policy. Indeed, neither PARALLEL nor VERTICAL can balance the EC blocks across tapes, possibly leading to more tape loads/unloads for tapes dedicated to EC blocks. The underlying objective of this experiment is therefore to assess the impact of this pitfall on the average weighted response time. In order to do so, the arrival rate is set to 65 requests per hour, RAIT block size is $B = 192\ MB$, and file sizes range between 1 GB and 1 TB. The compression rate of data is set to be 3x. The EC

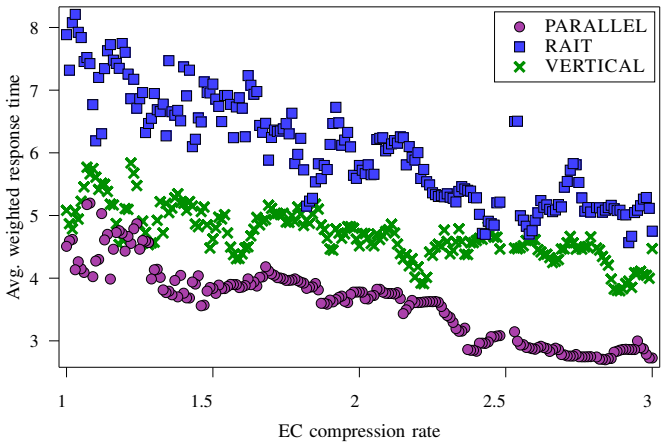


Figure 9. Impact of EC blocks compression rates on performance.

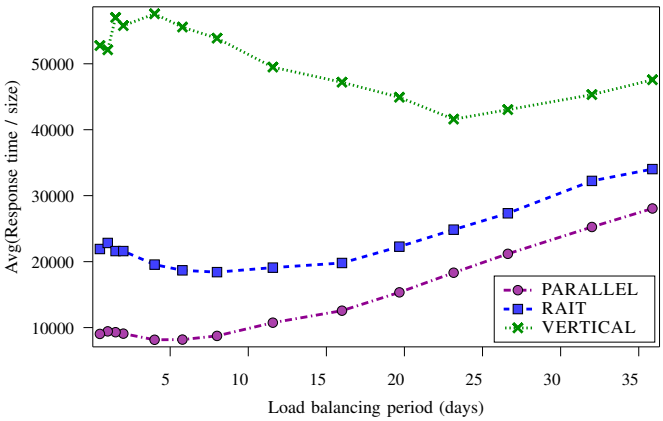


Figure 10. Impact of LOAD-BALANCER period on performance.

compression rate varies between 1x (no compression) and 3x (same compression rate as data).

As shown on Figure 9, I/O policies do not display the same sensitivity to EC compression rate. Both PARALLEL and RAIT are more affected by compression than VERTICAL. As a matter of a fact, while the later maintains its performance regardless of EC compression rate, the average weighted response time displayed by RAIT and PARALLEL increases when EC blocks are less compressed. The lower sensitivity of VERTICAL to EC compression comes from the fact that several requests are served before writing EC blocks, because these are aggregated. This is not the case for PARALLEL and RAIT.

Moreover, with both PARALLEL and VERTICAL, data tapes are always entirely filled, and EC blocks are written onto dedicated tapes. When EC blocks are far less compressed than data, tapes dedicated to EC are filled faster and require more frequent EJECT/LOAD operations. This also explain the higher sensitivity of PARALLEL to EC compression. On the contrary, with RAIT, the difference of compression between data and EC blocks degrades the balance of tape occupancy, leading to extra EJECT/LOAD operations on every tape drive (with RAIT, when a tape is filled, all loaded tapes are ejected).

Altogether, this experiment shows that both RAIT and PARALLEL display higher sensitivity to compression than VERTICAL. Whenever data is highly compressible, VERTICAL ties PARALLEL.

The following experiment focuses on the evaluation of the impact of the load balancing period MINLBINTERVAL on the average weighted response time. Indeed, this setting might significantly influence the behavior of the system under intensive workloads. Therefore, MINLBINTERVAL needs to be precisely tuned in order to fully exploit the archival architecture. In this experiment, file sizes are chosen between 1 GB and 1 TB, RAIT block size is $B = 192$ MB, while the mean arrival rate is set to $\lambda = 65$ requests per hour. Results depicted on Figure 10 show that the load balancing period indeed has a significant impact on performance: all three policies reach a minimum average weighted response time for a particular value of MINLBINTERVAL. Interestingly, the best value of MINLBINTERVAL is not the same for every policy: although both RAIT and PARALLEL perform better when LOAD-BALANCER is called at most every 6 days, VERTICAL benefit from a significantly higher value: 24 days. This is due to the fact that VERTICAL often has more pending requests than the other policies, and requires more time to serve each request. Remember that whenever an I/O process is destroyed by LOAD-BALANCER, loaded tapes are ejected. Therefore, calling the load balancing process too frequently may cause (in the worst case) tapes to be unloaded after each request has been served, leading to higher response times. All in all, this experiment shows that the load balancing period MINLBINTERVAL need to be precisely tuned in order to fully benefit from the parallel storage system.

C. Results with multiple policies

Based on the previous results, a novel strategy using multiple policies is introduced: HETERO. As seen above, VERTICAL represents the best solution when writing small files, while PARALLEL is the best choice for larger files. In the following experiment, the I/O policy used to write a file is now dynamically chosen by the system, based on its size.

The purpose here is to fully benefit from both policies in order to enhance the overall performance of the storage system. In order to assess the corresponding improvement, the impact of the arrival rate on the average weighted response time is again evaluated. File sizes are now chosen among a broader range: from 10 MB to 10 TB. A file smaller than 1 GB will be processed using VERTICAL, while larger files will use PARALLEL.

Results depicted on Figure 11 show that the HETERO strategy clearly outperforms all single policy strategies. HETERO can handle up to 7 requests of any size per hour while the best single policy strategy, PARALLEL, is limited to 0.6 requests per hour. The single strategy using VERTICAL does not perform well, since it is able to maintain a reasonable average weighted response time until arrival rate reaches 0.003

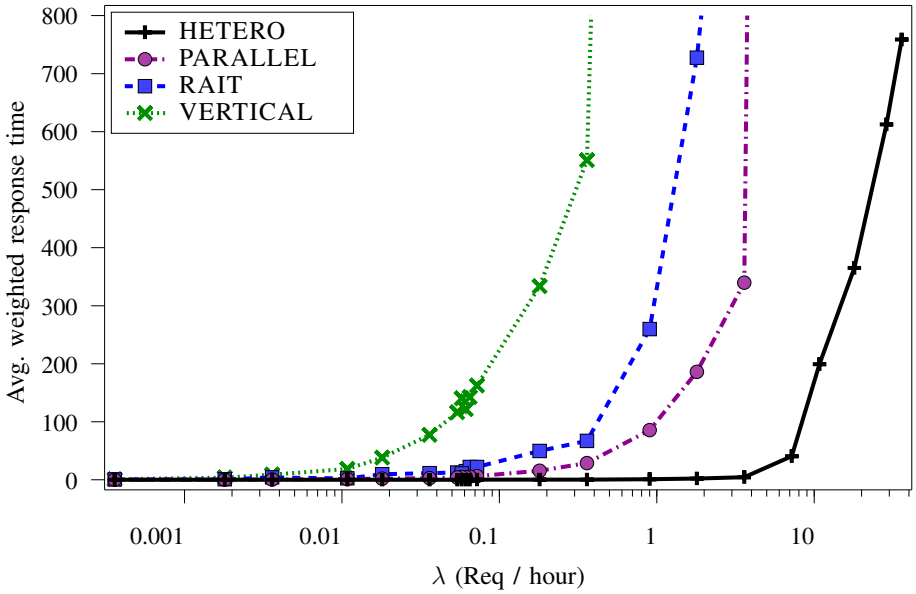


Figure 11. Impact of arrival rates on the average weighted response time (with small, medium-size and large file together, and $B = 512$ MB for RAIT).

requests per hour. RAIT does better with a maximum of 0.3 requests per hour.

Altogether, this experiment shows that a strategy using multiple policies, carefully choosing the I/O policy that will be used to handle a file based on its size, brings a dramatic performance increase. The performance is sustained at significantly higher rates than with any singly policy strategy.

VII. Conclusion

In this paper, we have first discussed the well-known RAIT policy for tape archival on a petascale supercomputer, and we have identified its shortcomings. We have introduced two new I/O policies, PARALLEL and VERTICAL, that either reduce file fragmentation, or increase the number of requests that can be served simultaneously, or both. Contrarily to RAIT which requires to carefully choose a blocksize, the new policies do not require any tuning.

We have conducted a comprehensive set of experiments to assess the performance of the three RAIT, PARALLEL and VERTICAL policies. We observed that for small files, VERTICAL provides the best weighted response time, while for medium-size and large files, PARALLEL is the clear winner. This has led us to propose an heterogeneous solution mixing policies (VERTICAL for small files, PARALLEL otherwise). Altogether, this latter approach provides a dramatic ten-fold improvement over each policy taken separately.

We hope that the lessons learnt in this study will help guide the final design decisions of the Blue Waters supercomputer, and more generally, of future large-scale platforms that will require even larger storage capacities, and always more efficient archival scheduling policies.

References

[1] Blue Waters, http://www.ncsa.illinois.edu/BlueWaters.

[2] J. Hick, "HPSS in the Extreme Scale," in *Report to DOE Office of Science on HPSS in 2018-2022, LBNL Paper LBNL-3877E, http://www.escholarship.org/uc/item/4wn1s2d3*. Lawrence Berkeley National Laboratory, 2009.

[3] J. Hughes, D. Fisher, K. Dehart, B. Wilbanks, and J. Alt, "HPSS RAIT Architecture," in *White paper of the HPSS collaboration, www.hpss-collaboration.org/documents/HPSS_RAIT_Architecture.pdf*, 2009.

[4] A. Drapeau and R. Katz, "Striped tape arrays," in *Mass Storage Systems, 1993. Putting all that Data to Work. Proceedings., Twelfth IEEE Symposium on*, Apr. 1993, pp. 257 –265.

[5] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the First USENIX Conference on File and Storage Technologies*, 2002, pp. 231–244.

[6] D. Teaff, D. Watson, and B. Coyne, "The architecture of the high performance storage system (hpss)," in *Proceedings of the Goddard Conf. on Mass Storage and Technologies*, 1995, pp. 28–30.

[7] R. W. Watson, "High Performance Storage System Scalability: Architecture, Implementation and Experience," in *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005)*. IEEE press, 2005.

[8] T. Johnson and S. Prabhakar, "Tape group parity protection," in *Mass Storage Systems, 1999. 16th IEEE Symposium on*, 1999, pp. 72 –79.

[9] B. Hillyer, R. Rastogi, and A. Silberschatz, "Scheduling and data replication to improve tape jukebox performance," in *Data Engineering, 1999. Proceedings., 15th International Conference on*, Mar. 1999, pp. 532 –541.

[10] S. Prabhakar, D. Agrawal, A. El Abbadi, and A. Singh, "Scheduling tertiary i/o in database applications," in *Database and Expert Systems Applications, 1997. Proceedings., Eighth International Workshop on*, Sep. 1997, pp. 722 –727.

[11] I. Pruhs, J. Sgall, and E. Torng, "On-line scheduling," in *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, J. Leung, Ed. CRC Press, 2004, pp. 15.1–15.43.

[12] SimGrid, URL: http://simgrid.gforge.inria.fr.

[13] A. Legrand, L.Marchal, and H. Casanova, "Scheduling Distributed Applications: The SimGrid Simulation Framework," in *Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003, pp. 138–145.

[14] A. B. Downey, "The structural cause of file size distributions," in *9th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2001)*, 2001, pp. 361–370.