2009 28th IEEE International Symposium on Reliable Distributed Systems (SRDS '09).

Co-scheduling of disk head time in cluster-based storage

Matthew Wachs, Gregory R. Ganger
Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA, USA
{*mwachs,ganger*}*@ece.cmu.edu*

***Abstract*—**
Disk timeslicing is a promising technique for storage performance insulation. To work with cluster-based storage, however, timeslices associated with striped data must be co-scheduled on the corresponding servers. This paper describes algorithms for determining global timeslice schedules and mechanisms for coordinating the independent server activities. Experiments with a prototype show that, combined, they can provide performance insulation for workloads sharing a storage cluster — each workload realizes a configured minimum efficiency within its timeslices regardless of the activities of the other workloads.

***Keywords*-performance isolation; quality of service; shared storage; performance; clustering; approximation algorithms; heuristics; strip packing**

I. Introduction

Employing a single storage infrastructure for multiple applications, rather than distinct ones for each, can increase hardware utilization and reduce administration costs. But, interference between application I/O workloads can significantly reduce the overall efficiency of traditional storage infrastructures, due to disruption of disk access locality and cache access patterns.

Recent work has provided mechanisms to insulate workloads sharing a single storage server such that each realizes a specific fraction (called the "R-value") of its standalone efficiency [22]. That is, efficiency loss due to interference is bounded to $(1 - R)$, regardless of what other workloads share the server. To provide this insulation guarantee, disk head time and cache space are explicitly partitioned among workloads. Each workload receives an amount of cache space and a timeslice of disk head time that bounds the impact of interference. Disk head timeslicing is akin to process scheduling on CPUs. Each timeslice is long enough to complete many requests, amortizing any overheads of switching between disk workloads.

Although effective for the single server case, disk head timeslicing faces difficulties for cluster-based storage. Specifically, when data is striped across multiple servers, a client read request can require a set of responses. Until all are received, the read request is not complete. Since each server only acts on a disk request within the associated workload's disk head timeslice, the client will end up waiting for the last of the timeslices. If the relevant disk head timeslices are not scheduled simultaneously, the delay could be substantial (5–7X in our experiments). Worse, if the disk head timeslices do not all overlap, the throughput of a closed-loop one-request-at-a-time workload will be one request per round of timeslices.

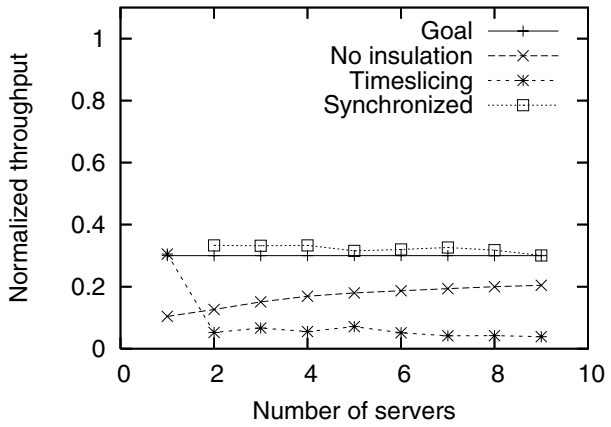

Figure 1. **Moving beyond one server.** Three workloads share a server and the throughput of one, which has been allocated a one-third share of disk time, is graphed. Without performance insulation, the other workloads interfere and the goal is not met. Timeslicing disk head time solves the problem on one server. But when the workloads are striped across servers, timeslicing becomes ineffective unless the timeslices are synchronized. (The y-axis is normalized against the throughput the depicted workload receives running alone on the corresponding number of servers.)

Providing performance insulation for cluster-based storage requires co-scheduling of each workload's disk head timeslices across the servers over which its data is striped. For cluster-based storage systems that allow different volumes to be striped differently (e.g., over more servers, over different servers, or using just parity or a 2-failure correcting code) [1], [23], there is the additional challenge of finding a schedule for the cluster.

This paper describes a cluster-based storage system that guarantees R-values to its workloads. To do so, it extends the mechanisms from the single-server previous work with explicit co-scheduling of disk head timeslices. It uses standard network time synchronization and synchronizes "time zero" for the disk head time scheduler. It implicitly coordinates the

work done by each server in co-scheduled timeslices, while allowing local request ordering decisions within timeslices. It assigns workloads to subsets of servers and organizes their timeslices to ensure that each striped workload's disk head timeslices are co-scheduled. Finding an assignment that works is an NP-complete problem, but there exist heuristics that allow solutions to often be found quickly. Although each of the heuristics yields quick answers in only 40–80% of cases, we find that running several in parallel yields a quick solution in over 95% of cases. When a quick solution cannot be found, introducing a small co-scheduling efficiency reduction helps.

Experiments confirm that, by adopting a global timeslice schedule found with heuristics and ensuring the servers are synchronized as they follow it, workloads receive the insulated efficiency expected. Efficient access during a controllable share of server time provides performance that is predictable and controllable.

II. Motivation

Shared storage provides a number of potential benefits over dedicated storage systems for each workload. For instance, spare resources can be made available to whichever workload is experiencing a surge in required capacity or throughput at a given point in time.

Unfortunately, interference between workloads often occurs when they share a storage system. Competition for a disk head can decrease the efficiency with which a workload's requests are completed. This occurs when the interleaving of access patterns results in a combined request stream that reduces their locality. For instance, when two active, streaming workloads share a disk, the combined sequence of requests at the disk often results in excessive seeking between the two workloads. This can reduce their throughput by an order of magnitude, even if disk time is divided evenly between the two workloads.

A. Performance insulation

This paper focuses on providing *performance insulation* [22] between shared workloads on a cluster-based storage system. Insulation preserves the efficiency that individual workloads receive when they share a storage system. Such insulation has been demonstrated in a single server, called Argon [22]. Argon takes a configuration parameter termed the *R-value* and tunes its explicit disk and cache sharing policies so that any workload's efficiency in the shared system is no less than that fraction (between 0 and 1 exclusive) of the workload's efficiency when it runs alone. For instance, suppose $R = 0.9$ and a workload has been allocated a one-third share of the server. If its standalone performance on that server would be P_{alone}, then Argon will guarantee its shared performance is no less than $0.9 \cdot \frac{1}{3} \cdot P_{alone}$. This guarantee is provided regardless of the characteristics of any other workloads on the same system.

One of the techniques used by Argon is round-robin disk-head timeslicing, which provides each workload stretches of uninterrupted disk time similarly to standard CPU multiplexing. Each workload in the system receives a dedicated timeslice. Timeslices are long (e.g., 140 ms) to allow workloads to maintain most of their spatial and temporal locality. Argon uses long-lived, stable round-robin timeslice schedules. Workloads are assigned fractions of total server time (e.g., $\frac{1}{3}$) sufficient to meet their performance needs, and timeslices are proportionally sized to satisfy these assignments. Each round, workloads are asssigned their usual timeslice regardless of how many or how few requests they have "in flight" at the time. Similarly, timeslices are not tailored to the specific requests a workload will issue in a particular round. An idle workload's timeslice is not skipped or shortened; this ensures that it will not be penalized if, for instance, it becomes active shortly after the beginning of its timeslice. The other workloads already receive the fraction of server time they need to achieve their performance goals within their own timeslices. Reappropriating idle time would only improve their performance beyond their goals.

B. Cluster-based storage

Argon is a standalone storage server. But, modern storage is increasingly provided by combining the resources of multiple servers. Doing so can provide significant benefits over standalone storage servers. For example, each server need provide only a fraction of required performance. Likewise, fault tolerance can be provided by storing redundancy data across servers rather than expensive engineering of one server. Cluster-based storage systems typically stripe data across servers, breaking large data blocks into smaller *fragments* that are placed on different servers. Striping so allows clients to increase bandwidth via parallel transfers to/from multiple servers.

Our goal is to construct a cluster-based storage system that provides the same guarantees as Argon (a standalone server). One design might be to just "run Argon" as each of the individual servers — each server would provide guaranteed efficiency for the fragments it is storing. The question that arises, however, is *how do these per-fragment efficiency guarantees compose into the block-level guarantees desired for a workload*?

Issues with timeslicing: With striping, a client request for a block translates to a fragment request at each server. The client request is not fulfilled until all its fragments are read and combined back into the original block. Unfortunately, the throughput for a block is not simply the sum of the throughputs for its fragments. While that sum is an upper bound, the response time observed by a client is that of the

slowest server to respond.[1] For closed workloads, slower response times directly reduce throughput.

Timeslicing can cause significant response time differences across servers in two ways. First, the list of workloads, ordering and length of timeslices, and overall round length (time before the schedule of timeslices repeats) may not match across the servers. Without arranging the schedules for each of the servers that a workload uses, its timeslices may not consistently "line up" across the servers. Second, even if the schedules are designed to co-schedule the timeslices of each workload, the phase of the servers (i.e., the point in wall-clock time when the round-robin order restarts) may not match (see Figure 2). This can occur either because no attempt was made to synchronize them, or because the servers diverged.

If these two issues are not mitigated, performance can be worse with timeslicing than without. In Figure 2, for instance, a single-threaded workload would only complete one request per round because there is no overlap between the corresponding timeslices at the servers — by the time the request completes at the second server, the timeslice is over at the first server.

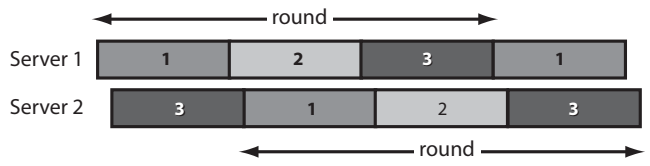

Figure 2. **Out-of-phase servers.** If the phase of identical rounds does not match across the servers, then a client must wait for the most-in-the-future occurrence of its timeslice across all the servers before its request completes.

III. Designing a schedule

To minimize administrator effort, we wish to have an automated means of creating timeslice schedules and changing them when workloads enter or leave the system. This section describes and evaluates algorithms for schedule design.

A. Problem specification

We assume a cluster of homogenous storage servers with one disk each.[2] Machines that have significantly different resources (i.e., disrupt homogeneity) should be placed in a separate cluster with like machines.[3]

Workloads are specified using two numbers: number of servers and share of servers. For instance, a workload may be striped across five servers and, to achieve the desired level of performance, need at least a one-third share of time on each of the five servers.

Finding a suitable schedule for the cluster amounts to finding round-robin timeslice orderings for each of the servers so that each workload receives its share of time at the required number of servers, and each workload's timeslices are co-scheduled. The system may select whichever servers are convenient for a given workload.[4]

Finding a solution should be relatively efficient (e.g., a few minutes or less), but need not be extremely fast, in practice. As described in Section II-A, timeslice schedules are long-lived because they are a function of the workloads, not of specific requests. A new schedule need be found only when workloads enter the cluster, or when the fraction of server time assigned to a workload is changed. Adding a new workload involves the substantial task of adding a new dataset, which makes it a time-consuming operation already. Rescheduling due to workload removal need not be fast, since existing workloads are already scheduled. Changing a workload's allocation usually occurs only early in a workload's lifetime, as the QoS control system determines the appropriate allocation. Most workload changes do not require a change in the schedules, since they affect only what a workload does within its timeslices, not the ability to maintain the R-value. Thus, our target is algorithms that can find solutions within a few minutes. Nonetheless, our algorithms parallelize very well and can be completed in seconds.

Figure 3 shows an example input list of workloads and an example solution to the problem.

B. Geometric interpretation

This problem may be recast as a geometric problem, as suggested by Figure 3. Consider each workload as a rectangle whose height is the number of servers it needs, and whose width is the fraction of time it needs on each of those servers. Consider a larger rectangle, whose height is the total number of homogenous servers in the cluster, and whose width is one (corresponding to the full share of time on a server). Can the set of smaller rectangles be placed into the larger rectangle without overlapping or rotating any of the smaller rectangles, or exceeding the boundaries of the larger rectangle? If so, then an appropriate schedule exists, and the rectangle placements can be directly translated into a suitable timeslice schedule.

[1] With a more flexible data distribution scheme — such as erasure coding — the client can read from any m of the n servers storing its data. While the request response time may no longer be that of the slowest server, it is still limited by the $n - m + 1^{st}$ slowest server.

[2] Machines that have multiple, independent disks can be thought of as separate machines in the context of this problem. From our perspective, machines with disk arrays can be treated as having a single, bigger disk.

[3] If a set of machines has minor variations, however, they may be clustered together and treated as all having the "lowest common denominator" of performance. Thus, for example, minor differences in disk characteristics (e.g., due to in-field replacement) are fine.

[4] There may be additional constraints, such as not exceeding the storage capacity available at a server, or perferring certain placements due to network topology, that we do not consider here.

Total servers = 7	
Num servers	**Proportion**
3	1/2
3	1/2
1	1/2
1	1/6
1	1/3
5	1/3
7	1/6

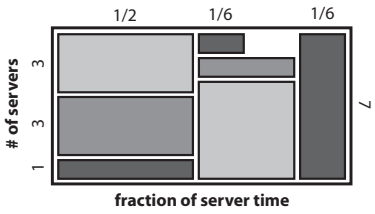

Figure 3. **Example problem instance and solution.** On the left is an example input to the scheduling algorithm; on the right is one possible solution. Rectangles correspond to workloads, with their height corresponding to the number of servers and their vertical location corresponding to which servers to use; their width corresponds to share of time, and their horizontal location corresponds to the span of time during which their timeslices are scheduled. The enclosing rectangle represents a single round in the cluster; the schedule is repeated indefinitely.

This geometric problem is known as the *strip-packing problem* or the *cutting-stock problem* [7]. It has been studied in industrial settings where a larger piece of material must be cut into smaller pieces to manufacture a product. Unfortunately, it is known to be NP-complete. The above formulation is the *decision* version of the problem, which asks *can the rectangles all fit in a larger rectangle of a given size?* The *optimization* version asks *what is the minimum width of the larger rectangle, for a fixed height, such that all the rectangles fit?* An *optimal* solution of a given problem is a packing of rectangles that uses the minimal possible width.

For our purposes, a solution that has a width of at most one is sufficient to provide all workloads their requested share of the server. It may be preferable to find a solution that is even narrower, if one exists, because it would give the workloads extra server time; but this is not necessarily to meet the insulation goals.

C. Related work

Strip packing has been explored by theoreticians seeking ways to accelerate searches for the optimal solution, for approximation algorithms that can provide solutions within a guaranteed distance from optimal, and for heuristics that may find "acceptable" solutions quickly.[5]

1) Exhaustive search: As with any NP-complete problem, one can enumerate all possible solutions and test whether they meet the requirements. However, this approach results in an exponential run time that will be prohibitive for sufficiently large problems.

In the case of strip packing, a naïve search might consider placing a rectangle at each possible coordinate location.[6] However, Fernandez de la Vega and Zissimopoulos [4] show that it suffices to consider a smaller (but still exponential) set of possible solutions. If there is a solution, then there exists a *left-bottom justified* solution — essentially, one in which there are no gaps. Thus, one need only consider putting rectangles at *active corners*, i.e. touching other rectangles or the boundaries.

2) Approximation algorithms: Approximation algorithms exist [4], [12] that are able to find solutions within $(1 + \epsilon)$ of optimal in time that is polynomial in the number of rectangles but exponential in other variables. Unfortunately, for our problem sizes, we believe that no point along the runtime-vs.-ϵ tradeoff will be acceptable (too much error is introduced and runtime remains high).

3) Heuristics: There are various heuristic techniques that are sometimes able to produce a solution quickly. They do not guarantee the optimality of the solution, nor whether their inability to find a solution indicates there is none. However, they can often be very effective in practice. In our case, we only need to find a solution with a width ≤ 1; optimality may not be important.

In a similar vein to Fernandez de la Vega and Zissimopoulos's use of left-bottom justified solutions, a series of heuristic methods [2], [8] will try simply placing rectangles in a fixed order one after the other as far to the bottom and left as possible. If holes that have been surrounded by already-placed rectangles can still be filled, this is known as Bottom-Left-Fill (BLF). Once all rectangles have been placed, the solution is checked to see if it exceeds the width limit. While BLF places the rectangles in whatever order was specified in the problem description, other orderings may be more successful. BLF-DW sorts the rectangles in order of decreasing width; BLF-DH by decreasing height. Lesh *et al*. [14] suggest sorting by decreasing area (BLF-DA) or perimeter (BLF-DP).

Lesh *et al*. [14], [15] further suggest that, if one of these heuristic methods does not work, it may still be possible to find a solution quickly by spending a limited amount of time randomly searching small perturbations to the initial heuristic orderings, a technique they name BLD*.

D. Relaxing the problem

Argon maintains a guaranteed level of efficiency, expressed by the R-value, when a single server is shared; we extend that concept to the clustered setting. We refer to the R-value being enforced at a particular server as R_{server}. Just as it may not be possible or practical to share a single server with perfect efficiency, it may not be reasonable to cluster with perfect efficiency. Thus, we introduce a second R-value, $R_{clustering}$, which represents the minimum efficiency maintained by the clustering scheme. The R-value observed at the client, then, is

$$\geq R_{clustering} \times \min_{servers} R_{server}$$

The R-value at the client is the ultimate indication of whether insulation has been achieved; $R_{clustering}$ and R_{server} are

[5] By "acceptable," we mean solutions that are not even guaranteed to be necessarily close to optimal, but might be subjectively good enough.

[6] When we refer to *placing* a rectangle at a coordinate location, we mean placing a specific corner at that location.

not externally visible and can be manipulated for the convenience of the storage system.

E. Our approach

We perform strip-packing to generate a schedule for the cluster as follows. We start by attempting to achieve $R_{clustering} = 1.0$. We create four parallel threads to attempt four different heuristics (BLF-DW, -DH, -DA, -DP). If the heuristic orderings do not lead to a solution, then we try nearby points in the solution space by permuting the heuristic sort orders.[7] Each of the threads explores orderings near its initial sort order. In our experience, no specific sort order is always a good starting point for this exploration, but one of them usually is. When a thread finds an acceptable solution, the other threads are halted and the solution is used. Figure 4 shows pseudocode specifying the exact sequence in which we search the solution space near a heuristic ordering. If a solution cannot quickly be found, we relax $R_{clustering}$ (for instance, we might next try $R_{clustering} = 0.95$) and repeat.

Initial setup: The inputs to our algorithm are the number of servers in the cluster and the list of workloads, each of which is described by a number of servers and a fractional share for each server. We create small rectangles corresponding to each of the workloads, as described in Section III-B. We create a large rectangle to represent the cluster with the height equal to the number of servers in the cluster and the width equal to $1/R_{clustering}$.

The width of the larger rectangle represents the period over which the schedule repeats; each workload's rectangle consumes a fraction of that round length. If $R_{clustering} = 1.0$ then, when we pack the workloads, a workload requesting a particular share of a server will receive that proportion of time. If $R_{clustering} < 1.0$, however, then the round length will be scaled up without scaling up the workloads. This creates more space into which to pack the rectangles, which may make the problem faster to solve or possible to solve where the original one was not. But, the workloads receive a smaller share of the round, resulting in a fractional decrease in performance equal to $R_{clustering}$.

F. Evaluation

To evaluate the efficacy of creating a timeslice schedule using our approach, we created a number of random problem instances representing sets of storage workloads. Our results show that exhaustive search is impractical; that starting exhaustive search with any one of the heuristic orderings does not improve mean solution time significantly; but that our approach of trying multiple heuristics in parallel and exploring nearby solutions does result in much faster solutions in the mean. Furthermore, for problems too large to solve even with this approach, relaxing the value of $R_{clustering}$ can create an easier-to-solve version of the problem.

[7]For instance, sorting the rectangles in decreasing width order may yield the heuristic ordering {A, B, C, D, E}. If placing rectangles in this order does not yield a satisfactory solution, we might next try the ordering {B, A, C, D, E}.

```
sched = blank schedule
for i = 1 to number of distinct types of workloads do
  remaining[i] = number of workloads of type i
call place_next_workload(sched, remaining)
exit with "No schedule found"

place_next_workload(sched, remaining)
  if remaining[i] = 0 for all i then
    exit with "Schedule found", sched
  for i = 1 to number of distinct types of workloads do /* L */
    if remaining[i] > 0 then
      /* Build a sched. by placing a workload of type i next */
      for s = 1 to number of servers do
        for t = beginning of round to end of round do
          if (s, t) is an active corner in sched
          and a workload of type i fits at loc. (s, t) in sched then
            sched2 = sched
            remain2 = remaining
            Place a wl. of type i at loc. (s, t) in sched2
            decrement remain2[i]
            call place_next_workload(sched2, remain2)
            /* Only consider the first placement that fits */
            skip to next iteration of loop L
```

Figure 4. **Search ordering.** If the heuristic placement methods do not immediately find a solution, nearby solutions are searched as specified by this algorithm. The particular heuristic being used affects the ordering of the types of workloads in the above code. For instance, if Decreasing Width is used, "workloads of type 1" refers to those with the greatest width.

1) Experimental setup:

Problem instances: To create a large number of workload sets, we generate lists of storage workloads randomly. A range of list sizes is generated to evaluate the growth of computation time as the number of workloads grows. For each workload, we choose the number of servers on which to store the workload uniformly at random from among the numbers 1, 3, 5, 7, and 9.[8] For each workload, we also choose uniformly at random the proportion of the servers' time it needs from among the fractions 1/2, 1/3, 2/3, 1/4, 1/5, and 1/6.

An appropriate cluster size is calculated by summing the areas of the workloads' rectangles and choosing a number of servers so that the cluster's rectangle has that area, rounded up (before adjusting for $R_{clustering}$). This corresponds to the cluster size that would have the least wasted resources for that set of workloads. If one or more of the workloads needs more than the computed number of servers, the cluster size is increased to match.

We used this procedure to generate 1000 random problem instances of each size depicted in the figures, with the exception of some of the $R_{clustering} = 0.9$ cases to keep

[8]These values are typical of threshold quorum schemes used for erasure coding, where an odd number ensures a majority exists for a bipartition of servers.

experiment time manageable; for 14, 20, and 30 workloads, we used 500 random problem instances and for 40 workloads we generated 200 problem instances.

Hardware: All experiments were performed on machines with Pentium 4 Xeon 3.0 GHz processors running Linux 2.6.24. The memory footprint of strip packing is small, so memory and storage were not bottlenecks.

2) Results:

Exhaustive method: We use the exhaustive method to solve these problem instances. Search times grew exponentially with the number of workloads. For ten workloads, they averaged around two seconds; for eleven workloads, around 18 seconds; for twelve, around 140 seconds; and for thirteen workloads, around 1250 seconds (about 20 minutes). Beyond thirteen workloads, exhaustive search was impractical. For instance, one fourteen-workload instance took about 6 hours.

Individual heuristics: We also tried applying the decreasing width, decreasing height, decreasing area, and decreasing perimeter heuristics to the problem. If a solution was not found by one of the heuristic orderings, we continued to explore the solution space, starting near the heuristic orderings, until a solution was found or the entire solution space had been exhausted, as described in Section III-E. No single heuristic improved the mean time to solution; search time averages over the sets of workloads of each size for each heuristic were indistinguishable from the unsorted exhaustive search.

Our approaches: Figure 5 shows solution speed with our approach — parallel execution of the four heuristics, continuing to search the solution space if a solution is not found. A timeout value, described in the next section, is used to terminate the search and return "no solution found" after a period of time. The plotted times correspond to running the four threads on a single CPU, each at quarter-speed. This represents a pessimistic run time; one might use CPUs in the storage cluster itself to perform this search in parallel, or a multi-core machine.

Figure 6 shows, for the $R_{clustering} = 0.9$ case, the proportion of problems solved for each of the four heuristics (and continued search of nearby orderings for a limited period of time) normalized against the number of problems solved using our approach. No one heuristic is sufficient to solve all of the problems in a reasonably short period of time. Our approach improves solvability by allowing whichever of the four is best for a particular problem to be used without knowing which it will be.

Figure 7 shows the tradeoff between $R_{clustering}$ and runtime for problems of size 13. Relaxing $R_{clustering}$ not only reduces run time, but also makes more of the randomly-generated problem instances solvable. To maintain the desired overall R-value, however, this approach would then incur the cost of increasing R_{server} to compensate.

Solvability and timeouts: We observed that our approach either finds a solution relatively quickly, or not at all, suggesting that we should halt the search after a period of time. To determine appropriate timeout values for each problem size, we ran all of the randomly-generated problem instances of that size without a timeout. Initially, many runs find a solution and terminate. Over time, we observed a decline in "completions" until a negligible or zero number of completions occurred per minute, at which point we halted the experiment. We then used the 90th percentile of these

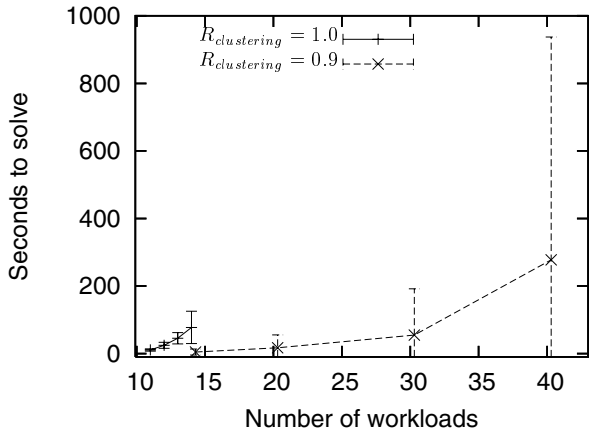

Figure 5. **Parallel heuristic search.** Running searches initialized with different heuristics in parallel allows the best-performing heuristic for a particular problem to find a solution faster. For the $R_{clustering} = 1.0$ case, however, an adequate sample size could not be achieved for problems of size 14 and greater due to growing run time. Relaxing $R_{clustering}$ further accelerates the search and makes larger problems tractable. The error bars show one standard deviation.

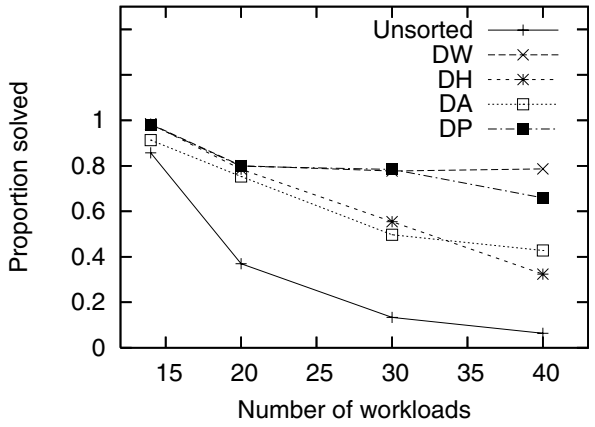

Figure 6. **Proportion of problems solved per heuristic.** In a limited amount of time (one minute for 14 and 20 workloads, two minutes for 30 workloads, and nine minutes for 40 workloads), initializing a search with one heuristic is not able to solve all of the problems that trying the combination of heuristics in parallel is able to solve. The proportion of problems solved is normalized against the number of instances solved using the parallel approach. While the performance of DW and DP is similar and both do well, the two alone are not sufficient.

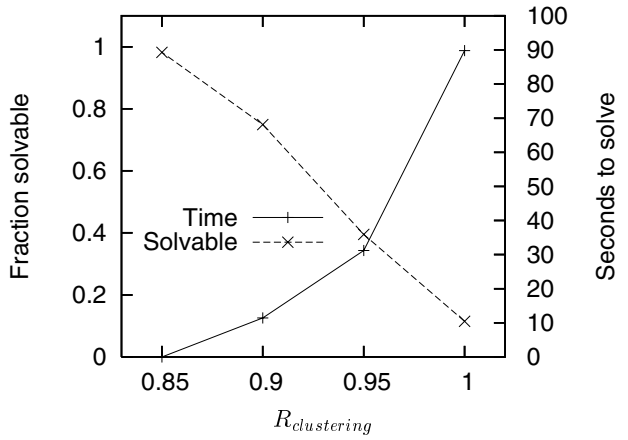

Figure 7. **R-value tradeoffs for 13 workloads.** Relaxing $R_{clustering}$ can accelerate solution speed and increase the number of solvable instances.

completion times (rounded up to the nearest second) for the timeout value for that size problem instance. This has the effect of sacrificing the ability to find a small number of solutions with outlying run times. For $R_{clustering} = 0.9$, timeouts were one second for problem sizes under twelve workloads; for twelve workloads the timeout was 2 seconds; for thirteen workloads, 5 seconds; for fourteen, 6 seconds; for twenty workloads, 30 seconds; for thirty, 114 seconds; and for forty workloads, 524 seconds.

Since many of these problem instances are too large to exhaustively solve, we cannot determine for certain that solutions do not exist beyond those we found. However, our experience with smaller instances, where we *can* compare the number of solutions found by our technique to exhaustive search, suggests that this strategy results in finding the vast majority. For instance, no further solutions were found by running to exhaustion the thirteen-workload problem instances that did not find a solution before we halted the original experiment; the percentile used to set the timeout directly determines what percent of problems will be solved in this case.

Parallelization in the cluster: Our experiments show that the single-CPU parallel heuristic search achieves our "few minutes or less" target. But, searching for a solution to the scheduling problem is an "embarrassingly parallel" problem which can be split up across a virtually unlimited number of CPUs. Thus, any CPUs available in the storage cluster for which the schedule is being computed can be exploited. For $R_{clustering} = 0.9$, if one CPU per server in the cluster is used to parallelize the search, then on average problems with twenty workloads take less than half a second, problems with thirty workloads take about a second, and problems with forty workloads take under four seconds.[9]

[9]Recall that the number of servers varies across problem instances.

IV. Coordination among servers

Once an appropriate schedule has been determined, the servers must follow it in a synchronized fashion. This section describes the requirements associated with such coordination and the solution we adopted.

A. Requirements

The servers must begin following the schedule of timeslices at approximately the same moment of wall-clock time.[10] Once begun, the servers must start each subsequent timeslice in the schedule at the same approximate time. If a timeslice begins late at one server for some reason, the next timeslice must either begin on time, or the servers must all retard the beginning of the next timeslice by the same amount.

In addition to coordinating the timeslices across the servers, there are other decisions made by the servers that impact a workload's performance if not also coordinated. For example, if a workload has more requests queued than can be handled in a single timeslice, the servers must choose which subset of requests to send to disk. Only those requests chosen by all the servers will complete at the client that round. But, notice that the issue here is the set of requests completed rather than the order in which they are completed — request scheduling within each timeslice does not need to be synchronized and can be local to each server, allowing each server's low-level disk scheduling to remain unchanged.

The number of requests per timeslice: Requests arrive at our storage server via a custom protocol before being turned into I/O system calls. We perform timeslicing between workloads by queueing requests in the user-level storage server, then issuing the corresponding system calls once a workload's timeslice begins. However, we are not able to cancel a request that has been sent to the kernel or disk; this complicates the implementation of our scheduler. Timeslices establish a period of time during which a workload's requests can execute, but the number of requests that can be executed during that period is not known with certainty in advance. If we send more requests than would fill the timeslice, we would delay the beginning of the next timeslice. If we send fewer requests than could fit, we would waste time.[11] Thus, we estimate how many requests would fill the timeslice based on historical observations of a given workload. We then send exactly that many to the disk, provided enough requests have been queued by the client.

Variations in disk service times may make the historical observations different across the servers. If one server issues more requests in a timeslice than the others, the client will still have to wait for the other servers to complete the extras.

[10]Because timeslices are long, e.g. 140 ms, small offsets among the servers — say, 0.5 ms — will not be significant.

[11]We could "trickle" additional requests to the disk until the timeslice ends, but this conservative strategy can hurt workloads that benefit from disk scheduling optimizations for concurrent requests.

If one server issues fewer requests than the others, on the other hand, it impedes client performance. Thus, care must be taken so that the servers issue approximately the same number in a given timeslice.

B. Symmetric operation

We prefer not to need a central coordinator. Our method for achieving this is to make decisions independently at each of the servers in such a way that they usually will agree across the servers without explicit coordination. We call this approach *symmetric operation*. We designed our servers to avoid central coordination of the beginning and end of timeslices, the number of requests to issue in a timeslice, and which specific requests to issue in a timeslice if there are more queued than can be issued.

Timeslices are co-scheduled by using `ntp` [17] to keep wall-clock time synchronized across the cluster. The management tool that determines the overall schedule of timeslices in a cluster also assigns a fixed wall-clock time for when the schedule should begin. Using this "time zero" and the schedule, the start and end times for each timeslice can be calculated. Once a server receives a schedule, it waits until the zero time and then uses its own clock to follow the schedule. If a server receives a schedule after the indicated time has passed, it joins the schedule in progress. If a workload overruns its timeslice, the server abbreviates the following timeslice to fall back in sync by the end of that second timeslice.

When more requests are queued than can be issued in a timeslice, the servers independently choose the same set to issue. In our procotol each request is labeled with a unique ID by the client. Relative to a specific client, newer requests have a numerically greater ID. Thus, even if requests are received in different orders at different servers, the request IDs can be used to establish a consistent temporal ordering. When choosing which subset of requests to issue in a given timeslice, then, each server can choose the oldest requests in the queue and be in agreement without explicit coordination. This approach also has the desirable effect of avoiding starvation.

Each server independently determines how many requests to issue in a given timeslice based on an exponentially-weighted moving average (EWMA) of the service times in previous timeslices for the associated workload at that server. This approach is not overly sensitive to occasional discrepancies, resulting in closely matching values across the cluster. Should a particular workload not be achieving its expected performance, there are two possible remedies. First, the α parameter for the EWMA, which represents the desired amount of smoothing, could be adjusted for that workload to promote more stable behavior. Alternatively, central coordination could selectively be provided for that workload.

C. Evaluation

We ran experiments to confirm that (1) timeslicing without synchronization results in poor performance and (2) that symmetric operation results in coordination between the servers and the desired performance insulation.

Experiments were run on a cluster of Pentium 4 Xeon 3.0 GHz machines running Linux 2.6.16.11. The machines had 2 GB of RAM, but the storage servers were directed to use only 1 MB of RAM for caching to avoid confounding cache effects with disk performance. Each server used two Seagate Barracuda ST3250824AS 250 GB 7200 RPM SATA drives, one as a boot drive and one as the volume exported by our storage server. The drives were connected through a 3ware 9550SX controller. Both the disks and controller support command queueing. The machines were connected over Gigabit Ethernet using Intel 82546 NICs. Clients use the same hardware and do not perform local caching. The software used was Ursa Minor [1], with the Argon storage server [22].

1) Varying the number of servers: The first experiment shows that, while simple timeslicing provides performance insulation on a single server, striping data across two or more servers requires coordination between the servers to provide acceptable performance. We store two files, each of size 100 GB, contiguously starting at the beginning of the disks. Three closed, read-only, random workloads with no think time are run on three separate client machines. The first and third workloads use the first file and have one outstanding request at a time. The second workload uses the second file and has four outstanding requests at a time. We assign each workload one-third shares of each of the servers and request an R-value of 0.9. We run the workloads for a period of eight minutes and monitor the throughput over the last five minutes. The block size is chosen so that each server must supply 4 KB. This holds the disk activity constant as we increase the number of servers, to emphasize clustering effects.

Figure 1 on page 1 shows the throughput of the first workload as we vary the number of servers under three scenarios.[12] The performance insulation goal is that the workload will receive a normalized throughput of $0.9 \cdot \frac{1}{3}$. Without timeslicing, the workload receives significantly less, regardless of the number of servers (in this case because Workload 2 — not shown — crowds it out with its higher degree of concurrency). Timeslicing solves the interference problem for the single-server case. But, with data striped over $\geq$ 2 servers, timeslicing results in worse performance due to lack of coordination. Synchronizing timeslices achieves the goal for each cluster size tested.

The other workloads are not shown in the figure for

[12] Standalone performance ranged from 305 KB/s for one server — within 7% of the datasheet performance for 4 KB random reads on this drive — to 2.2 MB/s for nine servers.

readability, but they behave similarly. In the *Timeslicing* case, they also miss their goal performance by a wide margin. In the *Synchronized* case, each achieves its goal.

2) Macrobenchmarks: The next experiment confirms that our approach works for more realistic workloads. We run Postmark, TPC-C, and a specific query from TPC-H. We ask the cluster to maintain an R-value of 0.85.

Postmark [11] is a benchmark designed to measure performance for small file workloads, such as on an email or newsgroup server. It measures the number of transactions per second that the system is capable of supporting. A transaction is either a file create or file delete, paired with either a read or an append.

The TPC-C workload mimics an on-line database performing transaction processing [20]. Transactions invoke 8 KB read-modify-write operations to a small number of records in a 5 GB database. The performance of this workload is reported in transactions per minute (tpm).

TPC-H is a decision-support benchmark [21]. It consists of 22 different queries and two batch update statements. Each query processes a large portion of the data in streaming fashion in a 1 GB database. Performance is measured by the elapsed time a query takes to complete. We run query 3 in this experiment (its runtime was closest to the other two benchmarks').

The workloads run on a cluster of six servers. Postmark is striped across five servers and is allocated $\frac{1}{3}$ of time on each. TPC-C and TPC-H are each striped across three servers, and each is allocated $\frac{2}{3}$ of time on its respective set of servers. Table I shows the performance each receives, normalized to its standalone performance. Again, uncoordinated timeslicing is inadequate because workloads must wait for the slowest server to respond. Creating a schedule for the cluster and coordinating the servers provides the desired performance for each workload.

V. Related work

This section discusses related work in three areas: quality of service for storage systems, gang- and co-scheduling for high-performance computing, and spindle synchronization in disk arrays. Discussed earlier were the Argon paper [22] that led to this work and the theoretical results on strip packing described in Section III-C.

Most storage quality-of-service (QoS) research [3], [9], [10], [16], [24] has focused on using control theory to provide performance guarantees to workloads without concern for efficiency. By avoiding the timeslicing performed by Argon, they may cluster more easily. (If control systems exist on individual servers, it would remain important that their decisions agree.) Workloads that benefit from locality or streaming disk bandwidth, however, can experience interference that causes an order of magnitude lower efficiency in these systems, making the ability to compose Argon-style guarantees desirable.

Timeslicing is employed to share CPUs among workloads in most common operating systems. *Co-scheduling* [19] and *gang scheduling* [6] enforce synchronized CPU timeslices in HPC clusters in the same spirit as our synchronization of disk timeslices. Despite the similarities, the approaches used to create process schedules for an HPC cluster are not directly applicable to storage systems. First, the strategies employed in the HPC setting may be constrained by communication topologies that do not apply to storage systems. Second, the cost of "context switches" can be significantly greater in storage systems than for CPUs; processor sharing techniques need not take as much care to minimize the occurrence of context switches. Thus, polynomial-time algorithms can be used for gang scheduling [5].

A similar coordination problem to ours is that of spindle synchronization in disk arrays [13], [18]. Without explicit synchronization among disks that store units of the same striped block, the rotational speeds and phases of the disks may differ. This can result in rotational latencies approaching the worst case (one whole rotation) instead of the average case as the number of disks increases.

Table I
Macrobenchmarks

Benchmark	Shared goal	Timeslicing	Synchronized
Postmark	$0.85 \times 1/3 = 0.28$	< 0.05	0.29
TPC-C	$0.85 \times 2/3 = 0.57$	0.21	0.58
TPC-H	$0.85 \times 2/3 = 0.57$	0.50	0.62

Timeslicing is unable to achieve the desired efficiency for three macrobenchmarks sharing a cluster. Only with a synchronized timeslice schedule do the workloads perform as expected. Performance is normalized against what each workload receives when it has exclusive use of its machines; bigger is better. For TPC-H, the metric is inverse runtime; for the others, transactions per second.

VI. Conclusion

Performance insulation can be realized in cluster-based storage by co-scheduling timeslices for each striped workload. Parallel execution of several heuristics enables quick discovery of global schedules in most cases. Explicit time synchronization and implicit work coordination enable the system to provide 2–3X higher throughput than without performance insulation.

Acknowledgment

We thank our colleagues Christos Faloutsos, Raja Sambasivan, Spencer Whitman, and Elie Krevat for discussions and input. We thank the members and companies of the PDL Consortium (including APC, Cisco, DataDomain, Facebook, EMC, Google, HP, Hitachi, Intel, LSI, Microsoft, NetApp, Oracle, Symantec, VMware) for their interest, insights, feedback, and support. We also thank Intel, IBM, Network Appliances, Seagate, and Sun for hardware donations that

enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0326453 and #CCF-0621499, by the Department of Energy, under award number DE-FC02-06ER25767, by the Army Research Office, under agreement number DAAD19–02–1–0389, and by an NDSEG Fellowship sponsored by the Department of Defense.

REFERENCES

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 59–72. USENIX Association, 2005.

[2] B. S. Baker, J. E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, **9**(4):846–55, November 1980.

[3] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. *Symposium on Reliable Distributed Systems* (Florence, Italy, 06–08 October 2003), pages 109–118. IEEE, 2003.

[4] W. F. de la Vega and V. Zissimopoulos. An approximation scheme for strip packing of rectangles with bounded dimensions. *Discrete Applied Mathematics*, **82**:93–101, 1998.

[5] D. Feitelson. *Job scheduling in multiprogrammed parallel systems*. IBM Research Report RC 19790 (87657), October 1994, Second Revision, August 1997.

[6] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, **16**:306–18, 1992.

[7] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, **9**:849–59, 1961.

[8] E. Hopper and B. C. H. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. *European Journal of Operational Research*, **128**:34–57, 2001.

[9] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, 12–16 June 2004), pages 37–48. ACM Press, 2004.

[10] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. *International Workshop on Quality of Service* (Montreal, Canada, 07–09 June 2004), pages 67–74. IEEE, 2004.

[11] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[12] C. Kenyon and E. Remila. Approximate strip packing. *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 31–6, October 1996.

[13] M. Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, **C–35**(11):978–988, November 1986.

[14] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. *New exhaustive, heuristic, and interactive approaches to 2D rectangular strip packing*. Technical report TR2003-05. Mitsubishi Electric Research Laboratories.

[15] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. *New heuristic and interactive approaches to 2D rectangular strip packing*. Technical report TR2005-113. Mitsubishi Electric Research Laboratories.

[16] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: virtual storage devices with performance guarantees. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 131–144. USENIX Association, 2003.

[17] D. L. Mills. *Network time protocol (version 3)*, RFC–1305. IETF, March 1992.

[18] S. Ng. Some design issues of disk arrays. *IEEE Spring COMPCON*, pages 137–142. IEEE, 1989.

[19] J. K. Ousterhout. Scheduling techniques for concurrent systems. *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, October 1982.

[20] Transaction Processing Performance Council. TPC Benchmark C, December 2002. http://www.tpc.org/tpcc/.

[21] Transaction Processing Performance Council. TPC Benchmark H, December 2002. http://www.tpc.org/tpch/.

[22] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. *Conference on File and Storage Technologies* (San Jose, CA, 13–16 February 2007), 2007.

[23] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. *Conference on File and Storage Technologies* (San Jose, CA, 26–29 February 2008), 2008.

[24] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: Storage Performance as a Managed Resource. *RTAS – IEEE Real-Time and Embedded Technology and Applications Symposium* (San Jose, CA, 04–07 April 2006), pages 125–134, 2006.