PARALLEL DATA LABORATORY
CARNEGIE MELLON UNIVERSITY

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

Matthew Wachs, Gregory R. Ganger

CMU-PDL-08-113
October 2008

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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.

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, Cisco, DataDomain, Facebook, EMC, Google, HP, Hitachi, IBM, Intel, LSI, Microsoft, NetApp, Oracle, Seagate, 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.

Keywords: performance isolation, quality of service, shared storage, performance, clustering, approximation algorithms, heuristics, strip packing

1 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.

Our previous work 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 [20]. 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 timeslice of disk head time that bounds the impact of interference. Disk head timeslicing is akin to process scheduling on CPUs.

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 all disk head timeslices do not overlap, the throughput of a closed-loop one-request-at-a-time workload will be one request per round of timeslices.

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. If all data is striped across all servers, the system needs to use an identical ordering of disk head timeslices across servers, handle striped requests in the same timeslices, and synchronize the timing of timeslice switches. 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, 21], 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. 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 yield 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 co-scheduling efficiency reduction helps. In such cases, the system compensates by increasing the per-server R-value used in automatically configuring per-server resource allocations.

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 they expect. Efficient access during a controllable share of server time provides performance that is predictable and controllable.

2 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, without administrator intervention.

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. The cache at a storage server can also be poorly shared across workloads, with one workload receiving more than it needs and flushing useful data of another.

2.1 Performance insulation

This report builds on our earlier work to provide *performance insulation* between shared workloads [20]. Insulation preserves the efficiency that individual workloads receive when they share a storage system. In our earlier work, we described a storage server, called Argon, that takes a configuration parameter termed the *R-value*. It 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.

Argon provides this guarantee via three techniques: round-robin timeslicing, amortization, and cache partitioning. Timeslicing provides each workload long stretches of uninterrupted disk time similarly to standard CPU multiplexing. Each active workload in the system receives a dedicated timeslice, and timeslices are allocated and scheduled in a round-robin fashion. Timeslice lengths are long (e.g., 140 ms) to allow workloads to maintain most of their spatial and temporal locality. (The actual timeslice length can be automatically derived from the desired R-value and measured disk characteristics.) Amortization ensures that streaming workloads effectively use timeslices, via aggressive prefetching and write coalescing within the storage server. Cache partitioning analyzes the access patterns of each workload and assigns it dedicated cache space sized to ensure its specified R-value of efficiency.

2.2 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. Many also store redundancy data (e.g., parity) for each stripe on different servers than the corresponding data. 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*? For some of Argon's techniques, such as cache partitioning, we expect this composition to be relatively straightfoward. Round-robin timeslicing, however, presents significant challenges when performed across servers.

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. For closed workloads, if one of the servers takes longer to respond, this directly reduces 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 carefully arranging the schedules for each of the servers that a workload uses, it may not be possible to consistently "line up" its timeslices 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 (e.g., see Figure 1). 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 1, 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.

To realize the insulation benefits of timeslicing in a cluster, it is necessary to find a cluster-wide schedule that co-schedules timeslices for each workload across all servers it uses. In addition, the servers must adhere to that schedule in a synchronized fashion — that is, timeslices must begin and end at the same time on the servers. Sections 3 and 4 describe and evaluate our solutions to each.

3 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.

3.1 Problem specification

We assume a cluster of homogenous storage servers with one disk each. Machines that have multiple, independent disks can be thought of as separate machines in the context of this problem. Machines that have significantly different resources (i.e. disrupt homogeneity) should be placed in a separate cluster with like machines. If a set of machines has minor performance variations, however, they may be clustered together and be treated as a number of machines with the "lowest common denominator" of performance.

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. To avoid the cost of switching between workloads more than necessary, the share of time is intended to be contiguous.

The problem of finding a suitable schedule for the cluster, then, amounts to finding round-robin timeslice orderings for each of the servers such that each workload receives its share of total 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.[1]

Figure 2 shows an example input list of workloads and an example solution to the problem.

[1]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.

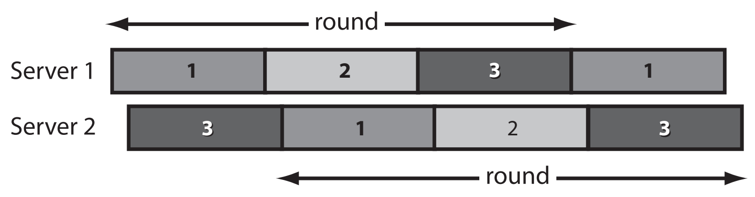

Figure 1: **Out-of-phase servers.** If the phase of identical rounds does not match across the servers, then a client must wait for the most-in-the-future occurrence of its timeslice across all the servers before its request completes.

3.2 Geometric interpretation

This problem may be recast as a geometric problem, as suggested by Figure 2. 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. If not, then no appropriate timeslice schedule exists.

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, such as wood or glass, must be cut into smaller pieces to manufacture a product. Unfortunately, it is known to be NP-complete (for instance, it is a more general version of the bin-packing problem). 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?*[2] An *optimal* solution of a given problem (in our choice of dimensions) is a packing of rectangles that uses the minimum 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.

3.3 Related work

Strip packing has been explored by theoreticians searching for ways to accelerate exhaustive searches for the optimal solution, for approximation algorithms that can provide solutions within a guaranteed distance from the optimal solution, and for heuristics that may find “acceptable” solutions quickly.[3]

Exhaustive search As with any NP-complete problem, one can enumerate all possible solutions and test whether they meet the requirements or whether they are the best solution seen up to that point (yielding the best possible solution at the end of the exhaustive search). However, this approach results in an exponential run time that will be prohibitive for sufficiently large problems.

[2]The problem is usually specified for a fixed width and variable height, but we reverse the dimensions because it works more naturally for our problem.

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

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

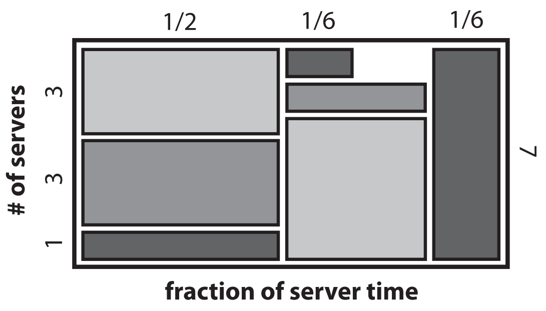

Figure 2: **Example problem instance and solution.** On the left is an example input to the scheduling algorithm; on the right is one possible solution. Rectangles correspond to workloads, with their height corresponding to the number of servers and their vertical location corresponding to which servers to use; their width corresponds to share of time and their horizontal location corresponds to what span of time during which their timeslices are scheduled. The enclosing rectangle represents a single round in the cluster; the schedule is repeated indefinitely.

In the case of strip packing, a naïve search might consider placing a rectangle at each possible coordinate location.[4] However, Fernandez de la Vega and Zissimopoulos [4] show that it suffices to consider a smaller set of possible solutions. If there is a solution, then there exists a *left-bottom justified* solution. Such a solution is generated by placing rectangles one after the other in some order without leaving gaps between rectangles; the authors show there is no strategic benefit to having gaps. Thus, it suffices to examine all possible orderings of the rectangles, and for each of those orderings, only the placements of successive rectangles that are at *active corners*, i.e. touching other rectangles or the boundaries. This strategy still results in an exponential search space, however.

Another technique for reducing the size of the search space further is the branch-and-bound method. Martello *et al.* [16] used branch-and-bound to solve or approximate solutions to a set of benchmark problems with around two dozen rectangles on the order of minutes to an hour on an 800 MHz Pentium III.

Approximation algorithms Approximation algorithms exist [4, 11] that are able to find solutions within $(1+\varepsilon)$ of optimal in time that is polynomial in the number of rectangles. They work by subdividing the large rectangle into smaller rectangles, packing subsets of the rectangles into each of these smaller problem instances, and then "gluing" the smaller instances back into a larger instance (see Figure 3).

There are two issues that make even the approximation forms of these theoretical algorithms potentially impractical for our use. First, while the algorithms are polynomial in the number of rectangles, the number of possible ways to fit rectangles into a subproblem appears as a constant in the running time analysis. This value is exponential in the size of the subproblems and the number of distinct sizes of rectangles. With small subproblems, running time will be fast but distance from optimality may be high because too many subproblems are being "glued together"; loss can be introduced at each "joint." With large subproblems, loss will be reduced, but at the expense of a dramatic increase in time required to consider subproblems. Second, loss cannot necessarily be made arbitrarily small; it is limited by the size of the problem instance. In other words, when finding a solution within $(1+\varepsilon)$ of optimal, the best ε achievable is a function of the problem size. For the size of the problems we expect to encounter, ε is expected to be too high.

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

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

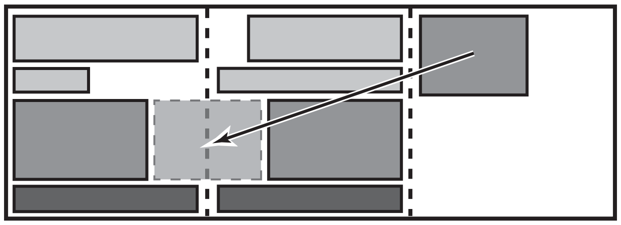

Figure 3: **Approximation algorithms.** These algorithms divide the larger problem into smaller subproblems that are solved optimally, but there can be significant loss in "gluing" them together into a larger solution. Here, the indicated rectangle wasn't placed in the more appropriate location, because it spanned two subproblems that were considered independently.

there is none. However, they can often be very effective in practice. Furthermore, despite not providing guaranteed bounds, they may do significantly better than the $(1+\varepsilon)$ approximation bound for our problem sizes. In our case we only need to find a solution with a width of one or less. If a heuristic method finds such a solution immediately, how close it is to optimal 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.* [13] suggest sorting by decreasing area (BLF-DA) or perimeter (BLF-DP).

Lesh *et al.* [13, 14] further suggest if one of these heuristic methods does not work on the first attempt, 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*.

Finally, Hopper and Turton [8] discuss combining these heuristics with meta-heuristic techniques such as simulated annealing, genetic algorithms, or hill-climbing. These techniques can start with an initial placement made with, for instance, BLF-DW, and refine the solution over a sequence of modifications in an attempt to approach the optimum.

3.4 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}. If clustering does not introduce any further loss in efficiency and each server is operating at at least R_{server}, then the overall efficiency seen by the client is still R_{server}. However, 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, called $R_{clustering}$, which represents the minimum efficiency maintained by the clustering scheme. The R-value observed at the client, then, is at least:

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

The R-value at the client is the ultimate indication of whether insulation has been achieved; $R_{clustering}$ and R_{server} are not externally visible and can be manipulated for the convenience of the storage system.

3.5 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 heuristic methods (BLF, BLF-DW, BLF-DH, BLF-DA). If the heuristic orderings do not lead to a solution, then we try two modifications. First we keep the same sort order, but instead of placing rectangles as far to the bottom and left as possible, we try all other combinations of locations at which to place them, subject to not introducing gaps. If none of these combinations are successful, we then try permutations to the heuristic sort orders for a limited period of time. 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$\}$. 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. If the minimum acceptable R-value is known up-front, we can alternatively start with a value of $R_{clustering} < 1.0$ rather than initially considering the best possible value.

Initial setup The inputs to our algorithm are the number of homogenous 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 one small rectangle corresponding to each of the individual workloads, with the height equal to the number of servers needed by that workload and the width equal to the proportion of the servers' time it needs (e.g., 1/3). 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 round length (the period over which the schedule repeats), and each workload's rectangle consumes a fraction of the round length. If $R_{clustering} = 1.0$, representing no loss of efficiency due to clustering, 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 lower proportional share of the overall round, resulting in a fractional decrease in performance equal to $R_{clustering}$.

3.6 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.

3.6.1 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. 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.

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

place_next_workload(sched, still_to_place)
 if still_to_place[i] = 0 for all i then
   exit with "Schedule found", sched
 for i = 1 to number of distinct types of workloads do
   if still_to_place[i] > 0 then
     /* Build up 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
           sched_copy = sched
           still_to_place_copy = still_to_place
           Place a wl. of type i at loc. (s, t) in sched_copy
           decrement still_to_place_copy[i]
           recursively call place_next_workload(sched_copy,
             still_to_place_copy)
```

Figure 4: **Search ordering.** If the heuristic placement methods do not immediately find a solution, nearby solutions are searched in the order specified by this algorithm. The particular heuristic method 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 the workload type with the greatest width.

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 for the list is then calculated as follows. The product of the number of servers and fraction of time is computed for each of the workloads, and these products are summed to find the total server-fraction product for the set of workloads. This value is similar to a quantity like "server-hours." We then round this number up and assign that many servers to the cluster. Geometrically, this is the same as summing the areas of the individual rectangles, then creating a larger rectangle with approximately the same total area (with a width of one and a height equal to the rounded-up desired area). There is no guarantee this problem instance is solvable, but it represents the cluster size that would have the least wasted resources for that set of workloads. There is one exception to this process: if the number of servers computed is less than the number that one of the workloads needs, the number of servers is adjusted 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 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 dual-Pentium 4 Xeon 3.0 GHz processors running Linux 2.6.24. Our code was single-threaded, so only one processor was used at a time. The memory footprint of strip packing is small, so memory and storage were not bottleneck resources.

3.6.2 Results

Exhaustive method We use the exhaustive method to solve these problem instances. The amount of time taken to find a solution or determine that one does not exist is shown in Figure 5 and in log scale in Figure 6. Beyond eleven workloads, exhaustive search was impractical. However, as examples, one twelve-workload instance took 400 minutes and one thirteen-workload instance took 160 hours (just under a week).

Individual heuristics Figures 5 and 6 also show the decreasing width (DW), decreasing height (DH), and decreasing area (DA) heuristics applied 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 3.5. No single heuristic was sufficient to improve the mean time to solution.

Our approaches Figure 7 shows solution speed with our approaches — parallel execution of the three heuristics and the original ordering, continuing to search the nearby solution space if a solution is not immediately found. The two lines represent $R_{clustering} = 1.0$ and $R_{clustering} = 0.9$. 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 8 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; however, it may not be worthwhile to use the unsorted order for larger problem sizes, because its utility appears to decline sharply.

Figure 9 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 thoretically solvable (i.e., there exists a solution at all). To maintain the desired overall R-value, however, this approach would then incur the cost of increasing R_{server} to compensate.

Solvability and timeouts 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 problem instances 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 95th percentile of these 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. Table 1 shows the values used.

Because many of these problem instances are too large for us to exhaustively solve, we cannot determine for certain that solutions do not exist beyond those we found. However, our experience with smaller problem 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 of solutions. For instance, no additional solutions were found by running to exhaustion 59 of the eleven-workload problem instances that did not find a solution before we halted the original experiment; the percentile used to determine the timeout directly determines what percent of problems will be solved in this case. Furthermore, the trend in the number of solutions found for the smaller problems, extended to larger problem sizes, suggests the appropriate number of solutions are being found for the larger problems by the heuristics. For instance, between 8% and 11.5% of the randomly-generated problems for sizes 7, 8, 9, and 10 were solvable for $R_{clustering} = 1.0$ using exhaustive search, and the fraction of instances solved by the time-limited search for problems of size 11, 12, and 13 fell into the same range.

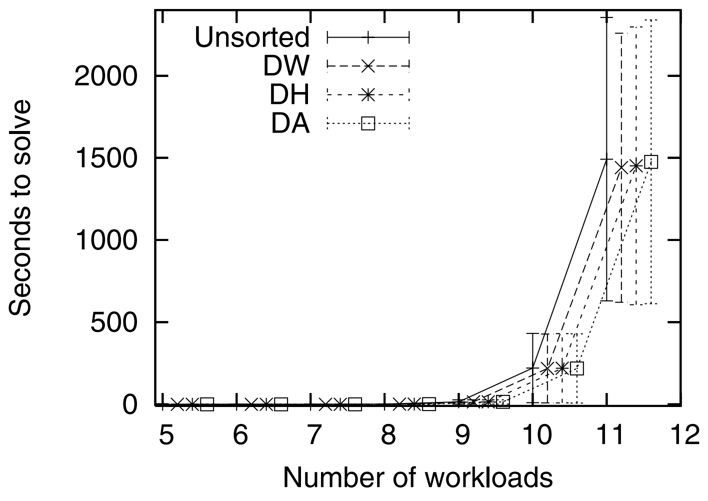

Figure 5: **Exhaustive search and heuristics.** Exhaustive search of the solution space takes impractically long when there are more than a few workloads. Initializing the search with one of the three heuristics (Decreasing Width, Decreasing Height, or Decreasing Area) does not remedy this problem; growth is exponential throughout the range plotted. In this graph, three of the lines are slightly offset in the horizontal direction to make them distinguishable. Error bars show one standard deviation in either direction.

Parallelization in the cluster Finally, if idle CPUs are available in the storage cluster for which the schedule is being computed, then they can be exploited for further speedup. 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. Figure 10 shows the run time for $R_{clustering} = 0.9$ if one CPU per server in the cluster is used to parallelize the search. (The number of servers in the cluster varies from problem instance to problem instance.)

4 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 solutions adopted.

Num. workloads	Timeout (sec.)
9	2
10	9
11	16
12	21
13	21
14	64
20	100
30	316
40	497

Table 1: **Timeouts.** Timeouts used for $R_{clustering} = 0.9$, calculated as described in Section 3.6.2. Only problem sizes large enough to have a timeout greater than one second are shown.

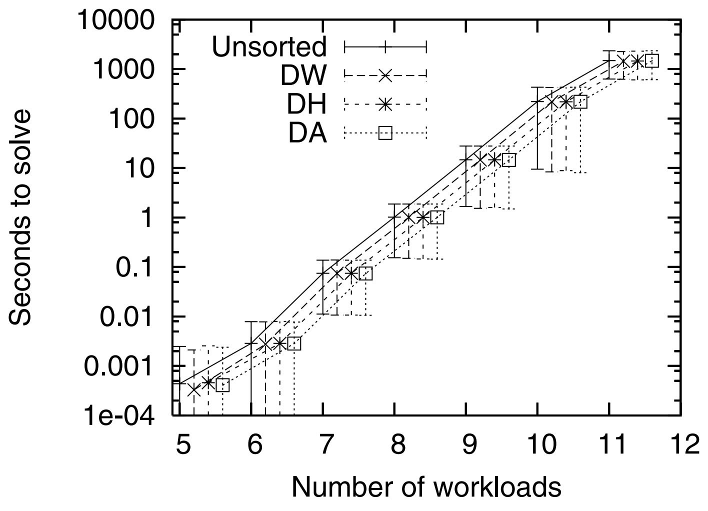

Figure 6: **Exhaustive search and heuristics.** Exhaustive search of the solution space takes impractically long. This graph shows the same values as the previous one, but with the *y*-axis in log scale. Three of the lines are offset horizontally for readability.

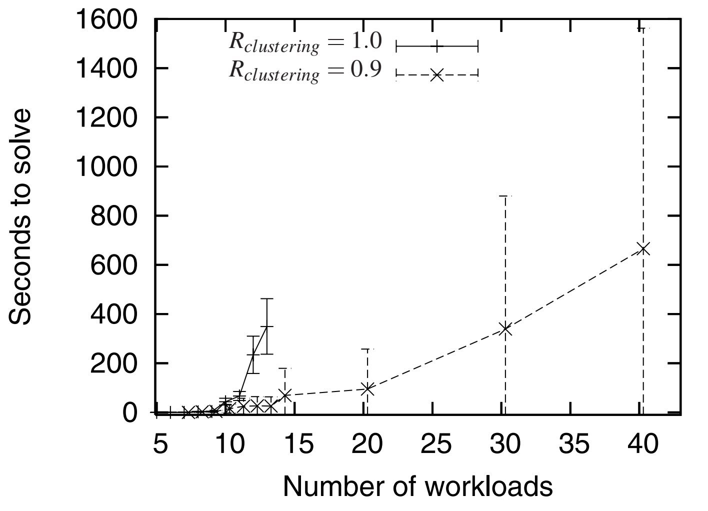

Figure 7: **Parallel heuristic search.** Running searches initialized with different heuristics in parallel allows the best-performing heuristic for a particular problem to find a solution faster. For the $R_{clustering} = 1.0$ case, however, an adequate sample size could not be achieved for problems of size 14 and greater to be plotted due to growing run time. Relaxing $R_{clustering}$ further accelerates the search and makes larger problems tractable.

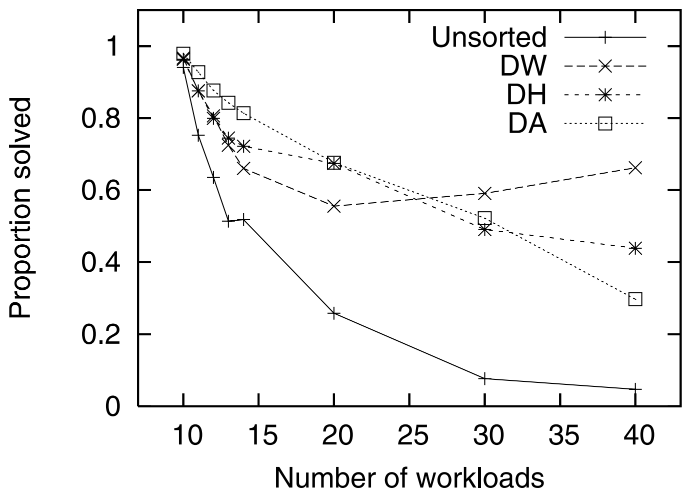

Figure 8: **Proportion of problems solved per heuristic.** In a limited amount of time (one minute for 10–13 workloads, two minutes for 14 workloads, five minutes for 20 workloads, ten minutes for 30 workloads, and thirty minutes for 40 workloads), initializing a search with one heuristic is not able 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.

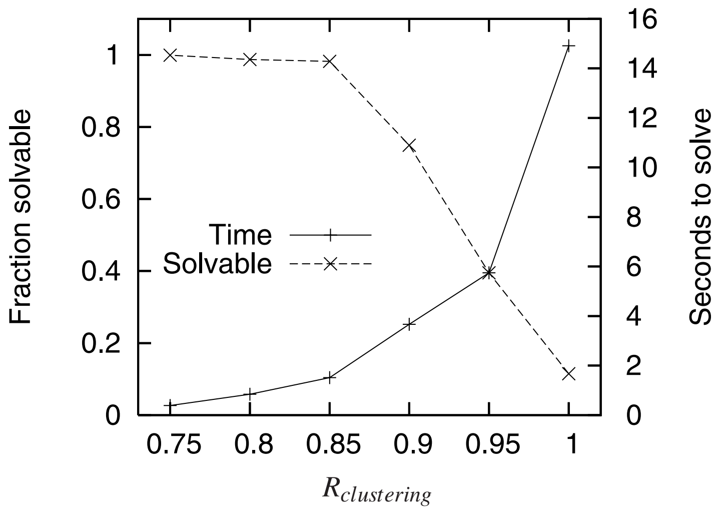

Figure 9: **R-value tradeoffs for 13 workloads.** Relaxing $R_{clustering}$ can accelerate solution speed and increase the number of solvable instances.

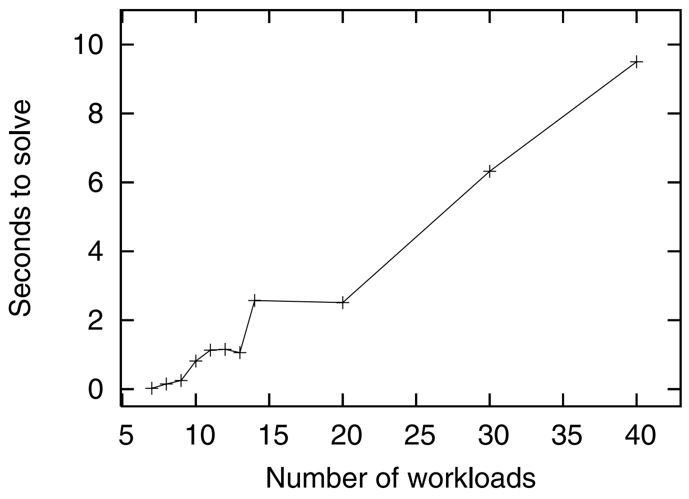

Figure 10: **Search parallelized across cluster.** Running the same search using one CPU of each machine in the cluster results in further speedups. (Total CPU time for these problem instances summed across the cluster would be the same values depicted in Figure 7.)

4.1 Requirements

The servers must begin following the schedule of timeslices at approximately the same moment of wall-clock time. (Because timeslices are long, e.g. 140 ms, small offsets among the servers — say, 0.5 ms — will not be significant.) Once begun, the servers must start each subsequent timeslice in the schedule at the same approximate time. Any offset or error must not compound over time unless it accumulates identically across the servers. For instance, if a timeslice begins late at one server for some reason, the next timeslice must either begin on time, or (if it is desired to give the late-starting workload a full timeslice) 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.

The number of requests per timeslice Our storage server runs at user-level and is not able to cancel a disk request that has been sent to the kernel. This complicates the implementation of timeslice-based scheduling. 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 at our level in the system. If we send more requests than would fill the timeslice, we would delay the beginning of the next timeslice and penalize the subsequent workload. If we send fewer requests than could fit, we would have unnecessary idle time at the end of the timeslice. 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. To attempt to issue exactly the right number of requests, we estimate how many 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.

Note that the servers may not compute the same number of requests to issue for a given workload. Variations in disk service times may make the historical observations that drive this decision differ across the servers. If one server issues more requests in a timeslice than the others, this does not improve client performance because the client will have to wait for the other servers to complete the extras. If one server issues fewer requests than the others, on the other hand, it impedes client performance. Thus, the servers should issue approximately the same number of requests for a given co-scheduled timeslice.

4.2 Initial solution: central coordination

One approach to keeping all the servers in sync is to centrally coordinate their actions. We implemented a central coordinator and added commands to our storage protocol to communicate between the coordinator and the servers. The coordinator is the "timekeeper" that monitors wall-clock time and determines when timeslices should begin and end. It sends `begin_timeslice` messages to servers at the appropriate times. The payload indicates the workload to execute next, the length of the next timeslice in milliseconds, and the number of requests the server should allow to be issued. A server that finishes a timeslice late, because a request was still in flight when the next `begin_timeslice` message arrives, begins the next timeslice as soon as possible and abbreviates the new timeslice (by issuing fewer requests) to return to sync.

At the end of a timeslice, each server sends a `report_requestsissued` message back to the coordinator. The payload indicates how many requests were actually issued to the disk, the actual length of time that the timeslice lasted (in case it started late), and the amount of idle time during the timeslice (in case the client did not keep the server busy the entire time). The central coordinator is able to calculate, for each of the servers, a per-request service time based on these three values and arrive at a common suggested number of requests to issue for that workload in the next round.

4.3 Final solution: symmetric operation

Although it worked, we prefer not to need a central coordinator. One 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 redesigned our servers to avoid the need for 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 requests 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 of requests to issue. In the protocol used by our storage system, each request is labelled 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 of requests. 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.

Our system chooses to issue the same, or almost the same, number of requests in co-scheduled timeslices across the set of servers as long as the disks are performing similarly. 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 be selectively provided for that workload.

4.4 Evaluation

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

Experiments were run on a cluster of dual-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, which exposes the disks to the OS through a SCSI interface. Both the disks and the controller support command queueing. The machines were connected over Gigabit Ethernet using Intel 82546 NICs. The same hardware was used for clients as for servers; the clients do not perform local caching. The software used was Ursa Minor [1], with the Argon storage server [20].

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 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 a fragment of size 4 KB per request. This holds the disk activity constant as we increase the number of servers, to emphasize clustering effects rather than disk factors.

Figure 11 shows the throughput of the first workload as we vary the number of servers under three scenarios. The *y*-axis indicates throughput normalized to the performance the first workload receives when it runs alone on the corresponding number of servers. The performance insulation goal, then, is that the workload will receive a normalized throughput of $0.9 \cdot \frac{1}{3}$. Without performance insulation, 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 two or more servers, timeslicing results in worse performance than with no insulation at all because of the lack of coordination. Many requests must wait nearly an entire round to complete, because they are held up by the server with the farthest-in-the-future timeslice. Synchronizing timeslices successfully achieves the R-value goal for each cluster size tested.

The other workloads are not shown in the figure for 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.

Varying the number of workloads We also examine the case where the number of servers is fixed at three, but the number of workloads is varied. For these experiments, each workload has a 32 GB file and uses 12 KB block sizes — each server stores a 4 KB fragment for a given block. The workloads each run on their own client machine, are closed with one outstanding request each, and are read-only. Figure 12 shows the throughput in requests per second during the first workload's share of time, as the number of workloads

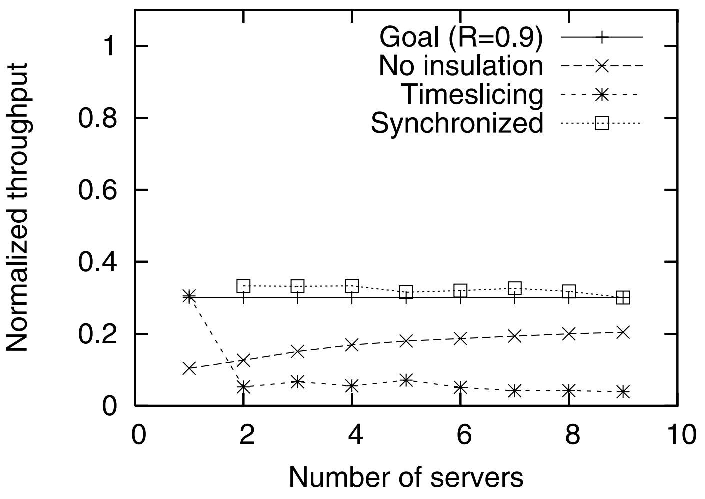

Figure 11: **Increasing the number of servers.** As soon as the number of servers across which fragments are striped is greater than one, timeslicing becomes ineffective unless the timeslices are synchronized across the servers. The *y*-axis shows throughput (while sharing with two other workloads) normalized against the throughput the depicted workload receives running alone on the corresponding number of servers.

is varied. (Its overall throughput decreases proportionally with the increase in the number of workloads, because it receives fewer timeslices per unit of time.) The other workloads are identical to the first and receive virtually the same level of performance for each scenario. With timeslicing but no synchronization of timeslices, the goal is missed by a wide margin. Adding synchronization reaches the goal.

Putting it all together The final experiment confirms that the entire process — starting with a list of workloads and a cluster; finding a schedule and disseminating it to the servers; and synchronizing their execution — results in the expected performance insulation. We use one of the random problem instances of size five and the schedule that was found when we solved it during the generation of Figure 7. We augment the problem instance with choices for file size and number of outstanding requests and ask the cluster to provide an R-value of 0.9. Expressed as tuples of (*num servers*, *share of time*, *file size in GB*, *outstanding*), the workloads are {(7, 1/6, 28, 2), (5, 1/3, 120, 3), (3, 1/2, 96, 1), (5, 1/2, 120, 4), (5, 1/2, 120, 5)}. We choose to make the workloads closed, uniformly random, and read-only with 4 KB fragments. We use a cluster of size ten, the minimum for this set of workloads. Each server runs with 1 GB of cache. The first workload benefits from a significant cache hit rate because it stores only 4 GB of data per server. If it receives insulated performance, this confirms the guarantees from cache partitioning compose as expected in the cluster as well. Figure 13 shows that the expected R-values are provided.

5 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 [20] that led to this work and the theoretical results on strip packing described in Section 3.3.

Most storage quality-of-service (QoS) research [3, 9, 10, 15, 22] has focused on using control theory to provide performance guarantees to workloads without concern for server efficiency. By avoiding the timeslicing performed by Argon, they may compose per-server guarantees into cluster-wide ones more nat-

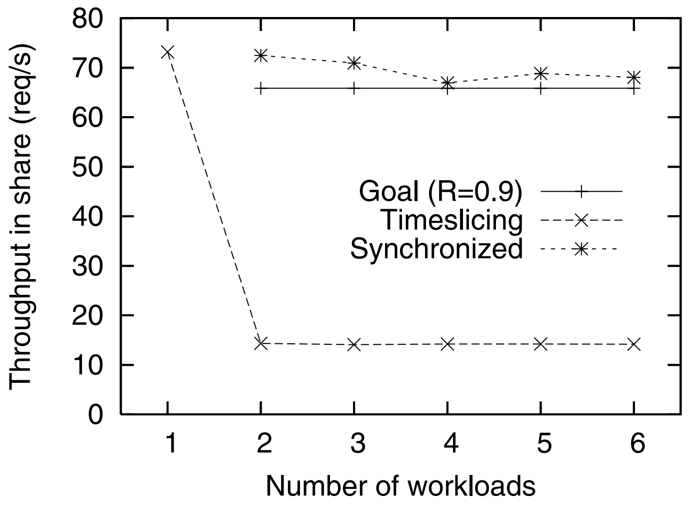

Figure 12: **Increasing the number of workloads.** Unsynchronized timeslicing results in unacceptable performance even when only two workloads share a server. Synchronizing the timeslices ensures insulated performance in a workload's share of time.

urally. (If control systems exist on individual servers, it would remain important that their decisions coincide across the servers.) 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. When pooling numerous CPUs to create high-performance computing (HPC) clusters, the same concerns about composability of performance across individual nodes are raised. *Co-scheduling* [19] and *gang scheduling* [6] enforce synchronized CPU timeslices in the same spirit as our synchronization of disk timeslices. Feitelson [5] provides a survey of the various scheduling issues in the HPC environment. 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 often are 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 care to minimize the occurrence of context switches or provide workloads with contiguous spans of time. On a theoretical level, "zero-cost preemption" of a resource is compatible with linear programming formulations (which assume arbitrary divisibility of the modelled variables) whereas "non-preemption" of a resource is only compatible with integer programming formulations (which do not) [5]. Thus, the lower cost of preemption in processor scheduling is more likely to be conducive to polynomial-time approaches, and indeed polynomial-time algorithms are used for gang scheduling [5]. Finally, we note that, if a clustered storage system is being used to provide the file system for an HPC cluster, coordination between the two scheduling regimes may be needed.

A similar coordination problem to ours is that of spindle synchronization in disk arrays [12, 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. Some disks have physical inputs or use command-set extensions to match speed and phase in an array.

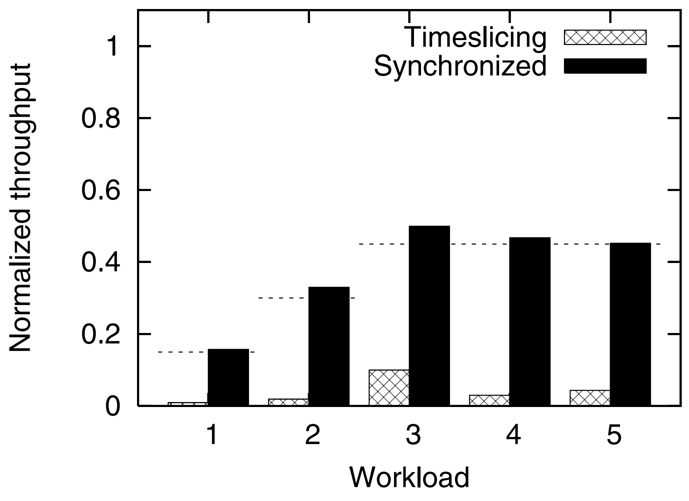

Figure 13: **Putting it all together.** Five workloads with different numbers of servers and shares of server time run on a cluster. The techniques described in Section 3 are used to generate a schedule of timeslices across the ten servers. Timeslices are coordinated across the cluster as described in Section 4. The *y*-axis shows throughput normalized against the performance each workload receives running alone on its set of servers. The lines show the $R = 0.9$ throughput for the share of server time each workload was assigned in the problem instance.

6 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.

Acknowledgements

We thank our colleagues Raja Sambasivan, Spencer Whitman, and Elie Krevat for discussions and input.

References

[1] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay 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] Brenda S. Baker, Jr. E. G. Coffman, and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, **9**(4):846–55, November 1980.

[3] David D. Chambliss, Guillermo A. Alvarez, Prashant Pandey, Divyesh Jadav, Jian Xu, Ram Menon, and Tzongyu 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] Wenceslas Fernandez de la Vega and Vassilis Zissimopoulos. An approximation scheme for strip packing of rectangles with bounded dimensions. *Discrete Applied Mathematics*, **82**:93–101, 1998.

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

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

[7] P. C. Gilmore and Ralph 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] Wei Jin, Jeffrey S. Chase, and Jasleen 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] Magnus Karlsson and Christos Karamanolis. Choosing replica placement heuristics for wide-area systems. *International Conference on Distributed Computing Systems* (Tokyo, Japan, 24–26 March 2004), pages 350–359. IEEE, 2004.

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

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

[13] Neal Lesh, Joe Marks, Adam McMahon, and Michael Mitzenmacher. *New exhaustive, heuristic, and interactive approaches to 2D rectangular strip packing*. Technical report TR2003-05. Mitsubishi Electric Research Laboratories.

[14] Neal Lesh, Joe Marks, Adam McMahon, and Michael Mitzenmacher. *New heuristic and interactive approaches to 2D rectangular strip packing*. Technical report TR2005-113. Mitsubishi Electric Research Laboratories.

[15] Christopher R. Lumb, Arif Merchant, and Guillermo 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.

[16] Silvano Martello, Michele Monaci, and Daniele Vigo. An exact approach to the strip-packing problem. *INFORMS Journal on Computing*, **15**(3):310–19, 2003.

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

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

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

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

[21] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. *Conference on File and Storage Technologies* (San Jose, CA, 26–29 February 2008), 2008.

[22] Theodore M. Wong, Richard A. Golding, Caixue Lin, and Ralph 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.