

Parallel Data Layout Optimization of Scientific Data through Access-driven Replication

John Jenkins,^{*†‡} Xiaocheng Zou,^{*} Houjun Tang,^{*} Dries Kimpe,[†] Robert Ross,[†] Nagiza F. Samatova,^{*}

^{*}Department of Computer Science, North Carolina State University
{xzou2, htang4, nfsamato}@ncsu.edu

[†]Mathematics and Computer Science Division, Argonne National Laboratory
{jenkins,kimpe,rross}@mcs.anl.gov

[‡]Corresponding author: jenkins@mcs.anl.gov

Abstract—Efficient I/O on large-scale spatio-temporal scientific data requires scrutiny of both the logical layout of the data (e.g., row-major vs. column-major) and the physical layout (e.g., distribution on parallel filesystems). For increasingly complex datasets, hand optimization is a difficult matter prone to error and not scalable to the increasing heterogeneity of analysis workloads. Given these factors, we present a partial data replication system called RADAR. We capture datatype- and collective-aware I/O access patterns (indicating logical access) via MPI-IO tracing and use a combination of coarse-grained and fine-grained performance modeling to evaluate and select optimized physical data distributions for the task at hand. Compared with existing methods, we store all replica data and metadata, along with the original, untouched data, under a single file container using the object abstraction in parallel filesystems. Our system can produce up to many-fold improvements in commonly used subvolume decomposition access patterns, while the modeling approach is capable of determining whether such optimizations should be undertaken in the first place.

I. INTRODUCTION

In high-performance computing (HPC) systems and parallel filesystems such as PVFS [1], Lustre [2], and GPFS [3], the distribution of data across multiple storage devices is a difficult problem, which numerous works have been dedicated to solving. The combination of high-dimensionality (multiple variables distributed in a spatiotemporal domain) and distributed requests over many processes complicates making an informed decision about how to place data to result in high performance. The problem is exacerbated when noncontiguous access patterns are induced on storage, such as subvolume access. Even optimizations made to reduce or eliminate noncontiguous disk access, such as two-phase collective I/O [4], create new access patterns for which the data distribution may not be optimized.

Previous works have looked at data layout optimization in an HPC context in two general respects: modifying the logical layout of data with the goal of producing specialized data organizations for a specific usage (e.g., range-query processing on scientific data [5], [6], [7]) and optimizing the physical distribution of datasets to better match the mapping of process requests to I/O servers [8], [9], [10], either in place or as separate entities in storage, and with varying degrees of adaptability. However, these works have some combination of the following potential problems that we wish to mitigate: mod-

ified logical formats introduce both interoperability concerns and difficulties related to manual management of the custom format; works that provide multiple data layouts or replicate data in multiple formats rely on creating directories/files for each, leading to a large number of files to process any time the dataset is used; and the distribution formats are either fixed or optimize for a single metric (e.g., disk thrashing via DiskSim [11], requests to a single segment of file).

Given these problems, we present a model-driven, adaptive layout optimization framework, called RADAR, using direct parallel filesystem semantics. Our layout optimization is based on partial replication, allowing a controllable increase in dataset sizes in exchange for I/O performance optimization. Furthermore, as opposed to previous works, which fix either the regions of data to replicate or the replication format, we allow variability in both. In particular, we present the following contributions:

1) *Adaptive, storage-aware replica management policy.* Given a set of I/O access patterns, our replica layout manager (Section II-A) uses an I/O performance modeling approach to (1) create replicas with varied layouts for performance optimization of input access patterns and (2) to rank replicas for inclusion under storage-limited scenarios. Furthermore, our approach can gracefully handle imbalances in both server loads and client loads, using performance modeling to account for the former and distribution heuristics to account for the latter. Using a prototype MPI-IO driver, we show that our method is effective at accelerating common subvolume decomposition tasks, showing multifold speedups under many scenarios.

2) *Single-container, nonintrusive dataset storage.* All replica data and metadata are stored alongside the original, unchanged data in a single file container, achieved through direct object-storage semantics (Section II-B). Distribution of replica data among a fixed set of replica objects is enabled through a combination of sparse-file capabilities and an object-slice-based allocation scheme (Section II-B3).

3) *Datatype- and collective-aware MPI-IO tracing.* As an enabling technology, we develop a tracer capable of collecting full logical I/O requests with low overhead at the MPI-IO-level (Section II-C). It is configurable to collect either precollective or postcollective optimizations (or both).

Our paper is organized as follows. Section II describes the framework, including any necessary background. Section III

presents our experiments. Section IV examines related work. Section V briefly summarizes our conclusions.

II. METHOD

The system workflow, in which access patterns are garnered from applications and replicated data layouts are created to optimize for those access patterns, is realized by a number of components, as shown in Figure 1. First we develop a datatype-aware, collective-aware I/O trace layer that captures I/O requests. Then we process traces using a pattern analyzer, outputting access patterns of interest, such as strided access. Our replica layout manager ingests these patterns and, along with previously generated patterns, determines what data to replicate and in what format. Our replica-aware, object-storage-based ADIO implementation matches I/O requests to replications, redirecting the subsequent EOF object operations.

Since the bulk of our methodology lies in the layout manager and works independent of the method of replica distribution and access pattern generation, we first discuss it in Section II-A. Next, we describe the data management policies employed by our method in Section II-B, followed by the tracing and trace-analysis components in Section II-C. The replica-aware ADIO driver is discussed in Section II-D.

A. RADAR Layout Manager

The goal of RADAR’s layout manager component is to create replicas with a layout that improves I/O performance under a given set of access patterns. Given this goal, we consider three questions: (1) How do we generate a data layout to best improve performance under a particular access pattern? (2) How do we measure performance “improvement” itself? and (3) How do we augment our choices with both temporal information and limits on storage overhead?

To provide a framework capable of answering these questions in a flexible manner, the layout manager uses the following strategies. (1) To optimize in the presence of concurrent accesses, we generate replicas for *time-delimited pattern sets*. (2) To quickly generate and evaluate candidate replica sets, we adopt a *two-phase* performance-modeling approach, using a coarse-grained performance model to quickly produce and select candidate replica sets and using a fine-grained performance model to compare with the original data layout. (3) Given user-driven, space-constrained scenarios, we employ a configurable *aging mechanism* to aid in selecting new replicas for inclusion, alongside the potential performance improvement calculated by the performance models. Algorithm 1 gives an overview of the replica creation and decision process, and the following sections discuss the individual components.

1) *Pattern Preprocessing*: Preprocessing of the patterns is driven by our optimization goals: create a replica enabling efficient access of the pattern, while being aware of concurrent system operations. The latter has been examined in previous work by converting all accesses into log-structured, effectively creating a one-to-one process-to-server mapping [12], [9]. Here we are looking at flexible distribution among multiple servers for read optimization, rather than write optimization.

Algorithm 1: RADAR layout manager overview

```

in-out : DS (data store): RADAR MPI file consisting of pattern
           store PS, replication store RS, replica metadata RM
input  : Pats: set of access patterns to process
input  :  $\Delta t$ : time window size
input  :  $\mathcal{M}_f, \mathcal{M}_c$ : fine, coarse-grained perf. models
input  :  $\gamma$ : decay function
input  :  $\sigma$ : storage upper bound

// split patterns into sets of concurrent accesses
1  $\beta \leftarrow$  partitioning of Pats into buckets by  $\Delta t$ 
2 processed  $\leftarrow \{ \}$ 
// process each pattern subset
3 for  $b \in \beta$  do
// use coarse-grain model to make candidate replica
4   reps  $\leftarrow$  opt_replicas( $\mathcal{M}_f, b$ )
// use fine-grain model to estimate perf. benefit
5    $C_{\text{diff}} \leftarrow \mathcal{M}_f(b, \text{orig\_layout}) - \mathcal{M}_f(b, \text{reps})$ 
6   append { $C_{\text{diff}}, \text{reps}$ } to processed
// read existing (benefit, age, replicas) tuples, applying decay
7 existing  $\leftarrow [ \{ \gamma(c, \text{age}), R \} \mid (c, \text{age}, R) \in \text{PS} ]$ 
8 merge  $\leftarrow$  sort (processed  $\cup$  existing)
9 write replica sets in decreasing order from merge, evicting
  replicas from merge in increasing order as required by  $\sigma$ , until
  no more can be added

```

Each pattern in our analyzed traces has starting and ending times. Given these and a value δ , the patterns are partitioned into *buckets*, each corresponding to a time window of length δ . Each bucket is then considered a single entity for the purposes of performance modeling and optimization. Since each bucket need not be sorted in our method, the overall process is linear in the number of patterns.

2) *Replica Generation and Ranking – Performance Modeling*: We use a simple, constant-time performance model to generate candidate replicas and a more involved model to give a relative performance comparison between the original data layout and candidate layouts. This design decision is made in order to quickly generate replicas for testing, while retaining the ability to evaluate against unbalanced access patterns with respect to either the amount of data requested per-process or the amount of data processed by each I/O server.

a) *Preliminaries*: Our models use latency/bandwidth measurements over both network and storage, assuming serialization of requests at the node level (both client and server) and requests to storage. Table I shows the relevant variables. Furthermore, we make a few simplifying assumptions across both models that, while harmful to general-purpose high-accuracy performance prediction, still allow us to make valid measures for comparative purposes over time-gated accesses in a manner that is computationally reasonable. First, we assume no pipelining of network and storage operations, insulating us from false positives arising from slightly different access schedules but giving a pessimistic view of system capabilities; we consider both types of costs equally from an optimization point of view. Second, resource contention is measured through the aggregation of request latencies and, in the fine-grained performance model, through penalization terms on nodes based on the number of distinct requests. This approach misses some phenomena observed in real runs or in full system/subsystem simulators [11], such as disk head

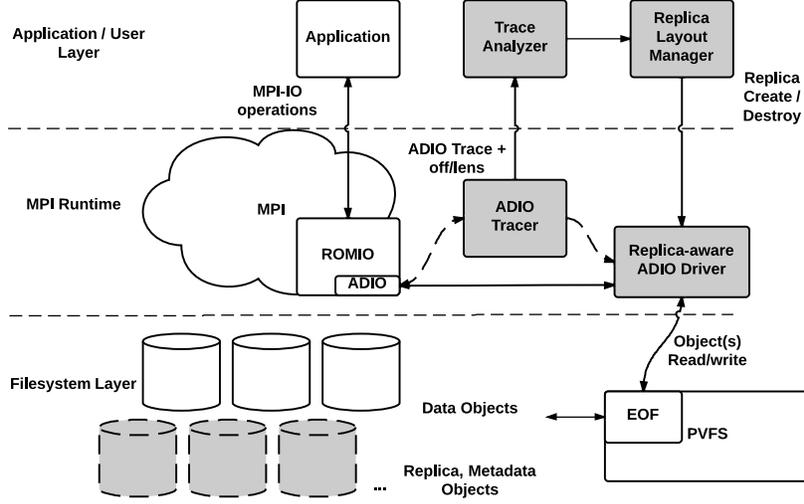


Fig. 1: RADAR components, across the I/O software stack. The shaded figures delineate our contributions.

TABLE I: Performance Model System Parameters

System parameters	
n	I/O servers
ℓ_{net}	I/O request (network) latency
b_{net}	Network per-byte transfer time
ℓ_{sto}	I/O request (disk) latency
b_{sto}	Storage per-byte transfer time
r_s	Local storage readahead
Per-pattern-set inputs	
\mathbb{P}	Set of access patterns with process mappings
p	I/O participants (clients)
m	I/O participants per node
Coarse-grain model input/outputs	
B	Total request size across all patterns
n_p	Servers contacted per client
r	Average request size per client per server = $B/(pn_p)$

thrashing.

b) Coarse-Grained Model: The coarse-grained model is a generalization of the cost model created by Song et al. [8] to optimize accesses under the following characteristics: uniform access sizes, perfect access distribution among servers corresponding to PVFS data layouts, and single time of issuance across all processes. This model, while not created for general-purpose I/O modeling, has proven useful for HPC applications with regular access patterns and is appropriate for driving our replica placement method, given that we are in full control of replica placement and can produce such regular accesses. First, we discuss the model and generalization; then we discuss how we find effective replica layouts using it.

The cost model by Song et al. has four separate costs that are summed to find the final result: T_e , the establishment time for all network operations, T_x , the time to transfer all request data across the network, T_s , the “startup” time for all storage accesses, and T_{rw} , the read/write time for all storage accesses. These costs are computed separately based on the PVFS data layout being used, which they fix to be a one-to-one process to server mapping (1-DV), a one-to-all mapping (1-DH), and a process-group to server-group mapping (2-D). Refer to Song et al. [8] for more details. We collapse this mapping based on a simple observation: the parameter being varied across each

of the models is the *servers contacted per client*. Making this an explicit variable n_p allows us to collapse the equations into a single set:

$$T_e = \max(mn_p, \lceil \frac{pn_p}{n} \rceil) \ell_{net} \quad (1)$$

$$T_x = \max(mn_p, \lceil \frac{pn_p}{n} \rceil) r b_{net} \quad (2)$$

$$T_s = \lceil \frac{pn_p}{n} \rceil \ell_{sto} \quad (3)$$

$$T_{rw} = \lceil \frac{pn_p}{n} \rceil r b_{sto}. \quad (4)$$

Note that the n_p term is within the ceiling/maximum functions, and not factored out, to ensure the model will remain accurate for low-client-count scenarios (i.e., $pn_p < n$).

c) Coarse-Grained Model Usage: Given the definition of the coarse-grained model, we derive a simple replica creation process, using the following strategy. Assume the underlying accesses are regular and uniform, compute $\min_{n_p} (T_e + T_x + T_s + T_{rw})$, and then resolve any load balances by over/under provisioning replica striping across the servers. After computing B and an average m , simply calculate model values for $n_p \in \{1, 2, \dots, n\}$ and choose the minimum. Next, perform the logical striping under the assumption that r is the actual request size per client. Then, compute for each pattern which servers its data resides on. An example mapping is shown in Figure 2. The intuition behind this layout heuristic is that patterns with balanced accesses will be optimized as normal, and overprovisioning for unbalanced accesses with larger relative sizes will be made up for by underprovisioning for accesses with smaller sizes, mapping degree of concurrency to relative access size.

d) Fine-Grained Model: The fine-grained model shares similarities with the coarse-grained model, using latency/bandwidth modeling at both the network and the storage levels to generate an overall cost. However, whereas the coarse-grained model makes some significant assumptions about access characteristics, we need a more robust model capable of capturing load imbalance. Our approach consists of the following two steps, with the underlying steps of

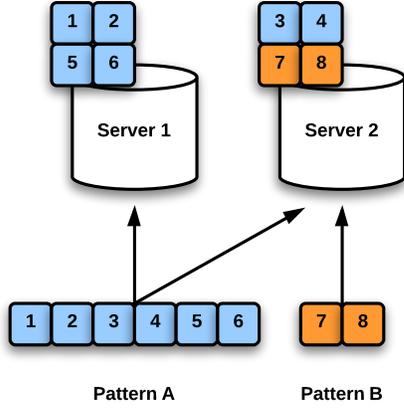


Fig. 2: Access pattern over and under provisioning based on model optimization on balanced accesses (for $n_p = 2$)

mapping each pattern in the pattern set to its respective client process/node and set of contacted servers: (1) compute a localized T'_e , T'_s and T'_{rw} for each server, and (2) calculate the total time to receipt T'_t for each client node based on the server calculations, with the maximum among them being the total request time.

The computation of T'_s and T'_{rw} is relatively straightforward, with T'_s being the number of noncontiguous blocks (measured at a page granularity and taking into account readahead) accessed times the storage access latency and T'_{rw} being the total size of all requests to the server times the inverse of the bandwidth. For T'_{rw} , we additionally adjust the performance to account for readahead: if two consecutive requests are within a readahead window (default 128 KB on Linux), then the disk latency cost is avoided at the cost of consuming the bytes separating the two requests.

The computation of T'_e and T'_t are more nuanced, since we must consider both access latency and wait times for request/response receipt. Because computing these wait times exactly would require a known access schedule (as well as a more architecturally accurate simulation), we instead compute an approximate. For T'_e , we compute the network latency times the number of incoming requests, with a *penalization term* ϵ_e . For this, we take the node contacting the server with the maximum number of outgoing requests, and we assume that the request to the server occurs after all of its other requests. Similarly for T'_t , the penalization term ϵ_t is computed by assuming that the data requested by the given node is issued after the server’s access schedule, for the contacted server with the largest load.

Considering the full set of pattern accesses, the computational cost of evaluating the model is best described by using a bipartite graph $G = (M, N, E)$, where M represents the client nodes, N represents the server nodes, and E the mapping of client nodes to contacted server nodes. Generating the graph requires time proportional to the number of contiguous blocks encoded by the pattern set times the average number of servers containing each block (at most the average vertex degree). Evaluating the model, akin to traversing each vertex and its

corresponding neighbor set, is linear in $|E|$.

3) *Replica Aging*: An important consideration in the RADAR methodology is how to represent the “age” of a particular pattern (or set of patterns), with the implication that the layout manager can prefer “younger” accesses to “older” ones. While raw timestamps can easily be gathered by tracers, real time as a measure of age is noisy and rife with pitfalls: machines can go down, scientists can go on vacation or not run applications over the weekend, and so forth. Hence, we need a more robust measure of time.

To provide a robust measure of age relative to application runs, RADAR defines age to be the amount of data read from the dataset, normalized by the original dataset size. For example, consider a simulation that writes X bytes of analysis data. Afterwards, consider three read-only analysis application runs (in order of execution): A , B , and C . Under our system, patterns generated during application A ’s run would have an age of zero, patterns generated during B would have an age of $\text{read}(A)/X$, and patterns generated during C would have an age of $(\text{read}(A) + \text{read}(B))/X$. The metadata required to maintain the ages, namely, the total amount of data read from a dataset, can be stored in the file metadata of the RADAR MPI file. While this method has some issues of its own, such as a bias toward larger applications, it nevertheless sufficiently captures the temporal context necessary to compare the relative age of access patterns.

Given this formulation of age, we use a configurable *decay* function, applying on the performance benefit of already existing pattern sets. That is, given a benefit measure b and the difference in age a between the current dataset state and the age at which the pattern set was generated, the result of the decay function $\gamma(b, a)$ is used in place of b in the replica selection process. Examples are the identity function ($\gamma(b, a) = b$) and an exponential decay function such as *half-life* ($\gamma(b, a) = b2^{-a/X}$, the half-life in this case corresponds to a reading of the full dataset).

B. EOF Data Management

1) *Background – PVFS and EOF*: Recently, the “End Of Files” (EOF) [13] extension to PVFS [1], a high-performance parallel filesystem, was created to expose the object storage abstraction directly into the client space, allowing a richer and more elegant mapping from application datasets to parallel storage systems. The object storage abstraction presents uniquely identified (typically via a 64-bit ID), extendible linear byte arrays as the elemental unit of storage; and most parallel filesystems today distribute a file to a set of objects using a distribution function, or *striping*. EOF cuts out the middleman between file and object set and allows applications to forward I/O operations directly to individual objects. For example, dataset metadata can be forwarded to a single object, while the data itself can be assigned distinct objects based on timesteps, variables, and so forth.

2) *RADAR Object Layout*: Since EOF exposes object-based storage and most components of our method operate on EOF objects, we first discuss the object layout of the various data/metadata in our system. More specific details can be

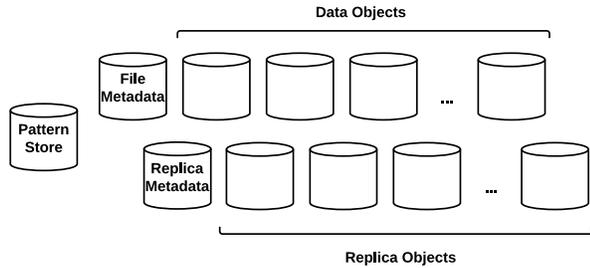


Fig. 3: EOF object layout for RADAR.

found in the system components’ respective sections. Figure 3 shows the various types of objects used in RADAR, all under a single filename. The original dataset is stored as it would be on any distributed filesystem, striped by some distribution across multiple storage devices (objects, in our case). We include a file metadata object since distribution in EOF is relegated to the user. For RADAR, at file create time we allocate a number of replica objects equal to the number of data objects. We do so both for practical reasons (limits on per-file concurrency) and for semantic reasons (currently, EOF cannot dynamically add or remove objects from a file container). A replica metadata object is used to store the mapping of replicas to object locations. The set of generated access patterns processed by RADAR is placed in a dedicated object for the results to persist across multiple application runs.

Under this data management scheme, only two sets of data exist outside the MPI/EOF file container used by RADAR: the trace output and trace analysis results. For completeness, we intend to integrate these intermediate datasets into the RADAR format, though an option for external output is still important as a tracer of the given granularity can potentially be useful for general I/O performance analysis.

3) *Replica Object Storage Strategy*: A yet undiscussed component of the RADAR process is how exactly to distribute a replica. The problem arises from the fact that we are using a shared set of replica objects to store multiple replicas with heterogeneous distributions.

To both simplify design and minimize usage of space, we exploit sparse-file capabilities in the local filesystems employed by PVFS. Essentially, file blocks not written to are not stored on the disk. The typical example is a file that is created on open, and written to once at byte offset 1GB. The file size reported by stat will simply be the last byte offset written, while the actual storage used will be a single disk block, along with metadata.

We divide all objects into allocation units we call *object domains* (ODs), an example of which is shown in Figure 4. An OD corresponds to the full set of replica objects, spanning a per-object address space with fixed, large-granularity sizes. Each replication is placed within a single OD, as shown in the example. In order to avoid biasing replica placement toward one object or another, replicas are assigned starting objects in round-robin order. In the figure, for example, the replica shown begins addressing at the leftmost object, while the next replica created will begin addressing at the next leftmost object.

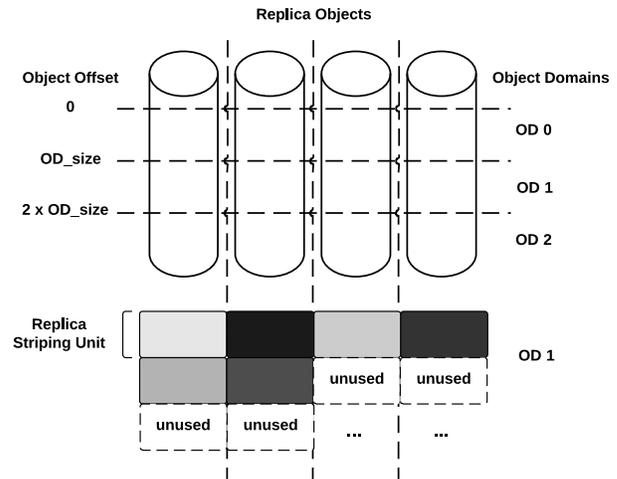


Fig. 4: Allocation units in RADAR (“Object Domains,” or ODs), and replica layout in an OD.

C. I/O Tracer and Analyzer

1) *Background – MPI-IO, ROMIO, and ADIO*: In the MPI standard, MPI-IO exposes a datatype-driven file decomposition and operating method (“views”). That is, MPI datatypes are defined much like they would for communication, except the locations are defined with respect to a linear byte store beginning at offset 0. This approach allows users to easily map data structures in memory to those in file. For example, reading and distributing multidimensional arrays can be simply performed by use of the `subarray` type for each process.

ROMIO is the implementation of MPI-IO [14] used in MPICH. ROMIO is implemented by mapping MPI-IO calls to the ADIO interface [15], which exposes necessary functionality in order to portably implement MPI-IO and allows separation between MPI functionality and the underlying filesystem calls (e.g., POSIX vs. native PVFS). Furthermore, the design of ROMIO is such that filesystem-specific ADIO targets can be dynamically specified through the filename argument by prepending the implementation name followed by a colon (e.g., “lustre:” and “pvfs2:”). This implementation is enabled by loading tables of function pointers of the particular filesystem’s implementation of ADIO based on either system defaults or user requests.

2) *ADIO Tracer*: With respect to MPI, our layout optimization techniques need to be aware of how MPI-IO calls translate into actual I/O requests. For example, optimizations such as two-phase or data sieving may be applied transparently to the user. PMPI-based (or POSIX-based) tracers make this difficult – neither have the sufficient information to distinguish the difference between logical accesses and physical (a PMPI tracer could conceivably perform this but would require reverse engineering of collective optimizations).

Hence, our tracer is designed as an ADIO implementation, outputting the ADIO function calls as well as offset/length pairs (either before or after applying collective optimization). By doing this, we gain the flexibility of examining the I/O request mapping induced by collective optimizations, while freeing up the namespace for other PMPI-based methods, such

as Darshan [16], [17], or even related works that modify data layouts (see Section IV). Within the MPICH source, we enable this primarily through function pointer swapping and auxiliary data structures added to the MPI file data structure. Hints passed via the `info` object enable and set options for the tracer; and, if requested, open calls are synchronized to allow for comparing times of operations in order to determine concurrency of accesses.

To enable trace functionalities in a generic manner, we simulate I/O calls from within the trace functions. While this approach may incur some computational overhead compared to directly embedding within specific ADIO implementations (such as our own), we feel that the generic ROMIO-level tracing can find use beyond this work, and we will explore it further. Simulating I/O involves iterating over the file datatype, generating offset/length pairs corresponding to file accesses.

Through simple configuration options (either MPI hints passed through an `info` object or environment variables), we can easily allow our simulated I/O to be applied immediately upon reaching a collective I/O request (before collective optimizations), only in the underlying serial I/O operations, or both, taking advantage of the fact that a single collective read/write is composed of an optional data exchange phase followed by numerous serial read/writes.

One issue of concern for tracing and trace analysis is determining whether operations on different processes are concurrent, and possibly conflicting. To provide sufficient information for this, we perform a barrier at file open time, take the wall-clock time, and record only per-process time deltas. This approach avoids performing clock synchronization while giving us a reasonable indication of when operations occur relative to one another, sufficient for this work.

The output of the tracer includes all ADIO calls, as well as any offset/length pairs generated through reading or writing, separated on a per-process basis. Currently, the trace is in plain-text format, and we do not attempt compression; compression methodologies such as inline pattern analysis and/or off-the-shelf compressors (e.g., Zlib [18]) will be explored in future work.

3) *Trace Analyzer*: The access patterns we are primarily concerned with are contiguous access patterns (sequential access of a large space with fixed or average-size request sizes) and k -d strided access patterns (accesses that differ in offset by a fixed value, or stride), both of which are common in HPC I/O workloads. These accesses occur over a linear address space of bytes but in practice typically correspond to accesses along spatiotemporal domains or across multiple variables (e.g., temperature, pressure in a climate simulation).

To gather the desired access patterns, we built a variant of the IOSig trace analysis software [19], [20]. We similarly use a template matching approach; but since our tracer works at the ADIO level and additionally processes datatypes, the processing of the traces has been rewritten. For more discussion pertaining to access pattern categorization and discovery, see [19], [20].

D. Replica-Aware ADIO Driver

The responsibilities of the RADAR ADIO driver are to interface with EOF, maintain the semantically varying sets of objects, and remap I/O requests into the replica space, as appropriate. Aside from the remapping portion, the rest of the processing is relatively simple, corresponding to loading/distributing file and replica metadata, updating the bytes-read component (for later use by the layout manager), and driving some RADAR-specific operations, such as performing the replication. The replication process itself is discussed in the next section, while the request remapping is presented in the section thereafter.

1) *I/O Processing in the Presence of Replicas*: Given a set of replicated data layouts and a set of I/O requests (e.g., generated by a call to `MPI_File_read`), the most difficult task for the RADAR I/O driver is to determine which, if any, replica to read/write from. This problem can be separated into two tasks: matching requests (offset/length pairs in file) to replicas, then choosing which among the replicas to issue the operation to, or none at all. Note that for this work we consider only full matches, in which each contiguous block of a request can be fully satisfied through reading from a single replica. Partial replica mapping, resulting in reading partly from a replica and partly from the original data, is a focus for future work; our intuition is that splitting a single request into multiple locations will result in increased latency costs, especially in a multireader environment. Answering this concern fully, however, requires more sophisticated performance modeling.

a) *Replica Matching*: The matching itself, given an offset/length pair and a replica, is a simple matter; but the main concern is coping with an increasing number of replications. For n replicas, it is preferable not to have to do a naive linear test of each one: for a large number of small, localized replicas, this approach would clearly not scale. Furthermore, arbitrary replicas over strided data (multiple dimensionalities, block sizes) make using classical spatial-partitioning data structures such as interval and R-trees unwieldy: the strided pattern would need to be flattened into its elemental blocks, expanding the tree size considerably. Replicas over contiguous data can benefit from these structures, however.

As a compromise between flattening the strided pattern representation and inducing a linear scan of the full replica set, we build an *inverted index* [21], [22] over the file. As the name suggests, an inverted index maps regions of file to a list of replicas overlapping with the regions, rather than mapping replicas to regions of file. Typically, inverted indexes have been used to accelerate searches such as large-scale document searching: search terms match to a set of containing documents, but they have also proved useful in other scientific data processing scenarios [7], [23]. First, we partition the extent of the file into *bins*, where each bin represents a distinct contiguous region of the file. Then, for each bin we list the replicas for which locations encoded by the replica overlap with the bin's byte boundaries. Hence, queries (i.e., finding the replicas that may satisfy a given offset/length pair) need only look at and process the list of replicas in the bins that intersect with the request. This representation is shown in Figure 5.

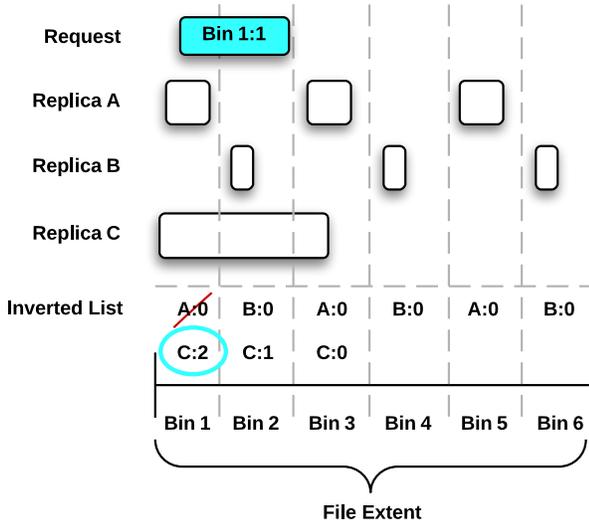


Fig. 5: Replica lookup using inverted index with bin extension

One problem with the inverted index approach is that multiple bins may need to be examined, and their replica lists intersected, in order to fully process queries. To avoid paying this additional cost, we perform a simple modification of the per-bin replica list to enable full querying using only a single bin. Namely, for each entry in the list of per-bin replicas, we record the *bin extension*, or the number of bins that the overlapping contiguous block extends across (in increasing byte order). For replicas over strided data, we record the maximum extension over all blocks that overlap in the bin. Then, given an offset/length pair to query on, we need only look at the intersecting bin with the smallest byte boundary and compare the bin extension of the request to the bin extension of the replica. This modification is also represented in Figure 5, with the numbers next to the identifiers representing the bin extension.

To help mitigate a worst-case linear search time of the inverted index (all replicas overlap with a particular bin) for every request, we additionally keep a one-element history of replica matches, with the assumption that it is more likely than not that a replica will be read multiple times in sequence by a client. For processing a request, the previous replica read is matched against first, followed by a query against the inverted index if the request does not match.

b) Replica Selection: Given an I/O request and a set of replicas that can satisfy the request, a significant problem is how to select which replica among the set to read from that would result in the best performance. This choice is nonexistent for writing, as all replicas must be updated: as noted earlier, our primary use-case is accelerating read performance for multiple analysis workloads; mixed read-write workloads would not fare well under our methods. Complicating the selection is that the available information to make this decision is inherently local (requests occur on a per-process or possibly per-MPI-aggregator level), consisting of only the replica metadata.

Our solution to this problem is a simple heuristic we call

TABLE II: Performance Model Variables

System parameters		
n	8	I/O servers
r_s	128KB	Local storage readahead
ℓ_{net}	$32.9\mu s$	I/O request (network) latency
ℓ_{sto}	$6.20ms$	I/O request (disk) latency
b_{net}	$0.00112\mu s$ (867MB/s)	Network per-byte transfer time
b_{sto}	$0.0212\mu s$ (44.98MB/s)	Storage per-byte transfer time

smallest containing block (SCB). The idea behind SCB is that of *specialization*: we consider replicas with a finer granularity to be more specialized than those without and hence should be prioritized in the replica selection process. This generally means that replicas over strided data will more than likely be selected over replicas over contiguous data, as each of the strided data would be more sparse.

Other methods, such as performance models, could certainly be “swapped” in to make the replica selection as well, as the data layout is known to all. However, inherently local performance models cannot have a big-picture view of the system by which many I/O operations could be happening at any given time. In this sense, we are relying on the RADAR layout manager to make informed choices pertaining replicas, and allowing the I/O mechanism itself to be simpler.

III. EXPERIMENTAL EVALUATION

All experiments were run on the Fusion cluster at Argonne National Laboratory. Each node contains two quad-core Intel Xeon processors at 2.53 GHz with 32 GB RAM, and nodes are connected by InfiniBand QDR. Each node in Fusion contains local hard-disk storage (250 GB IBM iDataPlex). Additionally, our implementation of RADAR is based on MPICH 3.0.2 and PVFS2 2.8.1, patched with EOF. Because of issues with InfiniBand support for PVFS on Fusion, both MPI communication and PVFS client-server communication are performed via TCP over InfiniBand.

A. RADAR-Specific Setup

Since we use a modified version of PVFS and since each node in Fusion has local storage, we assign a subset of the nodes to serve as PVFS I/O servers and use the remaining as I/O clients. We use eight I/O servers in all experiments. Hence, each server initially contains 8 GB of data, striped using 1 MB blocks.

Table II shows the performance model parameters we gathered via microbenchmarks on Fusion. We use the BMI pingpong utility in PVFS to gather network performance through PVFS, where BMI (Buffered Message Interface) is PVFS’s client/server communication interface. We use simple read benchmarking via collocating a PVFS client with a server to gather storage performance parameters. Note that the microbenchmark result for disk bandwidth is much lower than expected: the bandwidth when not going through PVFS is 90 MB/s. We were unable to eliminate this discrepancy, but we believe it to be a result of internal threading and buffering overheads on the PVFS server.

For our inverted list acceleration structure for replica lookup, we divided the file into 1,024 bins, each of which covering a 64 MB extent of data.

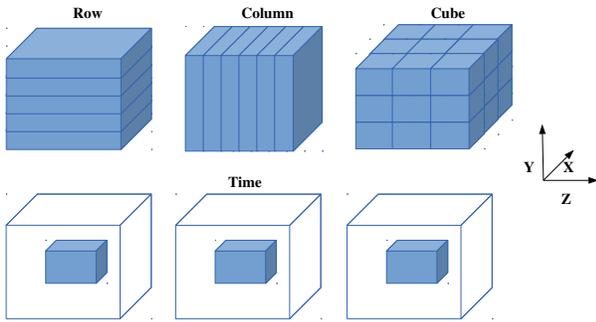


Fig. 6: Subvolume decompositions used in our evaluation (contiguous in order Z, Y, X , time).

B. Benchmarks

We evaluate our layout optimization work within the context of multidimensional array decomposition, a common set of accesses seen in numerous analysis workloads. Specifically, we define a three-dimensional volume of data over a fixed number of contiguous timesteps and use MPI-IO to read selections of the resulting dataset using four types of decompositions, shown in Figure 6: row-wise (distribute volume by contiguous plane), column-wise (distribute volume by noncontiguous plane), block-wise (distribute volume by 3D subvolume) and timestep-wise (distribute single subvolume from a range of timesteps). The row-wise decomposition induces contiguous patterns at each process, while the remaining decompositions induce multidimensional strided patterns at each process, or contiguous patterns at each aggregator process in the case of collective I/O. Note that these access patterns are a superset of the access patterns exhibited by several well-known benchmarks such as MPI-Tile-I/O [24], IOR [25], and PIO-bench [26], all of which perform accesses with regular (single- or multidimensional) strides.

For all experiments, we used a subvolume of (time, X, Y, Z) dimensions (in row-major) $128 \times 256 \times 256 \times 256$, each element of which is a 32-byte structure (e.g., four C doubles). The total size of this dataset is 64 GB.

C. Decomposition Performance

We test each decomposition using the following MPI process configurations with respect to performing I/O: independent I/O with all processes on each node participating, independent I/O with one process per node participating, and collective I/O with one aggregator per node. The first is a “naive” approach for parallel I/O, but one that is still used because of its simplicity as well as for access patterns such as log-structured, which structures data in the order written (typically for checkpointing/restarting, but some libraries such as ADIOS use log-structured with respect to each process’s writes). The second is similar to the first but represents the computation model of MPI+threads or MPI+accelerator, in which a single process on each node performs communication and I/O while intranode parallelism is used for computation. The third is more common because it allows optimizations such as two-phase I/O [4], although with the overhead of communication and data movement between I/O participants.

Figures 7, 8, 9, and 10 show performance under the different decompositions both before and after RADAR data replication. For these runs, we synchronize prior to running the decomposition and calculate bandwidth with respect to the maximum elapsed time for each individual read. The following points about these experiments are of interest.

- 1) All decompositions except the time-based decomposition decompose the same overall data size of 2 GB (four timesteps of 512 MB volumes). Thus, with an increasing number of clients, the average request size decreases and the number of requests increases, leading to potentially less efficient access when not using collective I/O.
- 2) The time-based decomposition defines a fixed-size subvolume for each client to read of size 64 MB. As the number of clients increase, the per-client requests remain the same, leading to an increase in the total request size.
- 3) The cube decomposition divides the volume into perfect cubes (1, 8, 27, 81, 125, etc.) no less than the number of clients, and clients are assigned multiple blocks to read, resulting in varying request granularities based on the number of clients. For instance, a four-client run will divide the subvolume into eight blocks and assign two blocks to each client. This can lead to both load imbalance (processes can be oversubscribed blocks compared with others) and varied access patterns due to the possibility of multiple smaller blocks combining into a single, large, contiguous block.
- 4) The difference between having a single client per node and having up to eight is marginal, because of the much larger performance potential of the network vs. storage.

Figure 7 shows the time-based decomposition. Here, I/O aggregation was disabled by ROMIO because of the per-process data being both non-interleaved and separated in storage; thus, it is not shown. Without replication, the aggregate performance is far below peak performance because of the noncontiguous accesses. The use of data reorganization through replication enables high performance across the spectrum, although tapering off once I/O servers begin processing requests from multiple clients.

Figure 8 shows the cube-based decomposition. This decomposition results in block sizes of high variance with a changing number of clients, which is a significant factor in the overall performance. In the figure, the performance implications can easily be seen between the four-client and eight-client decomposition for the nonreplication case, and the eight-client and 16-client decomposition for the RADAR case. Regardless, the use of RADAR helps smooth out the performance characteristics as a result of making the strided accesses contiguous per client and over/under-provisioning of replicas based on load. Additionally, for the small-client case, we notice performance regressions between the no-replication and replication case. We are currently unable to diagnose this difference; the generated layout by RADAR is the same and the I/O driver follows largely the same code path. Unfortunately, collective I/O results were unable to be gathered; we encountered crashes in the ROMIO implementation and/or MPI datatype processor when attempting to run our workload.

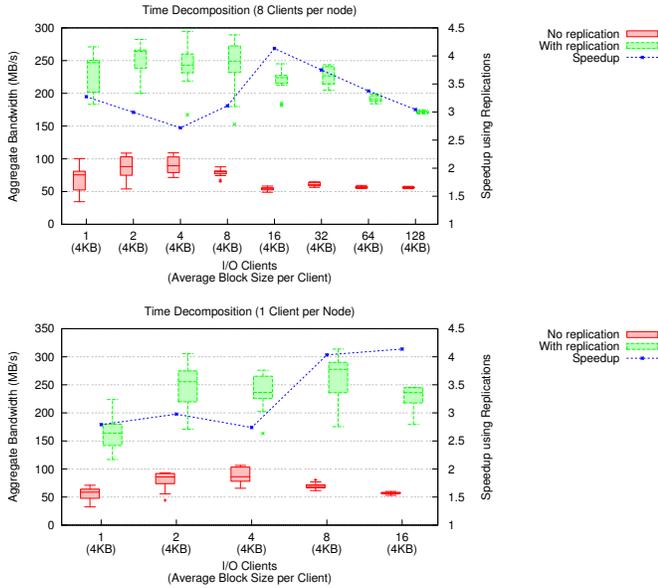


Fig. 7: Subvolume-over-time-decomposition results with different process configurations

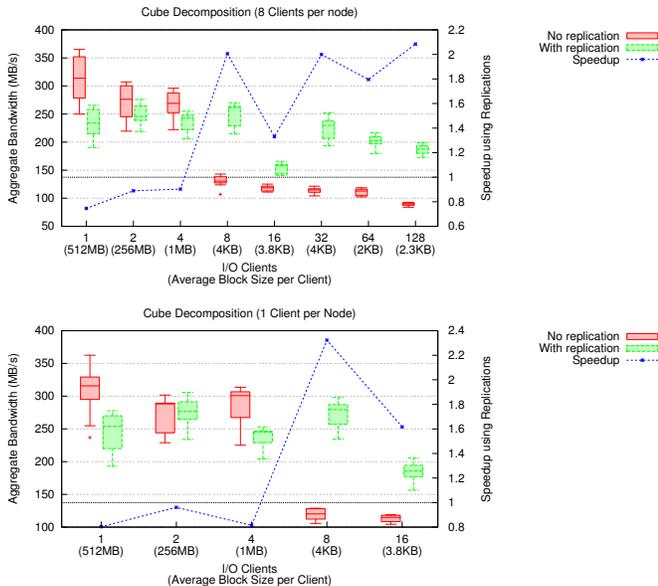


Fig. 8: Cube-decomposition results with different process configurations

Figure 9 shows the column-based decomposition performance. This represents a pathological case of I/O, as seen by the average contiguous block sizes. Hence, performance without collective optimizations or RADAR is far worse than any of the other decompositions as the number of clients increase. RADAR can greatly improve performance over the original data layout but tapers off for increasing client counts. The reason, aside from the increased number of requests per server, may also be implementation-based: the RADAR MPI-IO driver (along with the native PVFS2 driver) enter processed datatype offset/length pairs into a queue, then use PVFS2 *list I/O* [27] to send the list of requests to the I/O server in one go

(list I/O is most similar to the previously proposed POSIX I/O extensions `readx/writex` [28], an interface for specifying sets of noncontiguous file accesses to noncontiguous memory). The block size is so small in this case that the queue becomes full and issues correspondingly small requests. For fairness, we use the same queue depth of 64 as used by the PVFS2 ADIO implementation (nonconfigurable). As for collective I/O, RADAR exhibits the same regressions for small client counts as seen in the other benchmarks, as well as regressions for larger client counts. Performance for large numbers of clients drops likely because of underestimating the network capability when choosing the replica layout: see the following paragraphs on model performance.

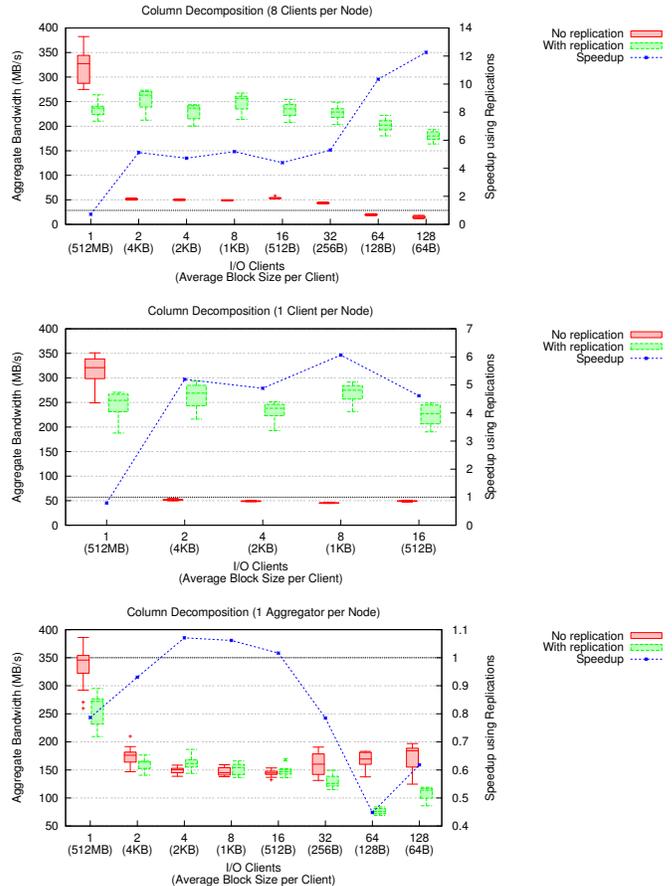


Fig. 9: Column-decomposition results with different process configurations

Figure 10 shows the row-based decomposition performance, representing the “best case” for parallel I/O without reorganization: large, contiguous, non-overlapping blocks. Here, since the storage is the primary bottleneck and block sizes are very large, RADAR has little benefit. We also show collective I/O results, but in this case performance is either unchanged or reduced because of the data movement overheads.

D. Model Verifications

We now look at how the performance modeling approach compares with the performance shown in Section III-C. The goal of the performance models is to show whether a specific

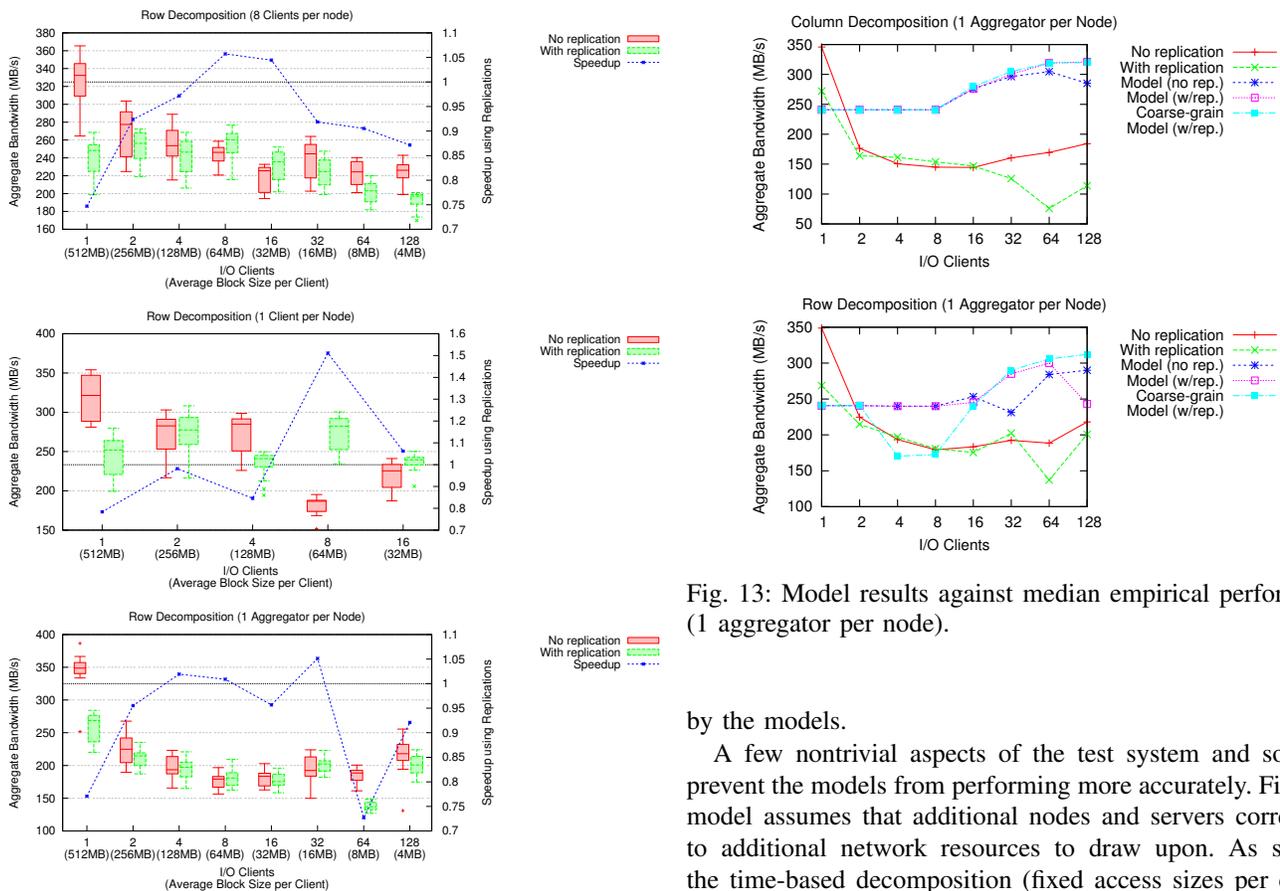


Fig. 10: Row-decomposition results with different process configurations

data layout can be improved by a modified data layout via replication. Note that this goal is different from strict performance accuracy: here, the primary measurement of interest is the accuracy of relative performance between two layouts (one with replicas, one without). Since the models do not perform full system simulation, they are unsuitable for general-purpose performance prediction.

Figures 11, 12, and 13 show the results, comparing the model-derived performance of both the original layout and the layout under replication with the median of the performance shown in Section III-C. We additionally show the estimated performance using the coarse-grained model corresponding to the best layout. In general, the “best layout” found corresponded to the heuristic of spreading each pattern’s data across as many servers as possible until overlap occurs, in which case the distribution contracts accordingly. Also, as discussed, results for time-based and cube-based collective decomposition are not shown.

Overall, the model results capture some degree of performance difference between the original layout and the candidate replica set layout. Large, contiguous accesses are shown to have little difference between both the original and replicated layouts, implying that creating the replica set would result in minimal, if any, gain. Alternatively, smaller, noncontiguous accesses are correctly shown to have a large degree of benefit

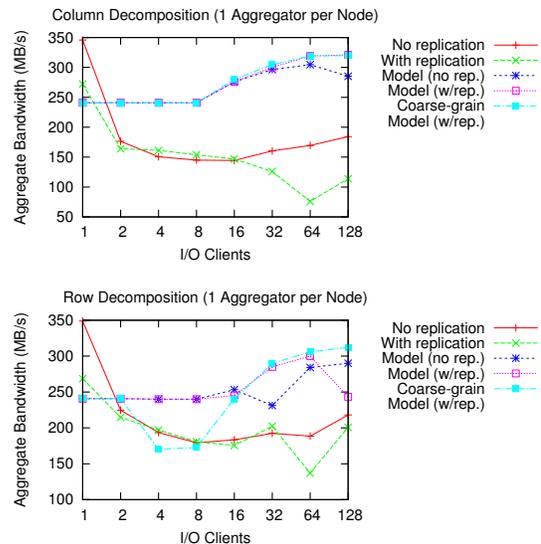


Fig. 13: Model results against median empirical performance (1 aggregator per node).

by the models.

A few nontrivial aspects of the test system and software prevent the models from performing more accurately. First, the model assumes that additional nodes and servers correspond to additional network resources to draw upon. As seen in the time-based decomposition (fixed access sizes per client), this is not the case. Hence, more accurate, architecture-specific network modeling is needed. Furthermore, the cost models generally overestimate the performance: we believe this overestimation to be the cause of our readahead simulation being optimistic in its ability to effectively cache pages without interfering with normal system performance. Our performance models also do not model the communication phase of optimizations performed during collective I/O and hence overestimate the overall performance.

E. Replica Inverted List Performance

To test the efficacy of our inverted list approach to replica lookup, we measured the median entries per bin for all the experiments shown in Section III-C. For these measurements, bins having no entries were discarded from the computation: we wish to measure effectiveness only for regions of file accessed by the read benchmarks. Furthermore, we used an internal threshold of ten for enabling the inverted list and, hence, have no results for run configurations that did not produce enough replications.

The results can be seen in Table III. For the individual I/O case of the read benchmarks tested, each process’s read workload reduces to a multidimensional strided access pattern and hence produces exactly one replica, meaning that the upper bound on the number of replicas is the core count. The effectiveness of the method depends on the degree of interleaving: fine-grained interleaving at the level of bytes, as seen by the column decomposition, obtain no benefit, since each replica touches all nonempty bins. In contrast, the other

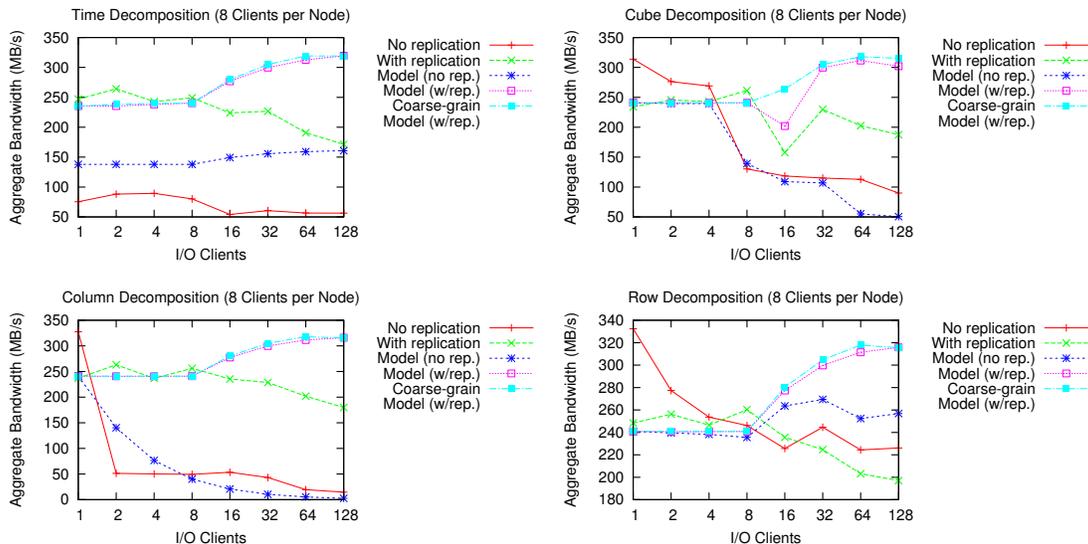


Fig. 11: Model results against median empirical performance (8 clients per node).

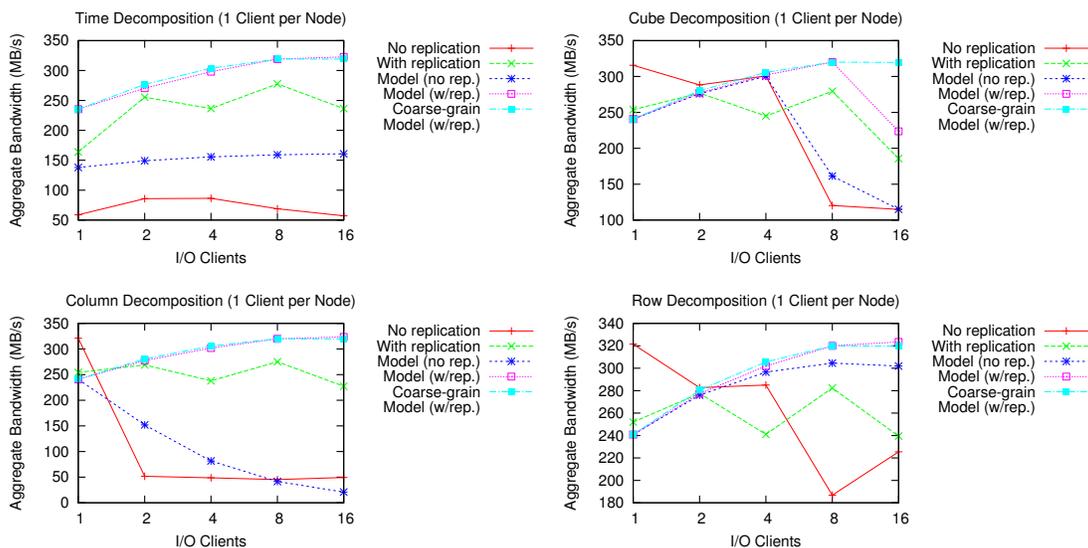


Fig. 12: Model results against median empirical performance (1 client per node).

decompositions see a larger degree of benefit, especially the time-based decomposition, in which each process accesses a distinct, nonoverlapping portion of file with a large separating stride.

IV. RELATED WORK

We divide our discussion of related work into three parts: replication in storage systems, replication outside storage systems, and detecting I/O access patterns.

A. Replication in Storage Systems

Data replication in storage systems is a well-researched topic in many domains. Many parallel/distributed filesystems, such as Hadoop File System (HDFS) [29] and Google File System (GFS) [30], [31], built for task-centric, data-intensive

TABLE III: Median replica entries per nonempty inverted list bin

Decomp.	Config.	Clients			
		16	32	64	128
Col	ind., 8/node	16.0	32.0	64.0	128.0
	ind., 1/node	16.0	N/A	N/A	N/A
	collective	N/A	1.0	2.0	4.0
Row	ind., 8/node	2.0	4.0	8.0	16.0
	ind., 1/node	2.0	N/A	N/A	N/A
	collective	N/A	N/A	1.5	2.0
Cube	ind., 8/node	6.0	8.0	16.0	36.0
	ind., 1/node	6.0	N/A	N/A	N/A
Time	ind., 8/node	1.0	1.0	1.0	1.0
	ind., 1/node	1.0	N/A	N/A	N/A

workloads, as well as the Ceph filesystem [32], have data replication as a first-order feature. Furthermore, for current filesystems used in HPC (e.g., PVFS [1], Lustre [2], PanFS [33])

where fault tolerance is typically provided through hardware redundancy, data replication is beginning to be explored. For example, Tantisiriroj et al. developed a *shim* layer for PVFS for the purpose of executing Hadoop workloads, enabling readahead buffering, making striping information available to the Hadoop scheduler, and modifying PVFS's data placement policy to provide data replication [34]. In addition, local filesystem replication has been explored in local filesystems in a performance context by reorganizing data to minimize rotational latency and maximize locality [35], [36].

Database systems also widely use replication, both as a fault tolerance method similar to filesystems and as a performance optimizer by replicating "chunks" of the database using query history as a guide [37], [38]. Also, recent works in MonetDB [39] have enabled dynamic replication of columns via a method called *database cracking* [40], [41], which replicates columns and reorganizes/indexes the replicated data as queries are performed on it.

Other areas such as on-demand (e.g., video) services [42], [43] and data grid systems [44], [45] have used replication to maximize data availability and throughput.

B. I/O Middleware and User-Level Replication

Recently, the use of replication to ensure high availability or improve performance have been explored outside the storage system, that is, as either new high-level libraries or alongside I/O middleware. Son et al. [46] show the possibility of handling replication-based fault tolerance at the middleware level, performing file block replication using the PMPI interface. For MPI-based applications that use a one-to-one, process-to-file mapping, Song et al. [8] created three replications with different file-to-disk mappings: file-per-storage-node, full striped files, and partially striped files. Yin et al. extended Song's model to handle one-dimensional and two-dimensional strided accesses [47]. Zhang et al. [9] used replication through PMPI to minimize disk head thrashing by playing back local disk traces with DiskSim [11].

C. Capturing and Detecting I/O Access Patterns

Because gathering and systematically deriving system usage information from an application or set of applications is a highly important task, many solutions have been developed that attack the problem from a multitude of directions. The majority of these methods cover more areas than just I/O. Numerous methods have been developed to work specifically on MPI and PMPI (MPI's profiling interface), such as the MPI Parallel Environment (MPE) [48] for full MPI event tracing and mpiP [49] for lightweight, statistical measures. Dynamic instrumentation methods include automatic instrumentation at compile time through source code analysis [50], as well as runtime binary instrumentation through IOPin [51], based on the Pin [52] framework. The ScalaTrace family of MPI tracers focus on compressed trace generation [53], [54], [55], [56], using histogram generation and a combination of intranode and internode trace compression. Additionally, Darshan [16], [17] focuses on *center-wide* usage patterns by combining local, subsystem metrics (such as block device profiling with the

Sysstat [57] and fsstat [58] tools) and application-level metrics (instrumented through POSIX and MPI-IO).

Once acceptable profiles or logs of application/system performance are gathered, they can be mined for emergent patterns. Statistical learning methods can be used in a general sense to capture high-level patterns such as block-to-block association [59], [60], [61], [35]. Recent methods specifically for HPC have been developed, again typically through the MPI/MPI-IO layers. For example, Byna et al. developed an MPI-based I/O prefetching methodology based on detecting multidimensional striding patterns of varying sizes [19]. Also, He et al. investigated PLFS index compression using pattern recognition under a checkpointing use-case [62]. The IOSig [19] trace analyzer convert I/O operations to compact and parameterized representations called I/O signatures using a *template matching* approach, which iteratively attempts to match specific patterns (e.g., regularly strided) to the sequence of I/O accesses.

V. CONCLUSION

Effective data distribution in large-scale analysis systems is an integral component of achieving high-performance I/O, especially in the presence of complex, noncontiguous workloads such as the volume decompositions we have presented. Through the tight coupling with a filesystem view of the data as a set of distinct objects, we were able to create arbitrary data layouts optimized for the access patterns induced on the dataset, all in a single container. RADAR is a promising step in the direction of automated specialization of data layouts based on application-specific needs and access patterns, providing both increased performance and an initial ability to reason about the "worth" of layouts for the purpose of marshalling usage of limited space for optimized data distributions. We also have shown that optimization-aware tracing methodologies (i.e., MPI-IO two-phase aware) can be an effective tool for adaptive layout optimization works, alleviating the optimization layer from needing a deep understanding of intermediate optimizations.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

REFERENCES

- [1] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 317–327.
- [2] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, 2003.
- [3] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1083323.1083349>
- [4] R. Thakur and A. Choudhary, "An extended two-phase method for accessing sections of out-of-core arrays," *Scientific Programming*, vol. 5, no. 4, pp. 301–317, 1996.

- [5] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C. S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "ISABELA-QA: query-driven analytics with ISABELA-compressed extreme-scale scientific data," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC. New York, NY, USA: ACM, 2011, pp. 31:1–31:11. [Online]. Available: <http://doi.acm.org/10.1145/2063384.2063425>
- [6] Z. Gong, D. A. B. II, X. Zou, Q. Liu, N. Podhorszki, S. Klasky, X. Ma, and N. F. Samatova, "PARLO: PArallel Run-time Layout Optimization for scientific data explorations with heterogeneous access patterns," in *the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13)*, Delft, The Netherlands, 2013.
- [7] J. Jenkins, I. Arkatkar, S. Lakshminarasimhan, D. A. Boyuka II, E. R. Schendel, N. Shah, S. Ethier, C. Chang, J. Chen, H. Kolla, S. Klasky, R. Ross, and N. F. Samatova, "ALACRITY: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying," *Transactions on Large Scale Data and Knowledge Centered Systems (TLDKS)*, vol. 8220, pp. 95–114, 2013.
- [8] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A cost-intelligent application-specific data layout scheme for parallel file systems," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 37–48. [Online]. Available: <http://doi.acm.org/10.1145/1996130.1996138>
- [9] X. Zhang and S. Jiang, "InterferenceRemoval: Removing interference of disk access for mpi programs through data replication," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 223–232. [Online]. Available: <http://doi.acm.org/10.1145/1810085.1810116>
- [10] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "A segment-level adaptive data layout scheme for improved load balance in parallel file systems," in *Cluster, Cloud and Grid Computing (CCGrid), IEEE/ACM International Symposium on*, 2011, pp. 414–423.
- [11] J. S. Bucy, J. Schindler, S. Schlosser, G. Ganger, and Contributors, "The disksim simulation environment version 4.0 reference manual," Carnegie Mellon University Parallel Data Lab, Tech. Rep. CMU-PDL-08-101, 2008.
- [12] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12. [Online]. Available: <http://doi.acm.org/10.1145/1654059.1654081>
- [13] D. Goodell, S. J. Kim, R. Latham, M. Kandemir, and R. Ross, "An evolutionary path to object storage access," in *Proceedings of the Seventh Workshop on Parallel Data Storage*, ser. PDSW '12, 2012.
- [14] R. Thakur, R. Ross, E. Lust, and W. Gropp, "Users guide for ROMIO: A high-performance, portable MPI-IO implementation," Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-234, 2004.
- [15] R. Thakur, W. Gropp, and E. Lusk, "An abstract-device interface for implementing portable parallel-I/O interfaces," in *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 180–187. [Online]. Available: <http://dl.acm.org/citation.cfm?id=795667.796725>
- [16] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *IEEE International Conference on Cluster Computing*, ser. Cluster'10, 2009, pp. 1–10.
- [17] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOC)*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2027066.2027068>
- [18] J.-I. Gailly and M. Adler, "Zlib general purpose compression library," <http://zlib.net/>, Jan. 2012.
- [19] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008*. IEEE, 2008, pp. 1–12.
- [20] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting application-specific parallel I/O optimization using IOSIG," in *Cluster, Cloud and Grid Computing (CCGrid)*, 2012, pp. 196–203.
- [21] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed. Morgan Kaufmann, 1999.
- [22] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys*, vol. 38, no. 2, 2006. [Online]. Available: <http://doi.acm.org/10.1145/1132956.1132959>
- [23] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova, "Scalable in situ scientific data encoding for analytical query processing," in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: <http://doi.acm.org/10.1145/2462902.2465527>
- [24] "Parallel I/O benchmarking consortium," <http://www.mcs.anl.gov/research/projects/pio-benchmark/>.
- [25] "Interleaved or random (IOR) parallel filesystem I/O benchmark." [Online]. Available: <https://github.com/chaos/ior>
- [26] F. Shorter, "Design and analysis of a performance evaluation standard for parallel file systems," Master's thesis, Clemson University, 2003.
- [27] A. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp, "Non-contiguous I/O through PVFS," in *IEEE International Conference on Cluster Computing*, 2002, pp. 405–414.
- [28] M. Vilayannur, S. Lang, R. Ross, R. Klundt, and L. Ward, "Extending the POSIX I/O interface: A parallel file system perspective," Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-302, 2008.
- [29] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: <http://dx.doi.org/10.1109/MSST.2010.5496972>
- [30] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: <http://doi.acm.org/10.1145/945445.945450>
- [31] M. K. McKusick and S. Quinlan, "GFS: Evolution on fast-forward," *Queue*, vol. 7, no. 7, pp. 10:10–10:20, Aug. 2009. [Online]. Available: <http://doi.acm.org/10.1145/1594204.1594206>
- [32] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1298455.1298485>
- [33] 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," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 2:1–2:17. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1364813.1364815>
- [34] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the duality of data-intensive file system design: Reconciling HDFS and PVFS," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'11. New York, NY, USA: ACM, 2011, pp. 67:1–67:12. [Online]. Available: <http://doi.acm.org/10.1145/2063384.2063474>
- [35] M. Bhadkankar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for self-optimizing storage systems," in *Proceedings of the 7th Conference on File and Storage Technologies*, ser. FAST '09. Berkeley, CA, USA: USENIX Association, 2009, pp. 183–196. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1525908.1525922>
- [36] H. Huang, W. Hung, and K. G. Shin, "Fs2: Dynamic data replication in free disk space for improving disk performance and energy consumption," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 263–276. [Online]. Available: <http://doi.acm.org/10.1145/1095810.1095836>
- [37] S. Narayanan, U. Catalyurek, T. Kurc, V. S. Kumar, and J. Saltz, "A runtime framework for partial replication and its application for on-demand data exploration," in *High Performance Computing Symposium, SCS Spring Simulation Multiconference*, ser. HPC '05, 2005.
- [38] L. Weng, U. Catalyurek, T. Kurc, G. Agrawal, and J. Saltz, "Servicing range queries on multidimensional datasets with partial replicas," in *IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGrid '05, vol. 2. IEEE, 2005, pp. 726–733.
- [39] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in MonetDB," *Communications of the ACM*, vol. 51, pp. 77–85, December 2008. [Online]. Available: <http://doi.acm.org/10.1145/1409360.1409380>

- [40] S. Idreos, M. Kersten, and S. Manegold, "Database cracking," in *Proceedings of the 3rd International Conference on Innovative Data Systems Research*, ser. CIDR'07, 2007.
- [41] S. Idreos, "Database cracking: Towards auto-tuning database kernels," Ph.D. dissertation, University of Amsterdam, 2010.
- [42] T.-S. Chua, J. Li, B.-C. Ooi, and K.-L. Tan, "Disk striping strategies for large video-on-demand servers," in *Proceedings of the Fourth ACM International Conference on Multimedia*, ser. MULTIMEDIA '96. New York, NY, USA: ACM, 1996, pp. 297–306. [Online]. Available: <http://doi.acm.org/10.1145/244130.244231>
- [43] J. H. Korst, "Random duplicated assignment: An alternative to striping in video servers," in *Proceedings of the Fifth ACM International Conference on Multimedia*, ser. MULTIMEDIA '97. ACM, 1997, pp. 219–226.
- [44] H. Lamehamed, B. Szymanski, Z. Shentu, and E. Deelman, "Data replication strategies in grid environments," in *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, ser. ICA3PP'02. IEEE, 2002, pp. 378–383.
- [45] H. Stockinger, A. Samar, K. Holtman, B. Allcock, I. Foster, and B. Tierney, "File and object replication in data grids," *Cluster Computing*, vol. 5, no. 3, pp. 305–314, 2002.
- [46] S. W. Son, R. Latham, R. Ross, and R. Thakur, "Reliable MPI-IO through layout-aware replication," in *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O*, ser. SNAPI '11, 2011.
- [47] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur, "Pattern-direct and layout-aware replication scheme for parallel i/o systems," in *IEEE International Symposium on Parallel and Distributed Computing*, ser. IPDPS'13, 2013, pp. 345–356.
- [48] A. Chan, W. Gropp, and E. Lusk, "User's guide for MPE: Extensions for MPI programs," Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-ANL-98/xx, 2003.
- [49] J. S. Vetter and M. O. McCracken, "Statistical scalability analysis of communication operations in distributed applications," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, ser. PPOPP '01. New York, NY, USA: ACM, 2001, pp. 123–132. [Online]. Available: <http://doi.acm.org/10.1145/379539.379590>
- [50] S. J. Kim, Y. Zhang, S. W. Son, R. Prabhakar, M. Kandemir, C. Patrick, W.-k. Liao, and A. Choudhary, "Automated tracing of I/O stack," in *Proceedings of the 17th European MPI Users Group Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 72–81. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1894122.1894132>
- [51] S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary, "IOPin: Runtime profiling of parallel I/O in HPC systems," in *7th Parallel Data Storage Workshop*, ser. PDSW'12, 2012.
- [52] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: <http://doi.acm.org/10.1145/1065010.1065034>
- [53] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 696–710, Aug. 2009. [Online]. Available: <http://dx.doi.org/10.1016/j.jpdc.2008.09.001>
- [54] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz, "Preserving time in large-scale communication traces," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 46–55. [Online]. Available: <http://doi.acm.org/10.1145/1375527.1375537>
- [55] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable I/O tracing and analysis," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, ser. PDSW '09. New York, NY, USA: ACM, 2009, pp. 26–31. [Online]. Available: <http://doi.acm.org/10.1145/1713072.1713080>
- [56] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Probabilistic communication and I/O tracing with deterministic replay at scale," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 196–205. [Online]. Available: <http://dx.doi.org/10.1109/ICPP.2011.50>
- [57] S. Godard, "Sysstat utilities home page," <http://sebastien.godard.pagesperso-orange.fr/index.html>.
- [58] S. Dayal, "Characterizing HEC storage systems at rest," Carnegie Mellon University Parallel Data Laboratory, Tech. Rep. CMU-PDL-09-109.
- [59] T. M. Madhyastha and D. A. Reed, "Learning to classify parallel input/output access patterns," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 8, pp. 802–813, Aug. 2002. [Online]. Available: <http://dx.doi.org/10.1109/TPDS.2002.1028437>
- [60] J. Oly and D. A. Reed, "Markov model prediction of I/O requests for scientific applications," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, pp. 147–155. [Online]. Available: <http://doi.acm.org/10.1145/514191.514214>
- [61] N. Tran and D. A. Reed, "Automatic ARIMA time series modeling for adaptive I/O prefetching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 4, pp. 362–377, Apr. 2004. [Online]. Available: <http://dx.doi.org/10.1109/TPDS.2004.1271185>
- [62] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "Discovering structure in unstructured I/O," in *Proceedings of the Seventh Workshop on Parallel Data Storage*, ser. PDSW '12, 2012.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.