Optimizing I/O Performance of HPC Applications with Autotuning

BABAK BEHZAD, University of Illinois at Urbana-Champaign
SURENDRA BYNA and PRABHAT, Lawrence Berkeley National Laboratory
MARC SNIR, Argonne National Laboratory and University of Illinois at Urbana-Champaign

Parallel Input output is an essential component of modern high-performance computing (HPC). Obtaining good I/O performance for a broad range of applications on diverse HPC platforms is a major challenge, in part, because of complex inter dependencies between I/O middleware and hardware. The parallel file system and I/O middleware layers all offer optimization parameters that can, in theory, result in better I/O performance. Unfortunately, the right combination of parameters is highly dependent on the application, HPC platform, problem size, and concurrency. Scientific application developers do not have the time or expertise to take on the substantial burden of identifying good parameters for each problem configuration. They resort to using system defaults, a choice that frequently results in poor I/O performance. We expect this problem to be compounded on exascale-class machines, which will likely have a deeper software stack with hierarchically arranged hardware resources.

We present as a solution to this problem an autotuning system for optimizing I/O performance, I/O performance modeling, I/O tuning, and I/O patterns. We demonstrate the value of this framework across several HPC platforms and applications at scale.

CCS Concepts: • **Social and professional topics** → *File systems management*;

Additional Key Words and Phrases: HPC, storage, I/O, autotuning, performance optimization, parallel file systems

ACM Reference format:
Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2019. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Trans. Parallel Comput.* 5, 4, Article 15 (March 2019), 27 pages.
https://doi.org/10.1145/3309205

1 INTRODUCTION

High-performance computing (HPC) applications are constantly moving toward simulating scientific phenomena at finer granularities and massive scales by leveraging advances in parallel

This work is supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, the Texas Advanced Computing Center, and the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. It was partly supported by NSF grant 0938064.
Authors' addresses: B. Behzad, 66 E 40th AveSan Mateo, CA 94403; email: babakbehzad@gmail.com; S. Byna, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Mail Stop 50B-3238 Berkeley, CA 94720 email: sbyna@lbl.gov; Prabhat, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Mail Stop 59R4010A, Berkeley, CA 94720; email: prabhat@lbl.gov; M. Snir, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Urbana, IL 61801; email: snir@illinois.edu.

2329-4949/2019/03-ART15 $15.00
https://doi.org/10.1145/3309205

processing hardware. Finer granularities mean a larger amount of data, and this has caused the growth of data to be at an unprecedented rate. Such rapid growth of data in size and in complexity requires efficient techniques to manage the data on file systems. However, scalability of applications is often limited by poorly performing parallel I/O. Ensuring fast and efficient parallel I/O is critical for many HPC applications.

I/O can be a significant bottleneck on HPC application performance. The need to increase checkpoint frequency and the increasing emphasis on big data analytics increases the importance of I/O. On the other hand, parallel I/O systems are complex: I/O is often done at the application level by using a high-level library, such as Hierarchical Data Format, version 5 (HDF5) (Folk et al. 1999); HDF5 is implemented atop Message Passing Interface (MPI)-IO (Corbett et al. 1996), which, in turn, performs Portable Operating System Interface (POSIX) I/O calls against a parallel file system, such as Lustre (Schwan 2003). Each of these subsystems has multiple configuration parameters, and performance can be sensitive to their settings.

Configuring these parameters to obtain the best possible I/O performance depends on diverse factors, such as the I/O application, storage hardware, problem size, and number of processors. HPC application developers, typically experts in their scientific domains, do not have the time or expertise to explore the intricacies of I/O systems. They often resort to using default I/O parameter settings that can result in poor performance and inefficient use of available I/O bandwidth. As the complexity and concurrency of future HPC systems grow, we expect this to be a major obstacle to achieving high-performance I/O.

Application developers should be able to achieve good I/O performance without becoming experts on the tunable parameters for every file system and I/O middleware layer they encounter. Scientists want to write their application once and obtain reasonable performance across multiple systems—they want *I/O performance portability across platforms*. From an I/O research-centric viewpoint, a considerable amount of effort is spent optimizing individual applications for specific platforms. While the benefits are definitely worthwhile for specific application codes, and some optimizations carry over to other applications and middleware layers, ideally if a single optimization framework would be capable of generalizing across multiple applications.

To use HPC machines and human resources effectively, we must design systems that can *hide the complexity of the I/O stack* from scientific application developers without penalizing performance. Our vision is to develop a system that will allow application developers to issue I/O calls without modification and rely on an intelligent runtime system to transparently determine and execute an I/O strategy that takes all the levels of the I/O stack into account.

The main contributions of this article are as follows.

- —Design and implemention of an autotuning system that hides the complexity of tuning the parallel I/O stack
- —Demonstration of performance portability of the autotuning system across diverse HPC platforms
- —Development of an approach to construct automatically an I/O performance model
- —Use of the model thus constructed to reduce the search space for good I/O configurations
- —Demonstration of how scientific I/O kernels with different write patterns and various problem sizes can benefit from the autotuning framework
- —Usage of the I/O patterns as a key to an intelligent runtime system for tuning I/O

The rest of this article is organized as follows. Section 2 presents the background. In Sections 3 and 4, respectively, we introduce the HPC platforms and application I/O kernels we experimented with. Section 5 describes H5Tuner library as background needed for the following sections. Section 6 then describes the general autotuning framework we propose for solving HPC I/O

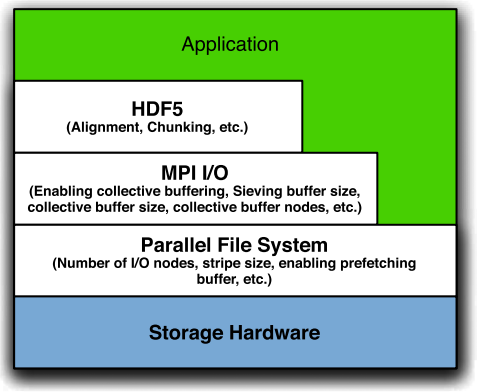


Fig. 1. Typical parallel I/O stack and various tunable parameters.

problems. Section 7 presents our genetic algorithm (GA) autotuning framework, which consistently demonstrates I/O write speeds between 2x and 100x. In Section 8 we then reduce the overhead of the GA approach significantly by using empirical models of the I/O performance. Section 9 focuses on I/O patterns as a key to reusing the I/O performance models of HPC applications. We describe an intelligent runtime system, capable of extracting I/O patterns from arbitrary applications and consulting the performance database to propose an improved I/O strategy. All the experimental results are shown in detail in Section 10. Section 11 discusses some issues and limitations of this work along with a comparison of the proposed solutions. Section 12 describes the research projects related to each of the components of this work. We conclude in Section 13 with some possible future research directions.

2 BACKGROUND

A parallel I/O subsystem typically consists of various layers of middleware libraries and hardware. The most common parallel I/O stack in current HPC machines has high-level I/O libraries and file formats (e.g., HDF5, NetCDF, and ADIOS), I/O middleware (e.g., MPI-IO and POSIX), parallel file systems (e.g., Lustre, GPFS, and PVFS), and storage and I/O hardware. When parallel applications perform I/O operations, the data moves from individual processors to the storage hardware through the multiple layers of the stack.

Figure 1 shows a contemporary parallel I/O software stack with HDF5 (HDF5 2010) as the high-level I/O library, MPI-IO as the middleware layer, and a parallel file system. We chose HDF5 as the high-level library in this study because it is the most used parallel I/O library at supercomputing facilities (Pearah and Soumagne 2016). While each layer of the stack exposes tunable parameters for improving performance, little guidance is provided to application developers on how these parameters interact with one another and affect overall I/O performance.

Additionally, to achieve good I/O performance, each of the layers offers optimization strategies. For instance, MPI-IO provides two modes of writing data to disks: independent I/O and collective I/O (Thakur et al. 1999). With independent I/O, each MPI process writes the data to storage independently of other processes of the application. In collective I/O mode, the data is collected at a few aggregator processes, and the aggregators write the data to storage. The collective I/O mode is preferable when the number of MPI processes is large, because too many requests to the file system degrade I/O performance.

A typical implementation of a collective I/O write operation includes two phases: the data collection phase at aggregators and the I/O phase (del Rosario et al. 1993). Each MPI process first

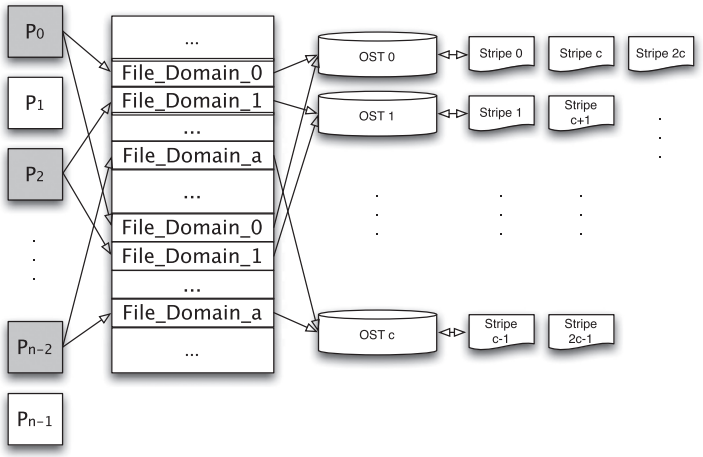


Fig. 2. File domain assignment in a configuration with "*a*" I/O aggregators (processes shaded in gray).

analyzes its request to the file and calculates the *start offset* and *end offset*. These two variables identify the segment of the file accessed by the processor. After calculating these variables, each process sends its values to all the other processes. The aggregators then compute the partitions, called *file domains*, of the file they are responsible for reading/writing. In ROMIO (Thakur et al. 1999), which is the basis for many MPI-IO implementations, the aggregators split the range of the file being updated equally in a block-cyclic distribution. Figure 2 shows an example file domain assignment in a configuration with "*a*" I/O aggregators, each of them in charge of one file domain.

Parallel file systems, such as Lustre, typically use multiple storage servers to parallelize I/O operations. Lustre uses object storage targets (OSTs) for storing chunks of data. Lustre allows users or applications to control the number of OSTs, called the *stripe count*, and the size of contiguous chunks of data, called the *stripe size*, for storing the data. The MPI-IO aggregators write blocks of size equal to the stripe size in a round-robin fashion (Knaak and Oswald 2009). Several algorithms have been proposed for selecting the aggregators and writing data to stripes (Liao and Choudhary 2008). Among these, the CB alignment algorithm 2 has been developed by the Cray MPT library (Cray 2016). Figure 2 illustrates the CB algorithm 2, where the block size used to partition the file into domains is equal to the stripe size, consequently written to OSTs in a round-robin fashion. In this algorithm, Cray's MPT sets the collective buffering buffer size equal to the Lustre stripe size. Therefore, the main I/O parameters to tune are the Lustre stripe count, Lustre stripe size, and MPI-IO number of collective buffering nodes (aggregators).

Note that from now on, we define *write time* to be the time elapsed from calling a write operation in a higher-level library until the function is done, consisting of all the communication and I/O time needed for this operation.

3 HPC PLATFORMS

The experiments in this article are conducted on the following HPC platforms.

—**Edison:** Edison is a supercomputer at the National Energy Research Scientific Computing Center (NERSC). It is a Cray XC30 system consisting of 5,576 24-core nodes with 64GB of memory per node. It has the Cray Aries with Dragonfly topology and three Lustre file systems with aggregate bandwidth of 168GB/s. We only used a scratch2 file system in these experiments with a maximum of 96 OSTs and 48GB/s peak I/O bandwidth. Cray's MPI library v7.0.4, HDF5 v1.8.11, and H5Part v1.6.6 were used on Edison.

Table 1. Details of Various HPC Systems Used in This Article

HPC System	Architecture	Node Hardware	File-system	Storage Hardware	Peak I/O BW
NERSC/Edison	Cray XC30	Intel Ivy Bridge processors, 24 cores per node, 64GB memory	Lustre	96 OSTs, 24 OSSs	48GB/s (Byna et al. 2012)
NERSC/Hopper	Cray XE6	AMD Opteron processors, 24 cores per node, 32GB memory	Lustre	156 OSTs, 26 OSSs	35GB/s (Byna et al. 2012)
TACC/Stampede	Dell PowerEdge	Xeon E5-2680 processors, 16 cores per node, 32GB memory	Lustre	160 OSTs, 58 OSSs	159GB/s (Schulz 2013)
ALCF/Intrepid	IBM BG/P	PowerPC 450 processors, 4 cores per node, 2GB memory	GPFS	640 IO nodes, 128 file servers	47GB/s (write) (Lang et al. 2009)

—**Hopper:** Hopper is another supercomputing system located at NERSC. It is a Cray XE6 system containing 6,384 24-core nodes with 32GB of memory per node. It employs the Gemini interconnect with a 3D torus topology. We used a Lustre file system with 156 OSTs and a peak bandwidth of about 35GB/s for storing data. We used Cray's MPI library v6.0.1, HDF5 v1.8.11, and H5Part v1.6.6 for compiling the I/O kernels.

—**Stampede:** Stampede is a Dell PowerEdge C8220 cluster at the Texas Advanced Computing Center. It has 6,400 16-core nodes with 32GB of memory per node. It uses Mellanox Fourteen Data Rate (FDR) InfiniBand technology with a two-level fat-tree topology. Stampede's Lustre file system with 160 OSTs (in our experiments, for consistent comparisons, we use 156 OSTs as the maximum stripe count for Stampede as well) has shown a peak of 159GB/s I/O bandwidth.

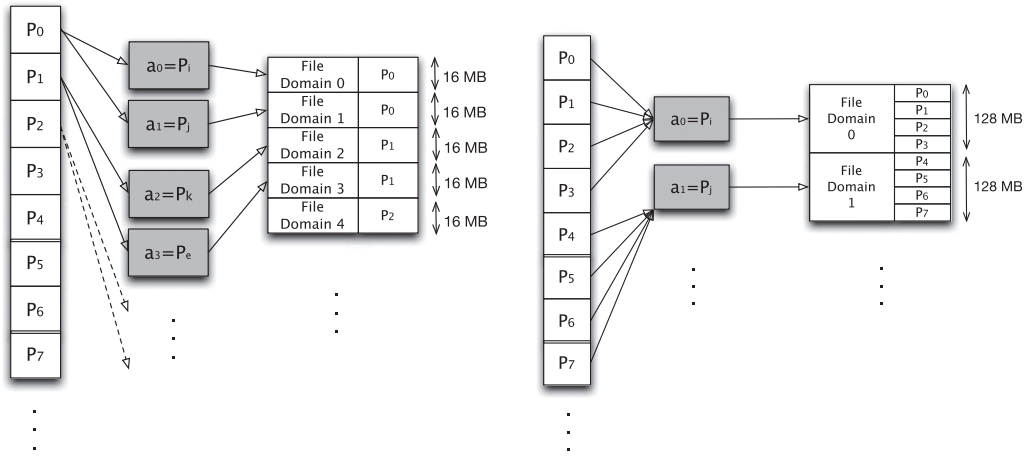
—**Intrepid:** Intrepid, an IBM BlueGene/P (BG/P) system at the Argonne Leadership Computing Facility (ALCF), is a 40-rack 0.5 petaflop system. Each rack contains 1,024 nodes with 850MHz quad-core processors and 2GB RAM per node. It is also equipped with 640 I/O nodes and 128 file servers with more than 7.6PB of storage. Note that the I/O stack is different on Intrepid from that on Edison, Hopper, and Stampede. On Intrepid, the parallel file system is GPFS, whereas Edison, Hopper, and Stampede use the Lustre file system. GPFS uses dedicated I/O nodes (IONs) to act as proxies between the compute nodes and storage nodes. Each ION serves 64 4-core nodes, and the collection of the ION and compute nodes is called a *pset*.

Table 1 lists details of these HPC systems. We note that the number and type of I/O resources vary across these platforms.

4 APPLICATION I/O KERNELS

We chose one I/O benchmark and four parallel I/O kernels to evaluate our autotuning framework: Interleaved or Random (IOR), vector particle-in-cell (VPIC)-IO, VORPAL-IO, global cloud resolving model (GCRM)-IO, and FLASH-IO. The kernels are derived from the I/O calls of four applications: VPIC (Bowers et al. 2008), VORPAL (Nieter and Cary 2004), GCRM (Randall et al. 2003), and FLASH, respectively. These I/O kernels represent four distinct I/O write motifs with different data sizes.

—**IOR—I/O benchmark:** IOR (LLNL 2015) is an I/O benchmark developed at Lawrence Livermore National Laboratory (LLNL) for the procurement of the Accelerated Strategic Computing Initiative (ASCI) Purple. Since it is highly configurable and contains different I/O interfaces, it serves as one of the main HPC I/O benchmarks.



(a) Lustre stripe size = 16 MB (b) Lustre stripe size = 128 MB

Fig. 3. Partitioning of file domains and processors among aggregators in VPIC-IO.

—**VPIC-IO—plasma physics:** VPIC is a highly optimized and scalable particle physics simulation developed by Los Alamos National Laboratory (Bowers et al. 2008). VPIC-IO uses H5Part (Bethel et al. 2007) API to create a file, write eight variables, and close the file. H5Part provides a simple veneer API for issuing HDF5 calls corresponding to a time-varying, multivariate particle data model. VPIC-IO extracts all the H5Part function calls of the VPIC code to form the VPIC-IO kernel. The particle data written in the kernel is random data of floating-point data type. The I/O motif of VPIC-IO is a 1D particle of a given number of particles, and each particle has eight variables. The kernel writes 8M particles per MPI process for all experiments reported in this article.
Figure 3 shows the partitioning of VPIC-IO file domains for two Lustre stripe size settings. In VPIC-IO with a 1D-array pattern, each processor writes 4 bytes per particle for each variable (since all the variables are of 32-bit floating-point type) of this 1D dataset into the file in the order of its rank. All our experiments are conducted with 8 million particles, leading to write sizes of 32MB by each processor. Therefore, for each of the collective write calls, process 0 writes to file offset 0MB to 32MB, process 1 writes to file offset 32MB to 64MB, and so forth. The left figure shows the partitioning for a stripe size of 16MB and the right figure shows it for a stripe size of 128MB. The notation P_i refers to the MPI processes, while a_i refers to aggregators.

—**VORPAL-IO—accelerator modeling:** This I/O kernel is extracted from VORPAL, a computational plasma framework application simulating the dynamics of electromagnetic systems, plasmas, and rarefied as well as dense gases, developed by TechX (Nieter and Cary 2004). This benchmark uses H5Block to write nonuniform chunks of 3D data per processor. The kernel takes 3D block dimensions (x, y, and z) and the number of components as input. In our experiments, we used 3D blocks of $100 \times 100 \times 60$ with different numbers of processors. The data is written for 20 time steps.
VORPAL-IO leverages the H5Block library (Bethel et al. 2007), which uses the HDF5 library to handle block-structured data. VORPAL-IO partitions a 3D grid of points into a 3D grid of processes. Each process writes the sub block of points in its partition. For example, in a 128-process run with a block of size $300 \times 100 \times 60$ and a decomposition of (8, 4, 4), the size of the total block is $2400 \times 400 \times 240$. This kernel is also configured to run in a weak-scaling mode.

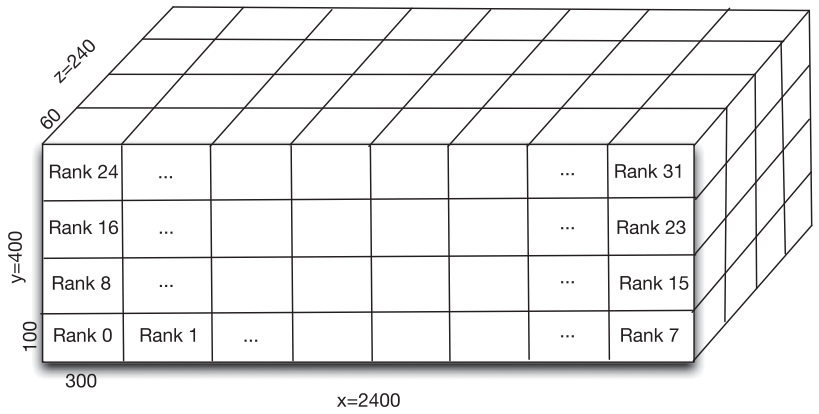


Fig. 4. 3D block structure of VORPAL-IO datasets in HDF5.

In terms of I/O pattern, VORPAL-IO is more complex than VPIC-IO. It writes 3D block-structured grids using 3D HDF5 datasets. We have configured this I/O kernel so that each block is of size $(300 \times 100 \times 60)$. In contrast to VPIC-IO, VORPAL-IO variables are of type double and of size 8 bytes. Therefore, the size of each block is 13MB. The method to scale VORPAL-IO is also different from that of VPIC-IO. VORPAL-IO has a configurable nonuniform grid decomposition scheme, in which the user can specify out of the number of processors how each of these three dimensions gets scaled. For example, for a 128-core run of VORPAL-IO, if the user chooses the block decomposition as $(8, 4, 4)$, the total x-dimension of the grid will be $2,400$ $(= 300 \times 8)$, and the y-dimension and z-dimensions will be 400 $(= 100 \times 4)$ and 240 $(= 60 \times 4)$, respectively. This grid and the way the blocks are assigned to each process rank are shown in Figure 4.

—**GCRM-IO—global atmospheric model:** This I/O kernel simulates I/O for a global atmospheric circulation model (GCRM), simulating the circulations associated with large convective clouds. This I/O benchmark also uses H5Part to perform I/O operations. The kernel performs all the GCRM I/O operations with random data. The I/O pattern of GCRM-IO corresponds to a semi-structured geodesic mesh, where the grid resolution and subdomain resolution are specified as input. In our tests, we used varying grid resolutions at different concurrencies. By default, this benchmark uses 25 vertical levels and 1 iteration.

—**FLASH-IO—high-energy density model:** The FLASH I/O benchmark routine mimics the I/O of the FLASH parallel HDF5 write operations. It has the data structures in a FLASH application and writes a checkpoint file, a plotfile with centered data, and a plotfile with corner data. At 512 cores, FLASH-IO creates a 122GB checkpoint file, 11GB centered data plotfile, and 12GB corner plotfile. At 4,096 cores, it creates a 973GB checkpoint file, 82GB centered data plotfile, and 92GB corner plotfile.

5 H5TUNER: SETTING I/O PARAMETERS AT RUNTIME

The H5Tuner component is an autonomous parallel I/O parameter injector for scientific applications with minimal user involvement, allowing parameters to be altered without requiring source code modifications and recompilation of the application. The H5Tuner dynamic library is able to set the parameters of different levels of the I/O stack—namely, the HDF5, MPI-IO, and parallel file system levels in our implementation. Assuming all the I/O optimization parameters for different levels of the stack are in a configuration file, H5Tuner first reads the values of the I/O configuration.

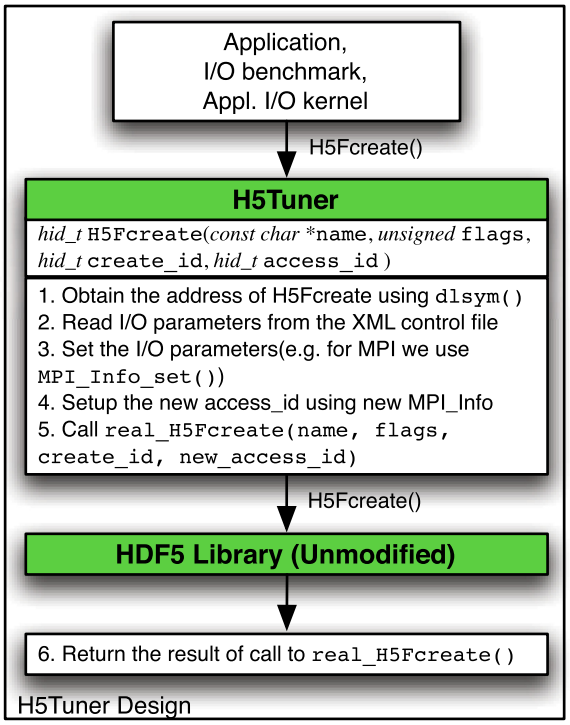


Fig. 5. Design of H5Tuner component as a dynamic library that intercepts HDF5 functions to tune I/O parameters.

When the HDF5 calls appear in the code during the execution of a benchmark or application, the H5Tuner library intercepts the HDF5 *initialization* function calls via dynamic linking. The library reroutes the intercepted HDF5 calls to a new library, where the parameters from the configuration are set and then the original HDF5 function is called by using the dynamic library package functions. This approach has the added benefit of being completely transparent to the user; the function calls remain exactly the same, and all alterations are made without change to the source code. We show an example in Figure 5, where H5Tuner intercepts an `H5FCreate()` function call that creates an HDF5 file, applies various I/O parameters, and calls the original `H5FCreate()` function call.

6 GENERAL AUTOTUNING FRAMEWORK

Figure 6 shows a general autotuning system that uses H5Tuner for applying a configuration proposed by the autotuning framework. The autotuning framework extracts the I/O kernel of an application using tracing tools such as I/O Tracer (Behzad et al. 2014b) or Skel (Logan et al. 2011) and runs the kernel with a preselected training set of tunable parameters. It also takes the I/O parameter space with its all-possible configurations as input and for each I/O kernel generates a set of k configurations that perform best. Each of these configurations is presented to H5Tuner in Extensible Markup Language (XML) format. The parameter space contains possible values for I/O tuning parameters at each layer of the I/O stack, and the configuration file contains the parameter settings that will be used for a given run. H5Tuner reads the configuration file and dynamically links to HDF5 calls of an application or I/O benchmark. After running the executable, the parameter settings and I/O performance results can be fed back to the autotuning framework in order to influence the contents of the next configuration file.

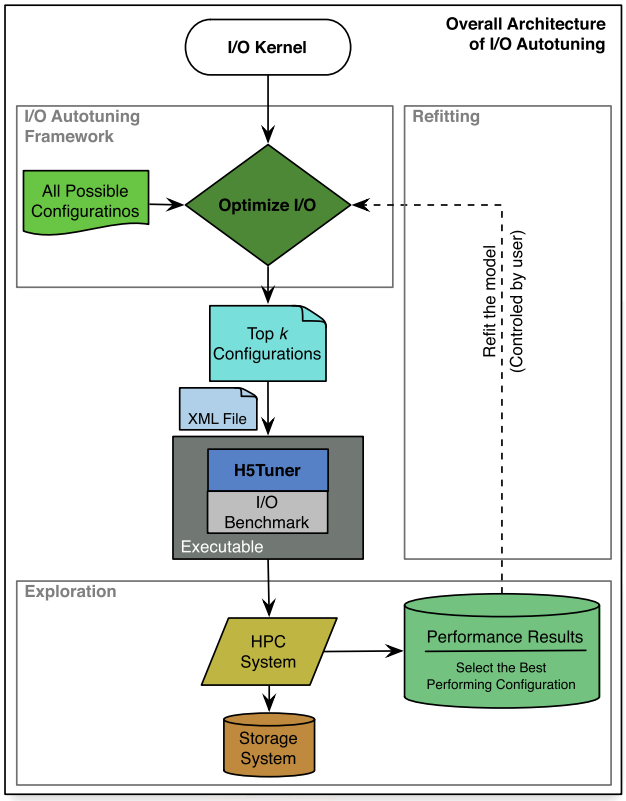


Fig. 6. Overall architecture of I/O autotuning.

Because of the large size of the parameter space and possibly long execution time of a trial run, finding optimal parameter sets for writing data of a given size is a nontrivial task. Depending on the granularity with which the parameter values are set, the size of the parameter space can grow exponentially and become unmanageably large for a brute-force and enumerative optimization approach.

For selecting tunable parameters, a naïve strategy is to execute an application or a representative I/O kernel of the application using all possible combinations of tunable parameters for all layers of the I/O stack. This is an extremely time- and resource-consuming approach, since a typical parameter space has many thousands of combinations. For instance, if we enable chunking on HDF5 with a Lustre platform, a simple set of parameters can lead to 336,000 possible configurations.

Instead of relying on the simplest approach, manual tweaking, this article discusses two different approaches for traversing the search space in a reasonable amount of time:

(1) **Genetic Algorithms—H5Evolve:** Adaptive heuristic search approaches such as genetic evolution algorithms and simulated annealing can traverse the search space in a reasonable amount of time. In H5Evolve, we explore genetic algorithms for sampling the search space.
(2) **I/O Performance Modeling:** Using a set of real measurements from running an I/O kernel as a training set, one can generate a regression model. We use such a model to predict the top-20 tunable parameter values that give efficient I/O performance and rerun the I/O kernel to select the best set of parameters under the current conditions.

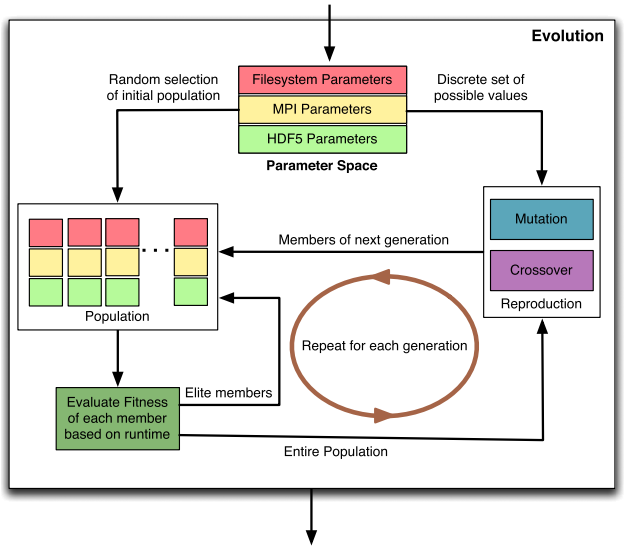


Fig. 7. Overall architecture of the H5Evolve framework.

Once we show the effectiveness of I/O tuning at multiple layers of tunable parameters using genetic algorithms and improve the configuration search process by developing an empirical performance prediction model for a selection of I/O kernels derived from real scientific simulations, one last challenge remains: tuning an arbitrary I/O phase in a simulation. For instance, when a simulation needs to perform a large write operation, an intelligent runtime system is required to identify the characteristics of the write operation, find optimal tunable parameters, and apply them at runtime without the need to stop the simulation for recompiling the simulation code with the optimal configurations. We present a solution to this problem in Section 9.

7 GENETIC ALGORITHMS—H5EVOLVE

The main challenge in designing and implementing an I/O autotuning system is selecting an effective set of tunable parameters at all layers of the stack. A reasonable approach for traversing the huge possible space for configurations is to search the parameter space with a small number of tests. Toward this goal, we developed H5Evolve to search the I/O parameter space using a genetic algorithm. H5Evolve samples the parameter space by testing a set of parameter combinations and then, based on the I/O performance, adjusts the combination of tunable parameters for further testing. As H5Evolve passes through multiple generations, better parameter combinations (i.e., sets of tuned parameters with high I/O performance) emerge.

A genetic algorithm (GA) is a metaheuristic for approaching an optimization problem, particularly one that is ill-suited for traditional exact or approximation methods. A GA is meant to emulate the natural process of evolution, working with a "population" of potential solutions through successive "generations" (iterations) as they "reproduce" (intermingle portions between two members of the population) and are subject to "mutations" (random changes to portions of the solution). A GA is expected, although it cannot necessarily be shown, to converge to an optimal or near-optimal solution, as strong solutions beget stronger children, while the random mutations offer a sampling of the remainder of the space.

As Figure 7 depicts, the workflow of H5Evolve is as follows. For a given benchmark at a specific concurrency and problem size, H5Evolve runs the genetic algorithm. H5Evolve takes a predefined

parameter space that contains possible values for the I/O tuning parameters at each layer of the I/O stack. The evolution process starts with randomly selected initial population. H5Evolve generates an XML file containing the selected I/O parameters (an I/O configuration) that H5Tuner injects into the benchmark. In all our experiments, the H5Evolve GA uses a population size of 15; this size is a configurable option. Starting with an initial group of configuration sets, the genetic algorithm passes through successive generations. H5Evolve uses the runtime as the fitness evaluation for a given I/O configuration. After each generation has completed, H5Evolve evaluates the fitness of the population and considers the fastest I/O configurations (i.e., the "elite members") for inclusion in the next generation. Additionally, the entire current population undergoes a series of mutations and crossovers to populate the other member sets in the population of the next generation. This process repeats for each generation. In our experiments, we set the number of generations to 40, meaning that H5Evolve runs a maximum of 600 executions of a given benchmark. We used a mutation rate of 15%, meaning that 15% of the population undergoes mutation at each generation. After H5Evolve finishes sampling the search space, the best-performing I/O configuration is stored as the tuned parameter set.

8 I/O PERFORMANCE MODELING

Recent studies, in addition to our work described in the previous section, have explored autotuning using genetic algorithms to traverse the search space systematically (Behzad et al. 2013). The GA approach initializes this traversal with random sets of parameters and produces new generations of parameter sets by applying mutation and crossover operations. The GA eventually determines parameter values that give near-optimal I/O performance. This approach is time-consuming, as the number of experiments required to converge might be prohibitively large. Another limitation is that parameter values are specific to each application and input size.

In this section, we present a statistical approach for automatic generation of an empirical performance prediction model that is used for pruning the search space significantly. We use a statistical approach to train an empirical base model that can be used to prune the search space. The base model then can be tuned further by running on the reduced sample space in order to capture the dynamic runtime conditions of a system. The advantages of our proposed method include fast reduction of the search space compared to using a GA approach and consideration of the dynamic conditions of a parallel I/O subsystem.

Figure 8 shows a high-level workflow of our proposed dynamic model-driven I/O tuning process. We define the training set based on the parameters for different levels of the I/O stack and for multiple problem sizes. Using the measured I/O performance of the kernel, we develop an empirical performance model.

Using this constructed performance model, we predict I/O performance for an exhaustive set of all combinations of tuning parameters. We then select the best-performing tuning parameter sets by sorting the predicted performance for further exploration. The number of best parameter sets for exploring the current conditions of an HPC system is a configurable option. Based on the measured I/O performance of the top-k parameter sets, we select the set that has the best I/O performance as the tuned I/O configuration for the I/O kernel for a given scale. Optionally, one can fine-tune the I/O model further by evaluating the performance results of the top k configurations iteratively. In this article, we select the best-performing I/O parameter set from running the top $k = 20$ configurations.

We examined nonlinear regression models in the context of modeling I/O write times for a given application. The main I/O parameters on a Lustre file system are Lustre stripe settings (e.g., stripe count and stripe size) and MPI-IO collective buffering settings (e.g., number of collective buffering nodes and collective buffering size) at the middleware layer.

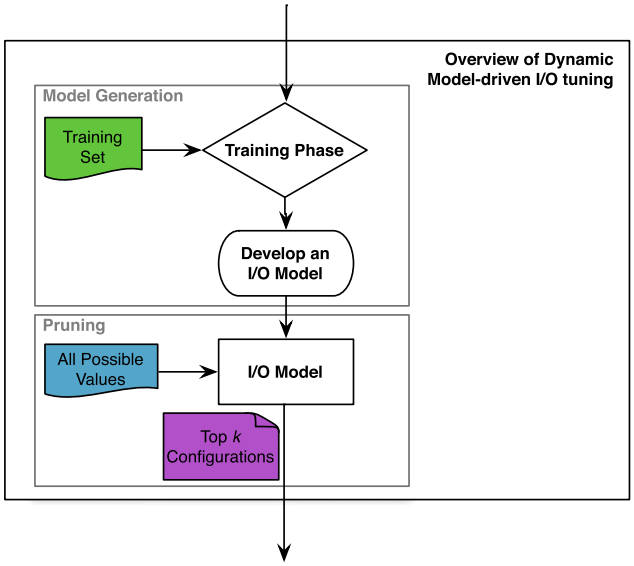


Fig. 8. Overview of our proposed dynamic model-driven I/O tuning process.

Table 2. Training Set for the Performance Model

# of Cores	Training Set Size
512	336
1,024	180
2,048	96

We conducted experiments for all three applications and different file sizes on all the platforms. The training set for each of the applications and their file sizes are shown in Table 2. The size of the training set is decreased as the core counts and file sizes increase due to the increase in the required resources.

The selection of the training set is automatic, with simple heuristics of limits on the allowable value ranges in order to cover the parameter space well. For example, the maximum number of aggregators is limited by the number of MPI processes of the application. Additionally, commands such as "lfs osts" obtain the number of OSTs available on a Lustre file system, which can be stored in a configuration file. Once the limits are known, to establish the training set, one can use all discrete integer values as possible tunable parameter values. Another strategy is to use powers-of-two or halves-of-max allowable values. An expert can set these values more judiciously. Since the training is done infrequently, the values can be decided based on the training set exploration time budget. We can easily dictate one strategy over the others.

Following the forward-selection approach on the entire training data set, we obtain one model for each application on each platform. To be concise, here we explicitly give the model only for VPIC-IO on Edison. s is the size of stripes, c is the strip count, a is the number of aggregations, and f is the file size:

$$m(\mathbf{x}) = \beta_1 + \beta_2 \frac{1}{s} + \beta_3 \frac{1}{a} + \beta_4 \frac{c}{s} + \beta_5 \frac{f}{c} + \beta_6 \frac{f}{s} + \beta_7 \frac{cf}{a}, \tag{1}$$

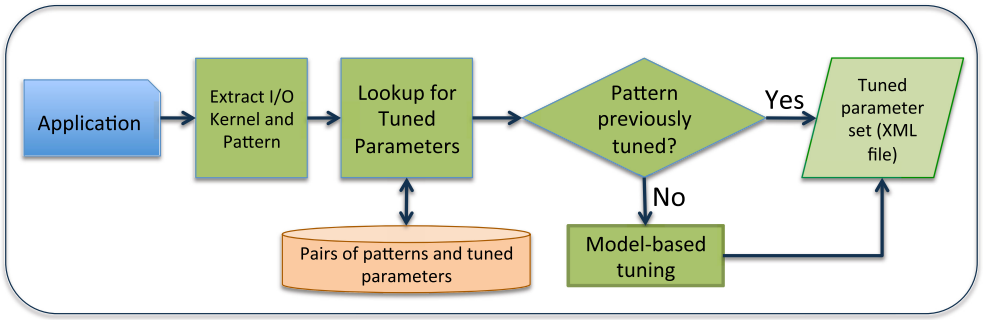


Fig. 9. Overview of our I/O autotuning framework.

with a fit to the data yielding

$$\hat{\beta} = [10.59,\ 68.99,\ 59.83,\ -1.23,\ 2.26,\ 0.18,\ 0.01].$$

The terms in Equation (1) are interpretable from the parallel I/O point of view. For instance, the write time would have an inverse relationship with the number of aggregators and stripe count because, as we increase those, the I/O performance tends to increase. It should have a linear relationship with the file size because increasing the file size causes an increase in the write time.

After training the model for the search space pruning step, the process of choosing the top-k configurations involves only evaluating the model, a task whose computational expense is negligible (relative to evaluation of a configuration) for our simple choice of models. Therefore, using such an approach will require evaluation of only a few configurations on the platform, decreasing the optimization time significantly.

In our experiments, shown in Section 10, the top 20 configurations always resulted in high I/O bandwidth. As opposed to our GA-based approaches, our approach does not spend excessive time in evaluating configurations that have low performance.

If our approach is not able to achieve competitive I/O rates, one can simply refit the model using the new results gathered at the exploration step. Also note that in our results, exploring the top 20 configurations proposed by the model was sufficient. The choice for the number of top-performing configurations is a variable that can be chosen based on one's tuning time budget.

9 PATTERN-DRIVEN I/O TUNING

In this section, we address the requirements of the autotuning framework discussed in this article. We first define *high-level I/O patterns* to characterize write operations. We use a tracing library to collect high-level I/O calls, such as the HDF5 data model definition and write calls. This library uses binary instrumentation to redirect a set of HDF5 calls to collect the required information. We analyze these traces to obtain the I/O pattern information of a simulation's I/O phase. We then match the patterns with previously tuned I/O kernels to obtain their optimal configurations. If a matching previously tuned pattern is not available, we use our empirical prediction model to find tuning parameters offline and store them in the database for future use.

Figure 9 illustrates an overview of our proposed overall I/O autotuning framework that can address the parallel I/O tuning problem. It starts by extracting the I/O pattern from the application. Once the pattern is extracted, a look-up phase follows in which the pattern is queried in a database of patterns. If the pattern is found in this database, then the model associated with it is used to suggest tuned parameters for it as XML files to be run with the application.

To be able to automatically extract the I/O activities of an application, we need to first extract the characteristics of the I/O operations it is conducting. The I/O trace of an application is used for this end. In previous work, we developed a multi-level I/O tracer tool, called Recorder (Luu et al. 2013); it uses dynamic library preloading and intercepting I/O functions at different levels of the I/O stack. We observed that the best level of the I/O stack to define I/O patterns is at the higher-level I/O libraries such as HDF5. Therefore, we used the Recorder to capture all the HDF5 I/O operations of an application. At the end of one run of the application on P processes, P trace files are generated by the Recorder library. Once these files are generated, in addition to extracting the I/O patterns from them, one can use them to create an I/O kernel from them by running the sequence of I/O calls in them on each processor either with or without a pause for the computation and communication of the application between them.

An I/O pattern of an application can be defined in many ways. Following the approach of the database community, we separate the I/O pattern of an application into two categories:

- **Physical Pattern:** The physical pattern of read/write operations is related to the hardware configuration and is specific to the file system, platform, and so forth. These patterns were discussed in preceding sections, and statistical models have been proposed for them. They are the models that have either a linear or inverse relationship with file system parameters, such as Lustre stripe settings, and the I/O middleware layer, such as MPI-IO settings. We showed that different I/O benchmarks have different relationships with these parameters and that one can generalize the models to take the number of processes and file sizes into account as well.
- **Logical Pattern:** The logical pattern is defined at the application level and is the focus of this section. This is the pattern that takes into account the number of processes that run the application as well as the distribution of the data between them. Higher-level I/O libraries divide the I/O operations into two categories.
  (1) **Metadata:** Metadata of a high-level library includes information about the data itself, such as datatypes, and dimensions. This also includes information about the data that the user may want to save such as attributes. The size of metadata is small and is typically stored in the first part of the file.
  (2) **Raw Data:** Raw data is the main data and the bigger portion of the file, and the main I/O time is spent in doing I/O operations for it. The main difference between the I/O operations of different applications exists in the access to raw data. Applications can do the I/O operations contiguously or noncontiguously. They can access the raw data in horizontal stripes or vertical stripes. They can even have random selections of this raw data. The main focus of this section is to abstract these kinds of patterns.

We believe that in order to have a more accurate definition of the logical I/O patterns, we can utilize the same division. High-level I/O libraries give us much more information in order to define and distinguish the way different applications conduct I/O operations. One example and probably the main one is the concept of *selection* in HDF5. Selection is an important and powerful feature of the HDF5 library, which lets the developers select different parts of a file and different parts of memory in order to conduct I/O operations. It also is the main mechanism for the processes to choose different parts of the file in a parallel I/O application. Therefore, we base our definition of I/O patterns on the concept of selection. In summary, we define the I/O pattern of an application as a coverage of the datasets based on the selections they make.

In HDF5 terminology, *hyperslab*s are portions of datasets, either a logically contiguous collection of points in a dataspace or a regular pattern of points or blocks in a dataspace. In a parallel HDF5 program, once each process defines both the memory and file hyperslabs, the process executes

```
$ ./H5Analyze WRITE 1 testlog/pH5example_4 4
.
.
.
I/O Pattern with HPF Terminology:
Dataset name: output/ParaEg0.h5/Data1
  - Dimension: 2
  - Distribution: <BLOCK, DEGENERATE>
  - Size: <6, 24>
Dataset name: output/ParaEg0.h5/Data2
  - Dimension: 2
  - Distribution: <BLOCK, DEGENERATE>
  - Size: <6, 24>
```

Fig. 10. Output related to the I/O pattern of H5Analyze code for pH5example code.

a partial read/write (Group 2011). The hyperslabs are selected using the `H5Sselect_hyperslab` function. The four parameters that can be passed to this function are `start`, `stride`, `count`, and `block`. The `start` array is used by each process to specify the starting location for the hyperslab. The `stride` array specifies the distance between two consecutive selected elements or blocks. The `count` array specifies the number of the elements or blocks to select; and the `block` array specifies the size of the block selected from the dataspace.

To abstract these patterns, we make use of the array distribution notation that was also used in high performance fortran (HPF) (Richardson 1996). HPF uses data distribution directives to help the programmer distribute data between processes. In particular, the `DISTRIBUTE` directive is used to specify the partitioning of the array data onto an abstract processor array. The basic distributions are `BLOCK`, `CYCLIC`, and `DEGENERATE`. A different distribution can be used for each dimension. Below is a short description of each of these distributions.

(1) **Block Distribution:** Each process gets a single contiguous block of the array.
(2) **Cyclic Distribution:** Array elements are distributed in a round-robin manner: i.e., the first element is on the first process, the second element on the second process, and so on.
(3) **Degenerate Distribution:** Degenerate distribution, represented by *, is basically no distribution or serial distribution. All the elements of this dimension are assigned to one processor.

The advantage of this representation is that it is succinct enough to be stored in a key-value store, called the I/O pattern repository. Once the distribution of each of the datasets is found and stored, the patterns should be queried in the database. Therefore, the runtime system we envision consists of three main components.

(1) **H5Analyze:** H5Analyze is a code we have developed based on pattern analysis provided by Zou et al. (2015) for analyzing HDF5 read and write traces. Our implementation contains structures for storing information about HDF5 files, dataspaces, datasets, selections, and operations. It accepts traces gathered by the Recorder (Luu et al. 2013) from all the processes as input and populates this information by reading these traces. Once this information is stored, the H5Analyze code starts to execute analysis on them in order to come up with the patterns and output them in HPF terminology. Figure 10 shows the output of H5Analyze for the pH5example, a simple test HDF5 application. As can be seen, once the correct arguments are given to H5Analyze, it can find out the dimension, distribution, and size of the access pattern of each dataset.

Table 3. I/O Rate and Speedups of I/O Benchmarks with Tuned Parameters over Default Parameters

Application/ # Cores	Platform	Bandwidth (MB/s)								
		VPIC-IO			VORPAL-IO			GCRM-IO		
		Default	Tuned	Speedup	Default	Tuned	Speedup	Default	Tuned	Speedup
128	Hopper	400	3,034	7.57	378	2,614	6.90	757	2,348	3.10
	Intrepid	659	1,126	1.70	846	1,102	1.30	255	1,801	7.05
	Stampede	394	2,328	5.90	439	2,130	4.85	331	2,291	6.90
2,048	Hopper	365	8,464	23.18	370	9,233	24.89	240	17,816	74.12
	Intrepid	2,282	5,964	2.61	2,033	4,842	2.38	414	870	2.10
	Stampede	380	13,047	34.28	436	12,542	28.70	128	13,825	107.73
4,096	Hopper	348	17,620	50.60	320	12,192	38.00	–	–	–
	Intrepid	2,841	7,014	2.46	3,131	7,766	2.47	–	–	–
	Stampede	–	–	–	–	–	–	–	–	–

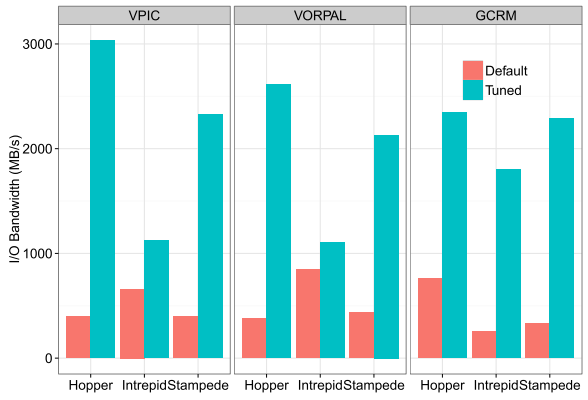
(2) **Key-Value Store:** To store the patterns associated with their I/O performance model, ultimately, we should use a database as the number of patterns increases. For now, however, we are using text files because it is easier to store the patterns in text files without requiring a global database.
(3) **Modeling Component:** If the pattern of the input application is not found in the key-value store, since no model is associated with it, the framework needs to come up with a model for it. This is the focus of the preceding sections and includes a training phase in which the model is trained for a set of different values for each of the parameters at different core counts. Note that we have chosen to have a separate component for this in our framework because it may be improved over time.

10 EXPERIMENTAL RESULTS

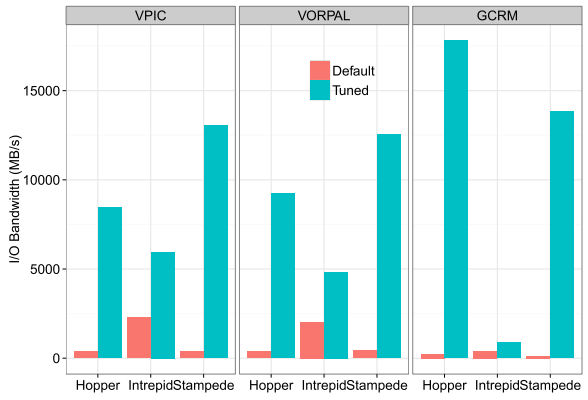
10.1 Genetic Algorithm Results

The plots in Figure 11 present the I/O rate improvement by using the tuned parameters that our autotuning system detected for the three I/O benchmarks. H5Evolve ran for 10 hours, 12 hours, and 24 hours for the three concurrencies to search through the parameter space of each experiment. In most cases, GA evolved through 15 to 40 generations. We selected the tuned configuration that achieves the best I/O performance through the course of the GA evolution. Figure 11 compares the tuned I/O rate with the default I/O rate for all applications on all HPC systems at 128, 2,048, and 4,096 core concurrencies. We calculated the I/O rate as the ratio of the amount of data a benchmark writes into an HDF5 file at any given scale to the time taken for writing the data. The time taken includes the overhead of opening, writing, and closing the HDF5 file. The I/O rate on the y-axis is expressed in megabytes per second. Readers should note that the range of the I/O rate shown in the three plots is different. The measured default I/O rate for a benchmark on an HPC platform is the average I/O rate we obtained after running the benchmark multiple times. The default experiments correspond to the system default settings that users of the HPC platform typically would encounter should they not have access to an autotuning framework.

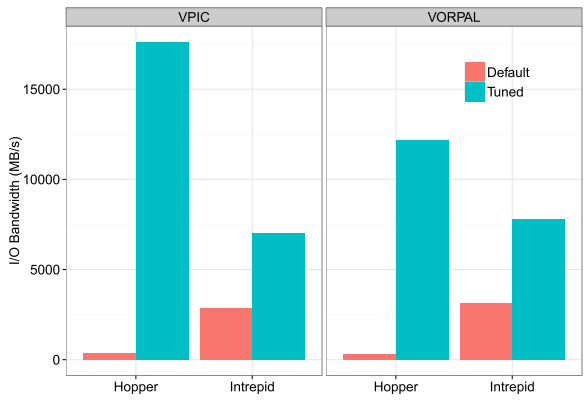
Table 3 shows the raw I/O rate numbers of the default and the tuned experiments for all 22 experiments. We also show the speedup that the autotuned settings achieved over the default settings for each experiment. For all the benchmarks, platforms, and concurrencies, the speedup numbers are generally between 1.3X and 38X, with 50X, 70X, and 100X speedups in three cases. We note that the default I/O rates for the Intrepid platform are noticeably higher than those on



(a) 128 cores



(b) 2,048 cores



(c) 4,096 cores

Fig. 11. Summary of performance improvement for each I/O benchmark running on (a) 128 cores, (b) 2,048 cores, and (c) 4,096 cores. The scales of the I/O bandwidth axes are different in the plots. Note that only a subset of the combinations were run due to limited access to the platforms.

Table 4. Highest Bandwidth Achieved for the Three Applications by Selecting the Best-Performing Configuration Suggested by Our Proposed Framework

# of Cores	I/O Kernel	File Size (GB)	Edison (GB/s)	Hopper (GB/s)	Stampede (GB/s)	Hopper Default (GB/s)
512	VPIC	128	8.19	3.00	9.30	0.39
	VORPAL	140.625	3.24	2.67	7.76	0.44
	GCRM	166.4	9.78	5.27	11.62	-
1K	VPIC	256	14.24	5.09	14.71	0.32
	VORPAL	281.25	9.91	2.34	9.10	0.41
	GCRM	166.4	14.63	6.70	13.28	-
2K	VPIC	512	19.72	8.18	14.75	0.40
	VORPAL	562.5	17.81	4.63	12.67	0.36
	GCRM	665.6	23.96	6.82	21.05	0.24
4K	VPIC	1,024	20.57	12.57	29.20	0.34
	VORPAL	1,197	10.26	4.50	15.35	0.31
	GCRM	2,600	16.64	10.59	26.99	0.41
8K	VPIC	2,048	24.32	18.93	-	0.20
	VORPAL	2,250	12.77	7.26	-	0.33
	GCRM	10,400	28.60	22.09	-	-
16K	VPIC	512	23.21	21.96	-	-
	VORPAL	4,394	15.20	9.45	-	-
	GCRM	10,400	24.58	19.73	-	-

Hopper and Stampede. Hence, the speedups on Hopper and Stampede with tuned parameters are much higher than those on Intrepid.

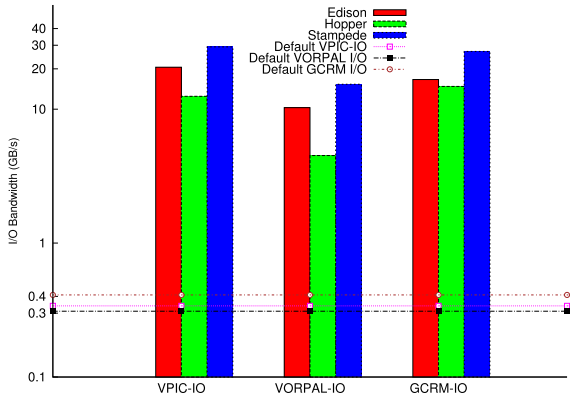
10.2 Performance Modeling Results

The best I/O bandwidth results we have obtained for each of the applications on different platforms are summarized in Figure 12. For each experiment, this is the best-performing configuration among the top 20 configurations predicted by the model. The figure shows the I/O bandwidth grouped by the number of cores from 4K to 16K. For all these experiments, we used the training phase experiments without a refitting phase. As can be observed, the I/O bandwidths of the kernels are in the range of 5–30GB/s, which is efficient performance for writing to one shared file on these platforms at their respective scales. We also show the default I/O performance of the applications for their respective concurrencies at 4K and 8K on the Hopper platform. Compared to the default performance, our tuned configurations perform 6X–94X better. We expect the default performance and our speedup to be at the same level for the other platforms. Note that for the Stampede platform, we have scaled our runs only up to 4K cores because of limitations of Stampede in running large-scale tests.

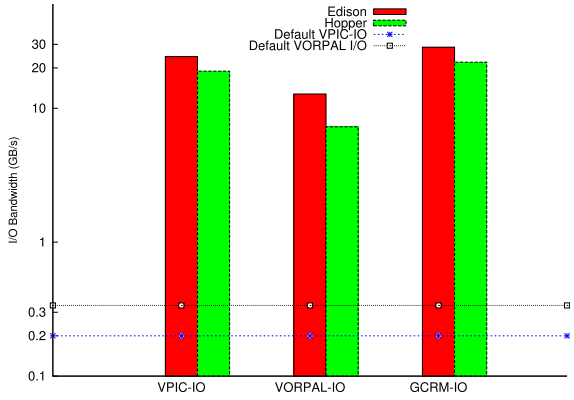
Table 4 summarizes the achieved I/O bandwidths for the three I/O kernels running at different concurrencies on the three platforms. The table also shows the size of the data written to the file system. The time to traverse the search space after training took less than 3 hours. In most cases, exploring the top 20 configurations took less than 1 hour, resulting in significant improvements to overall parallel I/O performance.

10.3 Pattern-Driven Tuning Results

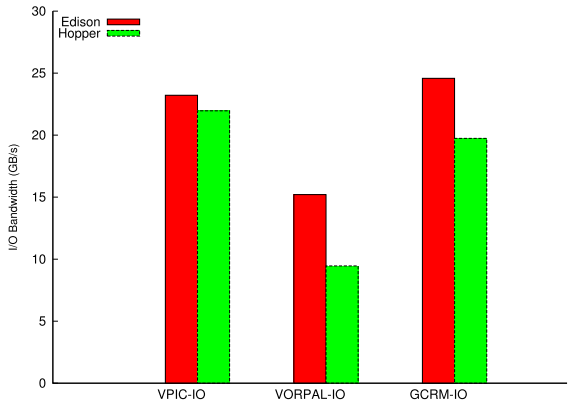
This section shows the results of the pattern-driven I/O tuning framework. The first experiment tests whether our framework is capable of identifying an I/O pattern exactly similar to what it has



(a) 4K cores



(b) 8K cores



(c) 16K cores

Fig. 12. I/O bandwidth for the three applications on a different number of cores. Note that subplots (a) and (b) use a log scale for the y-axis in order to show the value for default I/O bandwidth, while subplot (c) does not.

tuned before and configures the I/O correctly. The second experiment tests whether a new pattern, but similar to the ones in the database, is recognized and that the model used for the most similar application to it in the database can lead to acceptable I/O performance. The third experiment examines an arbitrary application that does not have any similar patterns in the database.

Note that for the results of this section, we use all the developed models in the preceding sections. Therefore, there was no tuning for any application for these experiments, and we have used the models developed for them in our previous work.

For the first experiment, we use the IOR benchmark. The second experiment uses a synthetic benchmark called Resemble-VORPAL-IO, which is similar to the VORPAL-IO pattern but with different block sizes. The third experiment involves a new I/O benchmark: FLASH-IO.

10.3.1 Application with the Same I/O Pattern. To have IOR issue write patterns similar to VPIC-IO, we configured it to use its HDF5 interface. Since VPIC-IO writes eight datasets, we need to configure IOR accordingly. To this end, we used segments (`-s 8` command line option) of IOR. VPIC-IO only writes operations, and we use writeFile (`-w` option) for IOR. Since each dataset of VPIC-IO contains 32MB of data per processor, we use the block size (`-b 32m` option) of IOR along with the transfer size of 32MB (`-t 32m` command line option).

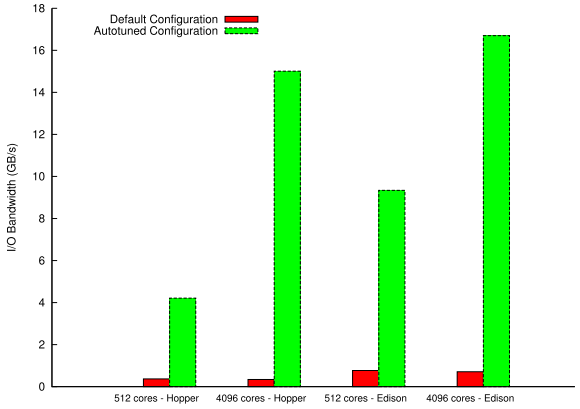
Figure 13(a) shows the performance of the autotuned configuration that was proposed for IOR, since it has the same pattern as VPIC-IO, on 512 and 4,096 cores of Hopper and Edison in Behzad et al. (2014a). As mentioned before, no modeling effort was done for this application and yet we can see that we get up to 4.21GB/s and 15.01GB/s on 512 and 4,096 cores of Hopper. On Edison, these numbers are 9.34GB/s and 16.70GB/s.

10.3.2 Application with Similar I/O Pattern. Resemble-VORPAL-IO is a synthetic benchmark generated by Record-and-Replay framework (Behzad et al. 2014b). It has an I/O pattern similar to those of the VORPAL-IO benchmark but with different block sizes of $64 \times 128 \times 256$ instead of $60 \times 100 \times 300$ for VORPAL-IO. The purpose of this experiment is twofold: (1) to show that applications with similar I/O patterns with slight differences only in block sizes can use the same I/O configuration to obtain good I/O performance, and (2) to show that requiring a threshold for the similarity between I/O patterns can save dramatic I/O tuning time.

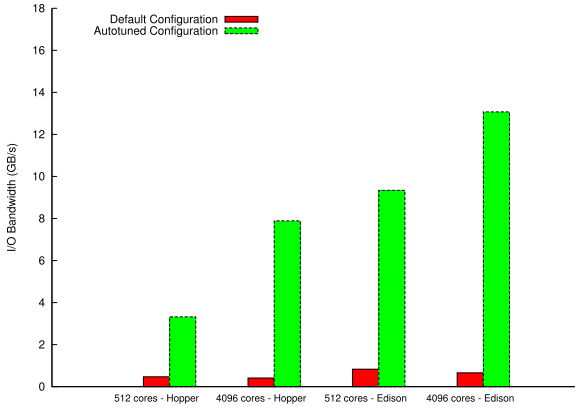
Figure 13(b) shows the performance of the autotuned configuration that was proposed for Resemble-VORPAL-IO on 512 and 4,096 cores of Hopper and Edison in Behzad et al. (2014a). Similar to the previous experiment, no modeling effort was done for this application, and yet we are able to get up to 3.32GB/s and 7.89GB/s on 512 and 4,096 cores of Hopper, respectively. On Edison the highest bandwidth achieved by this mechanism was 8.75GB/s and 13.07GB/s on the same number of cores.

10.3.3 New Application. The last experiment is designed to test an arbitrary application that has not been tuned before. For this experiment, we chose to test a well-known I/O kernel called FLASH-IO because it is popular in the HPC I/O community and also hard to tune. As in the previous experiment, we ran FLASH-IO at two scales, 512 and 4,096 cores of Hopper and Edison. The way that we calculate the bandwidth for this application is a little different from the other ones because it produces three files. The definition of bandwidth here is basically just the sum of all the output sizes divided by the runtime of the whole I/O benchmark, which is a conservative way of defining the I/O bandwidth of an application.

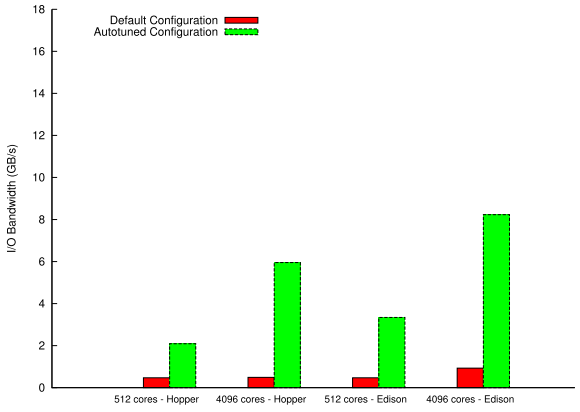
FLASH-IO is different from the other applications we have looked at mainly because it writes many datasets with different I/O patterns. In order to overcome this problem the framework considers the largest datasets and looks for those patterns in the database. Based on the output of H5Analyze, FLASH-IO has 34 datasets, of which 24 have the same size as the largest size of the



(a) IOR



(b) Resemble-VORPAL-IO



(c) FLASH-IO

Fig. 13. I/O performance of the autotuned (a) IOR, (b) Resemble-VORPAL-IO, and (c) FLASH-IO application on Hopper and Edison compared with the default configuration.

Table 5. Comparison of GA, Modeling, and Default Configuration

Method	Training Phase	Applying the Model	Per App. & Scale Tuning	App. Runtime (VPIC-8192 on Hopper)
GA	N/A	N/A	>10 hours	118 seconds
Model Fitting	>10 hours (can reuse)	<1 minute (automatic)	<1 hour	100 seconds
Default Config.	none	none	none	>3 hours

file. On 4,096 cores, this is about 40GB for each dataset. These datasets are 4D, and their patterns are also the same: <BLOCK, DEGENERATE, DEGENERATE, DEGENERATE>. Although the exact same pattern does not exist, GCRM-IO has the most similar pattern to this application, and, therefore, the framework uses the proposed configurations for GCRM-IO.

Figure 13(c) shows the performance of the autotuned configuration that was proposed by our framework for FLASH-IO based on the GCRM-IO model, on 512 and 4,096 cores of Hopper and Edison. Similar to the previous experiment, no modeling effort was done for this application, and yet we are able to get up to 2.09GB/s and 5.95GB/s on 512 and 4,096 cores of Hopper. On Edison, the highest bandwidth achieved by this mechanism was 3.34GB/s and 8.23GB/s on the same number of cores, respectively.

11 DISCUSSION AND LIMITATIONS

In this section, we first measure the overhead of the framework based on the order they appear in the architecture discussed in Section 6. Regarding the extraction of I/O pattern from an application, the framework needs at least one run of the application linked to the Recorder in order to gather the traces. The overhead of the Recorder library is minimal (Luu et al. 2013). Once the traces are gathered, H5Analyze analyzes the traces to determine the I/O pattern. H5Analyze is a sequential application, written in C programming language, that reads in the traces and comes up with the pattern with a fast turn-around time even with large-scale trace files. The looking-up phase is also fast because the number of patterns is small. If the pattern of an application is found, the I/O configuration of the application is proposed with an XML file. If not, the main part of the overhead of our framework is exerted: the modeling phase. For those patterns for which the framework is not able to find a match, the autotuning framework is used to initialize the modeling process by running the application with its training set. This may require more than several hours to come up with a nonlinear regression model. Once the model is developed, the framework will associate the pattern along with the new model for any future run of the application.

Table 5 shows a comparison of these approaches. With default configuration without any I/O tuning, each application will take more than 3 hours. With genetic algorithms, for each application and scale, a cost of more than 10 hours is paid for tuning. With the current approach, the cost of training is paid once, and then for each application, applying the model takes less than 1 hour with fast application run-time.

12 RELATED WORK

Autotuning in computer science is a prevalent strategy for improving the performance of computational kernels. Extensive research has been done in developing optimized linear algebra libraries and matrix operation kernels using autotuning (Whaley et al. 2001; Frigo and Johnson 1998; Bilmes et al. 1997; Vuduc et al. 2005; Williams et al. 2007; Datta et al. 2008; Williams et al. 2008). The search space in these efforts involves optimization of CPU cache and DRAM parameters along with code changes. All these autotuning techniques search various data structures and code transformations

using performance models of processor architectures, computation kernels, and compilers. Our study focuses on autotuning the I/O subsystem for writing data to a parallel file system in contrast to tuning computational kernels. A few key challenges are unique to the I/O autotuning problem. Each function evaluation for the I/O case takes on the order of minutes, as opposed to milliseconds for computational kernels. Thus, an exhaustive search through the parameter space is infeasible, and a heuristic-based search approach is needed. I/O runs also face dynamic variability and system noise whereas linear algebra tuning assumes a clean and isolated single node system. Moreover, the interaction between various I/O parameters and how they impact performance is not well-studied, making interpreting tuned results a complex task.

We use genetic algorithms as a parameter-space-searching strategy. Heuristics and metaheuristics have been studied extensively for combinatorial optimization problems as well as code optimization (Seymour et al. 2008) and parameter optimization (Casanova et al. 2000) problems similar to the one we addressed. Of the heuristic approaches, genetic algorithms seem to be particularly well-suited for real parameter optimization problems, and a variety of literature exists detailing the efficacy of the approach (Bäck and Schwefel 1993; Deb et al. 2002; Wright 1991). A few recent studies have used genetic algorithms (Tiwari and Hollingsworth 2011) and a combination of approximation algorithms with search space reduction techniques (Jordan et al. 2012). Both of these are again targeted to autotune compiler options for linear algebra kernels. We chose to implement a genetic algorithm to attempt to intelligently traverse the sample space for each test case; we found our approach produced well-performing configurations after a suitably small number of test runs.

Various optimization strategies have been proposed to tune parallel I/O performance for a specific application or an I/O kernel. However, they are not designed for automatic tuning of any given application and require manual selection of optimization strategies. Our autotuning framework is designed for tuning an arbitrary parallel I/O application. Hence, we do not discuss the exhaustive list of research efforts. We focus on comparing our research with automatic performance-tuning efforts.

A few research efforts have been conducted to autotune and optimize resource provisioning and system design for storage systems (Alvarez et al. 2001; Anderson et al. 2002; Strunk et al. 2008). In contrast, our study focuses on tuning the parallel I/O stack on top of a working storage system.

The Panda project (Chen et al. 1996; 1998b) studied automatic performance optimization for collective I/O operations where all the processes were used by an application to synchronize I/O operations such as reading and writing an array. The Panda project searched for disk layout and disk buffer size parameters using a combination of a rule-based strategy and randomized search-based algorithms. The rule-based strategy is used when the optimal settings are understood; simulated annealing is used otherwise. The simulated annealing problem is solved as a general minimization problem, where the I/O cost is minimized. The Panda project also used genetic algorithms to search for tuning parameters (Chen et al. 1998a). The optimization approach proposed in this project was applicable to the Panda I/O library, which existed before MPI-IO and HDF5. The Panda I/O is not in use now, however, and the Panda optimization strategy was not designed for current parallel file systems because of the new tunable parameters in the current systems.

You et al. (2011) proposed an autotuning framework for the Lustre file system on the Cray XT5 systems at Oak Ridge National Laboratory (ORNL). They search for file system stripe count, stripe size, I/O transfer size, and the number of I/O processes. This study uses mathematical models based on queuing models. The autotuning framework first develops a model in a training phase that is close to the real system. The framework then searches for optimal parameters using search heuristics such as simulated annealing, and genetic algorithms. Developing a mathematical model for different systems based on queuing theory can be further from the real system, however, and

may produce inaccurate performance results. In contrast, our framework searches for parameters on real systems using search heuristics.

Several efforts have been made to predict parallel I/O performance. Shan et al. (2008) use the IOR benchmark to match the I/O patterns of an application and predict I/O performance. Meswani et al. (2010) use a similar strategy by running the I/O operations of an application on a reference system and calibrating the performance of the reference system with a target system. Smirini et al. [1997] use a queuing network model to predict the performance of RAID-3 disks. Song et al. (2013) propose an analytical model to predict the cost of read operations for accessing data organized in different layouts on the file system. Kumar et al. (2013) use various machine-learning algorithms for improving performance of I/O in a PIDX file format library; their prediction focuses on network and I/O performance while keeping the stripe settings fixed. While many of these efforts seek to predict I/O performance accurately, our work uses the models to identify fruitful parameter values and then iterates in the executing and refitting stages by searching among this smaller set of parameter values. Using this approach, we have shown that our technique is fast and effective in achieving good I/O performance.

I/O Signature (Byna et al. 2008) is a notation consisting of five dimensions of I/O operations: operation, spatial offset, request size, repetitive behavior, and temporal intervals. These are then gathered by a framework for each application and stored persistently for later look-up in order to help prefetching.

Statistical models such as Markov models have been proposed for a long time producing and predicting I/O operations and file system performance, (Smirni and Reed 1998; Simitci and Reed 1998). These are then used more in the context of prefetching, caching, or scheduling.

Omnisc'IO (Dorier et al. 2014) is a grammar-based I/O model with the aim of capturing and predicting I/O operations of an application. At its heart, it uses an algorithm based on the Sequitur algorithm that, given a sequence of symbols, builds a grammar for text compression. It supports both spatial and temporal patterns in this regard. To be more general, the authors use the program's stack trace as the symbols of the grammar. One strength of their approach is that it does real-time prediction as the grammar is being updated in the algorithm. This is similar to what we called "real-time tuning" in the article.

He et al. (2013) correctly argue that a lot of information gets lost in a typical I/O stack as the data flows between its layers. Although high-level I/O libraries contain rich information about the data structures, eventually, they get down into simple offset and length pairs in the storage system. Their solution to this problem is to "rediscover these structures in unstructured I/O" using a gray-box technique. Our approach, however, is to avoid losing the data by intercepting it at the higher levels. In terms of framework design, there are some similarities such as the way the pattern detection engine works. Since it is at POSIX level, however, it has a local pattern structure and a global one. For the local one, a modified algorithm based on Lempel-Ziv 77 (LZ77) is presented and for the global patterns, these local patterns are sorted in order to check for a pattern between them. These are not necessary in our work.

13 CONCLUSIONS

In this article, we describe different parts of an I/O autotuning framework. We present two ways of implementing an autotuning framework for optimizing I/O performance of scientific applications. The first framework is capable of transparently optimizing all levels of the I/O stack, consisting of HDF5, MPI-IO, and Lustre/GPFS parameters, without requiring any modification of user code. We have successfully demonstrated the power of the framework by obtaining a wide range of speedups across diverse HPC platforms, benchmarks, and concurrencies. Note that NetCDF-4 is based on HDF5 and, therefore, all the HPC applications using NetCDF-4 can also benefit from this work.

As the second framework, we present a model-driven tuning framework that exploits nonlinear regression models to find the top-performing values for these parameters in order to decrease the amount of I/O time in HPC applications. We show that our approach achieves a significant portion of the available I/O performance of various HPC platforms for a range of applications. We propose a pattern-driven autotuning framework to solve this problem. It consists of components to extract I/O patterns, tune configurations for the detected patterns, store them in a database of patterns associated with their I/O model, and map an arbitrary I/O pattern to a previously tuned model in order to improve its I/O performance. We show that using these patterns, one can tune different sets of applications ranging from the ones that have been tuned, to the ones that are similar, to the previously tuned ones, and to totally new ones. We believe that the autotuning framework can provide a route to hiding the complexity of the I/O stack from application developers, thereby providing a truly performance-portable I/O solution for scientific applications.

ACKNOWLEDGMENTS

This research used resources of the National Energy Research Scientific Computing Center, the Texas Advanced Computing Center, and the Argonne Leadership Computing Facility. The authors acknowledge Marianne Winslett, William Gropp, Qunicey Koziol, Huong Luu, Stefan Wild, Wes Bethel, Mohamad Chaarawi, John Shalf, and Venkat Vishwanath for their support and guidance.

REFERENCES

Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. 2001. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.* 19, 4 (2001), 483–518.

Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. 2002. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*. USENIX Association, Berkeley, CA, Article 13.

Thomas Bäck and Hans-Paul Schwefel. 1993. An overview of evolutionary algorithms for parameter optimization. *Evol. Comput.* (1993), 1–23.

Babak Behzad, Surendra Byna, Stefan M. Wild, Mr. Prabhat, and Marc Snir. 2014a. Improving parallel I/O autotuning with performance modeling. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC'14)*.

Babak Behzad, Hoang-Vu Dang, Farah Hariri, Weizhe Zhang, and Marc Snir. 2014b. Automatic generation of I/O kernels for HPC applications. In *Proceedings of the 9th Parallel Data Storage Workshop (PDSW'14)*. IEEE Press, Piscataway, NJ.

Babak Behzad, Luu Huong Vu Thanh, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. 2013. Taming parallel I/O complexity with auto-tuning. In *Proceedings of 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2013) (SC'13)*.

E. W. Bethel, J. M. Shalf, C. Siegerist, K. Stockinger, A. Adelmann, A. Gsell, B. Oswald, and T. Schietinger. 2007. Progress on H5Part: A portable high performance parallel data interface for electromagnetics simulations. In *Proceedings of the 22nd IEEE Particle Accelerator Conference (PAC'07)*. 3396.

Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th International Conference on Supercomputing (ICS'97)*. 340–347.

K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. 2008. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Phys. Plasmas* (2008), 7.

Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. 2008. Parallel I/O prefetching using MPI file caching and I/O signatures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*. 44:1–44:12.

Surendra Byna, Jerry Chou, Oliver Rübel, Prabhat, and et al. 2012. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. 59:1–59:12.

Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. 2000. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*. 349.

Y. Chen, M. Winslett, Y. Cho, and S. Kuo. 1998a. Automatic parallel I/O performance optimization using genetic algorithms. In *Proceedings of the 7th International Symposium on High Performance Distributed Computing*. 155–162.

Y. Chen, M. Winslett, Y. Cho, S. Kuo, and Contact Y. Chen. 1998b. Automatic parallel I/O performance optimization in Panda. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*. 108–118.

Ying Chen, Marianne Winslett, Szu-wen Kuo, Yong Cho, Mahesh Subramaniam, and Kent Seamons. 1996. Performance modeling for the Panda array I/O library. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM) (Supercomputing'96)*.

Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat, and Parkson Wong. 1996. Overview of the MPI-IO parallel I/O interface. In *Input/Output in Parallel and Distributed Computer Systems*. Springer, 127–146.

Cray. 2016. Cray Message Passing Toolkit Release Overview. Retrieved from http://docs.cray.com/books/S-3689-24/.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*. 4:1–4:12.

Kalyanmoy Deb, Ashish Anand, and Dhiraj Joshi. 2002. A computationally efficient evolutionary algorithm for real-parameter optimization. *Evol. Comput.* (2002), 371–395.

Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. 1993. Improved parallel I/O via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News* (1993), 31–38.

Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. 2014. Omnisc'IO: A grammar-based approach to spatial and temporal I/O patterns prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. Piscataway, NJ, 623–634.

Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of Supercomputing*.

M. Frigo and S. G. Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*. 1381–1384.

The HDF Group. 2011. HDF5 Tutorial - Parallel Topics. Retrieved from http://www.hdfgroup.org/HDF5/Tutor/parallel.html.

HDF5. 2000-2010. Hierarchical data format version 5. Retrieved from http://www.hdfgroup.org/HDF5.

Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. 2013. I/O acceleration with pattern detection. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC'13)*. ACM, New York, NY, 25–36.

Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. 2012. A multi-objective auto-tuning framework for parallel codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA, 10:1–10:12.

David Knaak and Dick Oswald. 2009. *Optimizing MPI-IO for Applications on Cray XT Systems*. Report S-0013-10. Retrieved from docs.cray.com/books/S-0013-10//S-0013-10.pdf.

Sidharth Kumar, Avishek Saha, Venkatram Vishwanath, Philip Carns, John A. Schmidt, Giorgio Scorzelli, Hemanth Kolla, Ray Grout, Robert Latham, Robert Ross, Michael E. Papkafa, Jacqueline Chen, and Valerio Pascucci. 2013. Characterization and modeling of PIDX parallel I/O for performance optimization(SC'13). 67:1–67:12.

Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. 2009. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM, New York, NY, 40:1–40:12.

Wei-Keng Liao and A. Choudhary. 2008. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*. 1–12.

LLNL. 2015. IOR. Retrieved from https://github.com/chaos/ior.

Jeremy Logan, Scott Klasky, Jay Lofstead, Hasan Abbasi, Stephane Ethier, Ray Grout, Seung-Hoe Ku, Qing Liu, Xiaosong Ma, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. 2011. Skel: Generative software for producing skeletal I/O applications. In *Proceedings of the 2011 IEEE 7th International Conference on e-Science Workshops (ESCIENCEW'11)*. IEEE Computer Society, Washington, DC, 191–198.

Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. 2013. A multi-level approach for understanding I/O activity in HPC applications. In *IEEE International Conference on Cluster Computing (CLUSTER'13)*.

M. R. Meswani, M. A. Laurenzano, L. Carrington, and A. Snavely. 2010. Modeling and predicting disk I/O time of HPC applications. In *High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC)*. 478–486.

C. Nieter and J. R. Cary. 2004. VORPAL: A versatile plasma simulation code. *J. Comput. Phys.* (2004), 448–472.

David Pearah and Jerome Soumagne. 2016. HDF5 State of the union - Birds-of-feather. Retrieved from https://support.hdfgroup.org/pubs/presentations/SC2016-HDF-BoF-Slides.pdf.

David Randall, Marat Khairoutdinov, Akio Arakawa, and Wojciech Grabowski. 2003. Breaking the cloud parameterization deadlock. *Bull. Amer. Meteor. Soc.* (2003), 1547–1564.

Harvey Richardson. 1996. *High Performance Fortran: History, Overview and Current Developments*. Technical Report. 1.4 TMC-261, Thinking Machines Corporation.

Karl Schulz. 2013. Experiences from the Deployment of TACC's Stampede System. Retrieved from http://www.hpcadvisorycouncil.com/events/2013/Switzerland-Workshop/Presentations/Day_1/7_TACC.pdf.

Philip Schwan. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*.

K. Seymour, Haihang You, and J. Dongarra. 2008. A comparison of search heuristics for empirical code optimization. In *Proceedings of the IEEE International Conference on Cluster Computing*.

Hongzhang Shan, John Shalf, and Katie Antypas. 2008. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC'08*. ACM/IEEE, Austin, TX.

Huseyin Simitci and Daniel A. Reed. 1998. A comparison of logical and physical parallel I/O patterns. *Int. J. High Perform. Comput. Appl.* 12 (1998), 364–380.

Evgenia Smirni, Christopher L. Elford, Daniel A. Reed, and Andrew A. Chien. 1997. Performance modeling of a parallel I/O system: An application driven approach. SIAM. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.4971

Evgenia Smirni and Daniel A. Reed. 1998. Lessons from characterizing input/output bahavior of parallel scientific applications. *Int. J. Perform. Eval.* 33, 1 (1998), 27–44.

Huaiming Song, Yanlong Yin, Yong Chen, and Xian-He Sun. 2013. Cost-intelligent application-specific data layout optimization for parallel file systems. *Cluster Comput.* 16, 2 (2013), 285–298.

John Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. 2008. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, 21:1–21:16.

Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS'99)*. IEEE, Washington, DC, 182.

Ananta Tiwari and Jeffrey K. Hollingsworth. 2011. Online adaptive code generation and tuning. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, Washington, DC, 879–892.

Richard Vuduc, James Demmel, and Katherine Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*.

Clint Whaley, Antoine Petitet, and Jack Dongarra. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Comput.* (2001).

S. Williams, K. Datta, J. Carter, L. Oliker, J. Shalf, K. A. Yelick, and D. Bailey. 2008. PERI-auto-tuning memory-intensive kernels for multicore. *Journal of Physics: Conference Series* 125 (2008), 012038.

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07)*. 38:1–38:12.

Alden H. Wright. 1991. Genetic algorithms for real parameter optimization. In *Foundations of Genetic Algorithms*. Morgan Kaufmann, 205–218.

H. You, Q. Liu, Z. Li, and S. Moore. 2011. The design of an auto-tuning I/O framework on Cray XT5 system. In *Proceedings of the Cray Users Group Conference (CUG'11)*.

Xiaocheng Zou, Kesheng Wu, D. A. Boyuka, D. F. Martin, S. Byna, Houjun Tang, K. Bansal, T. J. Ligocki, H. Johansen, and N. F. Samatova. 2015. Parallel in situ detection of connected components in adaptive mesh refinement data. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'15)*. 302–312.

Received December 2015; revised October 2017; accepted January 2019