Taming Parallel I/O Complexity with Auto-Tuning

Babak Behzad
University of Illinois at Urbana-Champaign

Huong Vu Thanh Luu
University of Illinois at Urbana-Champaign

Joseph Huchette
Rice University

Surendra Byna
Lawrence Berkeley National Laboratory

Prabhat
Lawrence Berkeley National Laboratory

Ruth Aydt
The HDF Group

Quincey Koziol
The HDF Group

Marc Snir
Argonne National Laboratory, University of Illinois at Urbana-Champaign

1. ABSTRACT

We present an auto-tuning system for optimizing I/O performance of HDF5 applications and demonstrate its value across platforms, applications, and at scale. The system uses a genetic algorithm to search a large space of tunable parameters and to identify effective settings at all layers of the parallel I/O stack. The parameter settings are applied transparently by the auto-tuning system via dynamically intercepted HDF5 calls.

To validate our auto-tuning system, we applied it to three I/O benchmarks (VPIC, VORPAL, and GCRM) that replicate the I/O activity of their respective applications. We tested the system with different weak-scaling configurations (128, 2048, and 4096 CPU cores) that generate 30 GB to 1 TB of data, and executed these configurations on diverse HPC platforms (Cray XE6, IBM BG/P, and Dell Cluster). In all cases, the auto-tuning framework identified tunable parameters that substantially improved write performance over default system settings. We consistently demonstrate I/O write speedups between 2x and 100x for test configurations.

General Terms

Parallel I/O, Auto-Tuning, Performance Optimization, Parallel file systems

2. INTRODUCTION

Parallel I/O 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, and problem size/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.

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, it would be ideal if a single optimization framework was capable of *generalizing across multiple applications*.

In order to use HPC machines and human resources effectively, it is imperative that we 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.

In this paper, we present our first step towards accomplishing this ambitious goal. We develop an auto-tuning system that searches a large space of configurable parameters for multiple layers and transparently sets I/O parameters at runtime via intercepted HDF5 calls of the I/O stack to identify parameter settings that perform well. We apply the auto-tuning system to three I/O kernels extracted from real scientific applications and identify tuned parameters on three HPC systems that have different architectures and parallel file systems.

In brief, our paper makes the following research contribu-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13 November 17-21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.

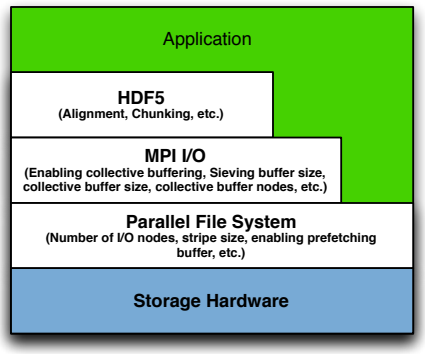


Figure 1: Parallel I/O Stack and various tunable parameters

tions:

- We design and implement an auto-tuning system that hides the complexity of tuning the Parallel I/O stack.
- We demonstrate performance portability across diverse HPC platforms.
- We demonstrate the applicability of the system to multiple scientific application benchmarks.
- We demonstrate I/O performance tuning at different scales (both concurrency and dataset size).

The remainder of the paper is structured as follows: Section 3 presents our I/O auto-tuning system; Section 4 discusses the experimental setup used to evaluate benefits of the auto-tuning system across platforms, applications, and at scale. Section 5 presents performance results from our tests and discusses the insights gained from the auto-tuning effort. Section 6 presents our work in context of existing research literature. Finally, Section 7 presents the limitations of current work and future research directions, and Section 8 offers concluding thoughts.

3. AUTO-TUNING FRAMEWORK

Figure 1 shows a contemporary parallel I/O software stack with HDF5 [27] as the high-level I/O library, MPI-IO as the middleware layer, and a parallel file system (Lustre, GPFS, etc.). While each layer of the stack exposes tunable parameters for improving performance, there is little guidance for application developers on how these parameters interact with each other and affect overall I/O performance. Ideally, an auto-tuning system should provide a unified approach that targets the entire stack and discovers I/O tuning parameters at each layer that result in good I/O rates.

The main challenges in designing and implementing an I/O auto-tuning system are (1) selecting an effective set of tunable parameters at all layers of the stack, and (2) applying the parameters to applications or I/O benchmarks without modifying the source code. We tackle these challenges with the development of two components: *H5Evolve* and *H5Tuner*.

For selecting tunable parameters, a naïve strategy is to execute an application or a representative I/O kernel of the

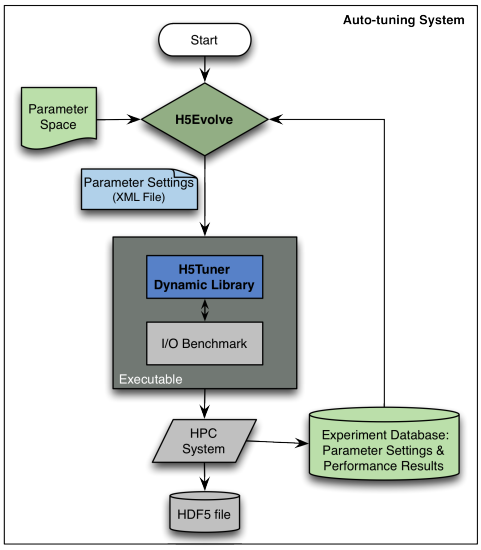


Figure 2: Overall Architecture of the Auto-Tuning Framework

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, as there are many thousands of combinations in a typical parameter space. A reasonable approach is to search the parameter space with a small number of tests. Towards this goal, we developed *H5Evolve* to search the I/O parameter space using a genetic algorithm (GA). H5Evolve samples the parameter space by testing a set of parameter combinations and then, based on 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.

An application can control tuning parameters for each layer of the I/O stack using hints set via API calls. For instance, HDF5 alignment parameters can be set using the `H5Pset_alignment()` function. MPI-IO hints can be set in a similar fashion for the collective I/O and file system striping parameters. While changing the application source code is possible if the code is available, it is impractical when testing a sizable number of parameter combinations. *H5Tuner* solves this problem by dynamically intercepting HDF5 calls and injecting optimization parameters into parallel I/O calls at multiple layers of the stack without the need for source code modifications. H5Tuner is a transparent shared library that can be preloaded before the HDF5 library, prioritizing it over the original HDF5 function calls.

Figure 2 shows our auto-tuning system that uses both H5Tuner and H5Evolve for searching a parallel I/O parameter space. H5Evolve takes the I/O parameter space as input and for each experiment generates a configuration file in 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 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 are fed back to H5Evolve and influence the contents of the

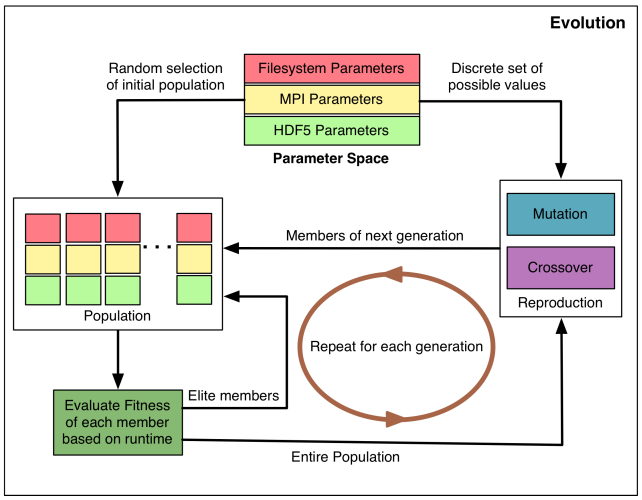


Figure 3: A pictorial depiction of the genetic algorithm used in the auto-tuning framework.

next configuration file. As H5Evolve tests various combinations of parameter settings, the auto-tuning system selects the best performing configuration for a specific I/O kernel or application.

3.1 H5Evolve: Sampling the Search Space

As mentioned previously, due to 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 unmanageably large for a brute force and enumerative optimization approach.

Exact optimization techniques are not appropriate for sampling the search space given the nondeterministic nature of the objective function, which is the runtime of a particular configuration. Instead of relying on the simplest approach, manual tweaking, adaptive heuristic search approaches such as genetic evolution algorithms, simulated annealing, etc., can traverse the search space in a reasonable amount of time. In H5Evolve, we explore genetic algorithms for sampling the search space.

A genetic algorithm (GA) is a meta-heuristic 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.

Our implementation, dubbed H5Evolve, is shown in Figure 3. It was built in Python using the Pyevolve [20] module, which provides an intuitive framework for performing genetic algorithm experiments in Python.

The workflow of H5Evolve is as follows. For a given benchmark at a specific concurrency and problem size, H5Evolve runs the genetic algorithm (GA). H5Evolve takes a predefined parameter space which contains possible values for the I/O tuning parameters at each layer of the I/O stack. The evolution process starts with a 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 of 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 be 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.

3.2 H5Tuner: Setting I/O Parameters at Runtime

The goal of the H5Tuner component is to develop 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 a 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. When the HDF5 calls appear in the code during the execution of a benchmark or application, the H5Tuner library intercepts the HDF5 function calls via dynamic linking. The library reroutes the intercepted HDF5 calls to a new implementation, where the parameters from the configuration are set and then the original HDF5 function is called 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 4, where H5Tuner intercepts an `H5FCreate()` function call that creates an HDF5 file, applies various I/O parameters, and calls the original `H5FCreate()` function call.

H5Tuner uses MiniXML [25], a small XML library to read the XML configuration files. In our implementation, we are reading the configuration file from user's home directory, giving the user the ability to change the configuration file. Figure 5 shows a sample configuration file with HDF5, MPI-IO, and Lustre parallel file system tunable parameters.

4. EXPERIMENTAL SETUP

We have evaluated the effectiveness of our auto-tuning

HPC System	Architecture	Node Hardware	Filesystem	Storage Hardware	Peak I/O BW
NERSC/Hopper	Cray XE6	AMD Opteron processors, 24 cores per node, 32 GB memory	Lustre	156 OSTs, 26 OSSs	35 GB/s [7]
ALCF/Intrepid	IBM BG/P	PowerPC 450 processors, 4 cores per node, 2 GB memory	GPFS	640 IO Nodes, 128 file servers	47 GB/s (write) [18]
TACC/Stampede	Dell PowerEdge C8220	Xeon E5-2680 processors, 16 cores per node, 32GB memory	Lustre	160 OSTs, 58 OSSs	159 GB/s [22]

Table 1: Details of various HPC systems used in this study

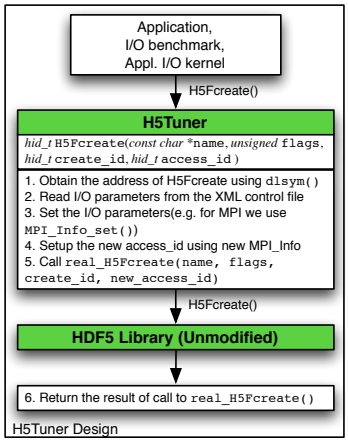


Figure 4: Design of H5Tuner component as a dynamic library which intercepts HDF5 functions to tune I/O parameters

```
<Parameters>
  <High_Level_IO_Library>
    <alignment> 0, 65536 </alignment>
  </High_Level_IO_Library>

  <Middleware_Layer>
    <cb_buffer_size> 1048576 </cb_buffer_size>
    <cb_nodes> 32 </cb_nodes>
  </Middleware_Layer>

  <Parallel_File_System>
    <striping_factor FileName="sample_dataset.h5part"> 4 </striping_factor>
    <striping_factor> 16 </striping_factor>
    <striping_unit> 65536 </striping_unit>
  </Parallel_File_System>
</Parameters>
```

Figure 5: An XML file showing a sample configuration with optimization parameters at different levels of the parallel I/O stack. The tuning can be applied to all files an application writes or to a specific file.

framework on three HPC platforms using three I/O benchmarks at three different scales. The HPC platforms include Hopper, a Cray XE6 system at National Energy Research Scientific Computing Center (NERSC); Intrepid, an IBM BlueGene/P (BG/P) system at Argonne Leadership Computing Facility (ALCF); and Stampede, a Dell PowerEdge C8220 cluster at Texas Advanced Computing Center (TACC). The I/O benchmarks are derived from the I/O traces of the VPIC, VORPAL, and GCRM applications. We ran these benchmarks using 128, 2048, and 4096 cores. In the following subsections, we briefly explain the I/O subsystem of the machines, the benchmarks, and the data sizes at different concurrencies.

4.1 Platforms

To demonstrate the portability of our framework, we chose three diverse HPC platforms for our tests. Table 1 lists details of these HPC systems; note that the number and type of I/O resources vary across these platforms. We also note that the I/O middleware stack is different on Intrepid from that on Hopper and Stampede. On Intrepid, the parallel file system is GPFS, while Hopper and Stampede use the Lustre file system.

4.2 Application I/O Kernels

We chose three parallel I/O kernels to evaluate our autotuning framework: VPIC-IO, VORPAL-IO, and GCRM-IO. These kernels are derived from the I/O calls of three applications, Vector Particle-In-Cell (VPIC) [6], VORPAL [19], and Global Cloud Resolving Model (GCRM) [21], respectively. These I/O kernels represent three distinct I/O write motifs with different data sizes.

VPIC-IO—plasma physics (1D array): VPIC is a highly optimized and scalable particle physics simulation developed by Los Alamos National Lab [6]. VPIC-IO uses the H5Part [5] 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. We extracted 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 float data type. The I/O motif of VPIC-IO is a 1D particle array 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 paper.

VORPAL-IO—accelerator modeling (3D block structured grid): 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 [19]. This benchmark uses H5Block to write non-uniform 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 100x100x60 with different number of processors and the data is written for 20 time steps.

GCRM-IO—global atmospheric model (semi structured mesh): This I/O kernel simulates I/O for GCRM, a global atmospheric circulation model, 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. 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.

4.3 Concurrency and Dataset Sizes

We designed a weak-scaling configuration to test the performance of the auto-tuning framework at three concurrencies, i.e., 128, 2048, and 4096 cores. The amount of data each core writes is constant for a given I/O kernel, i.e., the amount of data an I/O kernel writes increases proportional to the number of cores used. Table 2 shows the sizes of the datasets generated by the I/O benchmarks. The amount of data written by a kernel ranges from 32 GB (with 128 cores) to 1.1 TB (with 4096 cores).

I/O Benchmark	128 Cores	2048 Cores	4096 Cores
VPIC-IO	32 GB	512 GB	1.1 TB
VORPAL-IO	34 GB	549 GB	1.1 TB
GCRM-IO	40 GB	650 GB	1.3 TB

Table 2: Weak scaling configuration for the three I/O benchmarks

4.4 Parameter Space

H5Evolve can take arbitrary values as input for a parameter space. However, evolution of the GA will require more generations to search a parameter space with arbitrary values. To shorten the search time, we selected a few meaningful parallel I/O parameters for all the layers of the I/O stack based on previous research efforts [15] and our experience [7]. We have chosen most of the parameter values to be powers-of-two except some parallel file system parameters. We set the largest parameter value of Lustre stripe count to be equal to the maximum number of available OSTs, which is 156 on Hopper and 160 on Stampede. The GPFS parameters that we tuned are boolean. The process of curtailing parameter values to reasonable ranges based on knowledge of page sizes, min/max striping ranges and powers-of-two values can be done by one who is modestly familiar with the system. And this task needs to be performed only once on a per-system basis. Table 3 shows ranges of various parameter values. A user of our auto-tuning system can set the parameter space by simply modifying the parameter list in H5Evolve. Adding new parameters to search needs simple modifications to H5Tuner. The following is a list of parameters we used as part of the parameter space and their target platforms.

- Lustre (on Hopper and Stampede):
  - Stripe count (`strp_fac`) sets the number of OSTs over which a file is distributed.
  - Stripe size (`strp_unt`) sets the number of bytes written to an OST before cycling to the next OST.
- GPFS (on BG/P Intrepid):
  - Locking: Intrepid has a ROMIO (an MPI-IO implementation [26]) driver to avoid NFS-type file locking. This option is enabled by prefixing a file name with `bglockless:`.

Parameter	Min	Max	# Values
`strp_fac`	4	156/160	10
`strp_unt` / `cb_buf_siz`	1 MB	128 MB	8
`cb_nds`	1	256	12
`align(thresh, bndry)`	(0,1)	(16KB, 32MB)	14
`bglockless`	True	False	2
`IBM_largeblock_io`	True	False	2
`chunk_size`	10 MB	2 GB	25

Table 3: A list of the tunable parameters and their ranges used for experiments in this study. We show the minimum and maximum values for each parameter, with powers-of-two values in between. The last column shows the number of distinct values used for each parameter.

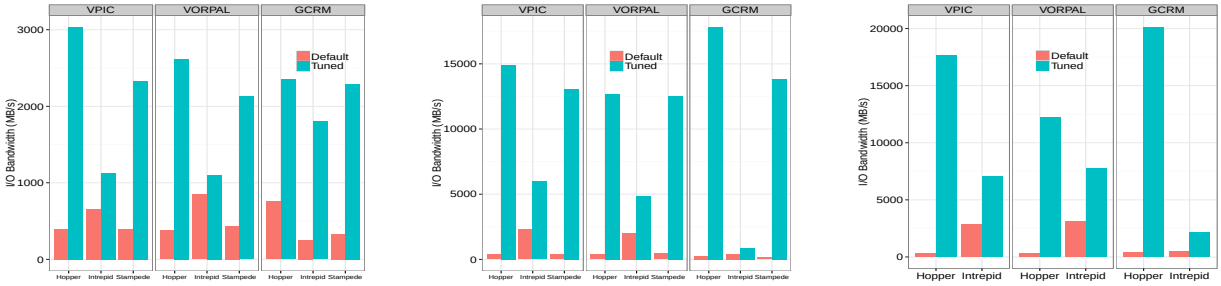
  - Large blocks: ROMIO has a hint for GPFS named `IBM_largeblock_io` which optimizes I/O with operations on large blocks.
- MPI-IO (on all three platforms):
  - Number of collective buffering nodes (`cb_nds`) sets the number of aggregators for collective buffering. On Intrepid, the parameter to set the number of aggregators is `bgl_nodes_pset`.
  - Collective buffer size (`cb_buf_size`) is the size of the intermediate buffer on an aggregator for collective I/O. We set this value to be equal to the stripe size on Hopper and Stampede.
- HDF5 (on all three platforms):
  - Alignment (`align(thresh, bndry)`): HDF5 file access is faster if certain data elements are aligned in a specific manner. Alignment sets any file object with size more than a threshold value to an address that is a multiple of a boundary value.
  - Chunk size (`chunk_size`): In addition to contiguous datasets, where datasets are stored in single blocks in files, HDF5 supports chunked layout in which the data are stored in separate chunks. We used this parameter specifically for the GCRM-IO kernel.

5. RESULTS

Out of the 27 experiments (3 I/O benchmarks `x` 3 concurrencies `x` 3 HPC platforms), we successfully completed 24 experiments in time for this paper. Due to computer resource allocation limitations on Stampede, we could not finish the three 4096-core experiments on that system. However, we expect the performance improvement trends in the remaining runs to be the same as the completed experiments.

In the following subsections, we first compare the I/O rates that our auto-tuning system achieved with those obtained using system default settings. We then analyze the achieved speedup with respect to different platforms, I/O benchmarks, and concurrency/scale in Sections 5.2, 5.3, and 5.4, respectively.

5.1 Auto-Tuning Framework



(a) 128 cores (b) 2048 cores (c) 4096 cores

Figure 6: Summary of performance improvement for each I/O benchmark running on (a) 128 cores, (b) 2048 cores, (c) 4096 cores. The I/O bandwidth axes' scales are different in each of the plots.

Application/ # Cores	Platform	Bandwidth (MB/s)								
		VPIC-IO			VORPAL-IO			GCRM-IO		
		Default	Tuned	Speedup	Default	Tuned	Speedup	Default	Tuned	Speedup
128	Hopper	400	3034	7.57	378	2614	6.90	757	2348	3.10
	Intrepid	659	1126	1.70	846	1102	1.30	255	1801	7.05
	Stampede	394	2328	5.90	439	2130	4.85	331	2291	6.90
2048	Hopper	365	14900	40.80	370	12669	34.16	240	17816	74.12
	Intrepid	2282	5964	2.61	2033	4842	2.38	414	870	2.10
	Stampede	380	13047	34.28	436	12542	28.70	128	13825	107.73
4096	Hopper	348	17620	50.60	320	12192	38.00	413	20136	48.67
	Intrepid	2841	7014	2.46	3131	7766	2.47	523	2177	4.16

Table 4: I/O rate and speedups of I/O Benchmarks with Tuned Parameters over Default Parameters

5.1.1 Tuned I/O Performance Results

The plots in Figure 6 present the I/O rate improvement using tuned parameters that our auto-tuning 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, the 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 6 compares the tuned I/O rate with the default I/O rate for all applications on all HPC systems at 128, 2048, and 4096 core concurrencies. We calculated I/O rate as the ratio of the amount of data a benchmark writes into a HDF5 file at any given scale to the time taken to write the data. The time taken includes the overhead of opening, writing, and closing the HDF5 file. The overhead of HDF5 call interception by H5Tuner, which is included in the time taken, was negligibly small, even at high core count. The I/O rate on the y-axis is expressed in MB/s. Readers should note that the range of I/O rates shown in each of the three plots is different. The measured default I/O rate for a benchmark on a HPC platform is the average I/O rate we obtained after running the benchmark three times. The default experiments correspond to the system default settings that a typical user of the HPC platform would encounter should he/she not have access to an auto-tuning framework.

Table 4 shows the raw I/O rate numbers (in MB/s) of the default and tuned experiments for all 24 experiments. We also show the speedup that the auto-tuned 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 48X, 50X, 70X, and 100X speedups in four cases. We note that the default I/O rates for the Intrepid platform are noticeably higher than those on Hopper and Stampede. Hence, the speedups on Hopper and Stampede with tuned parameters are much larger than those on Intrepid.

5.1.2 Tuned Configurations

Table 5 shows the sets of tuned parameters for all benchmarks on all systems for the 2048-core experiments. Due to space constraints, we cannot present a detailed analysis for all experimental configurations at the other two concurrencies; we generally observed similar trends for the 128-core and 4096-core experiments. First, we note that the tuned parameters are different for all benchmarks and platforms. This highlights the strength of the auto-tuning framework: while I/O experts and sysadmins can probably recommend good settings for a few cases based on their experience, it is hard to encapsulate that knowledge and generalize it across multiple problem configurations.

VPIC-IO and VORPAL-IO on Hopper and Stampede have similar tuned parameters, i.e., `strp_fac`, `strp_unt`, `cb_nds`, `cb_buf_size`, and `align`. On Intrepid, these two benchmarks include `bgl_nodes_pset`, `cb_buf_size`, `bglockless`, `IBM_largeblock_io`, and `align`. On all platforms, GCRM-IO achieved better performance with HDF5's chunking and alignment parameters, and Lustre parameters (stripe factor and stripe size) without the MPI-IO collective buffering parameters. We chose this parameter space for GCRM-IO as Howison et al. [15] demonstrated that the HDF5 chunking provides a significant performance improvement for this I/O benchmark. Moreover, we show that the auto-tuning system is capable of searching a parameter space with multi-

ple HDF5 tunable parameters. On Intrepid, GCRM-IO did not use GPFS tunable parameters because going through HDF5's MPI-POSIX driver avoids the MPI-IO layer, which is needed to set the GPFS parameters. Despite that, HDF5 tuning alone achieved 2X improvement.

I/O Kernel	System	Tuned Parameters
VPIC-IO	Hopper	`strp_fac=128, strp_unt=64MB, cb_nds=1024, cb_buf_size=64MB, align=(0,64K)`
VPIC-IO	Intrepid	`bgl_nodes_pset=512, cb_buf_size=128MB, bglockless=true, large-block_io=false, align=(8K, 1MB)`
VPIC-IO	Stampede	`strp_fac=128, strp_unt=8MB, cb_nds=512, cb_buf_size=8MB, align=(8K, 2MB)`
VORPAL-IO	Hopper	`strp_fac=128, strp_unt=16MB, cb_nds=1024, cb_buf_size=16MB, align=(1K,16K)`
VORPAL-IO	Intrepid	`bgl_nodes_pset=128, cb_buf_size=128MB, bglockless=true, large-block_io=true, align=(8K, 8MB)`
VORPAL-IO	Stampede	`strp_fac=160, strp_unt=2MB, cb_nds=512, cb_buf_size=2MB, align=(8K, 8MB)`
GCRM-IO	Hopper	`strp_fac=156, strp_unt=32MB, chunk_size=(1,26,327680)=32MB, align=(2K, 64KB)`
GCRM-IO	Intrepid	`chunk_size=(1,26,1048760)=1GB, align=(1MB, 4MB)`
GCRM-IO	Stampede	`strp_fac=160, strp_unt=32MB, chunk_size=(1,26,1048760)=1GB, align=(1MB, 4MB)`

Table 5: Tuned parameters of all benchmarks on all the systems for 2048-core experiments

We note some higher-level trends from Table 5. For the same concurrency and with the same benchmark, the tuned parameters are different on various platforms, even with the same parallel file system. For example, although the VPIC-IO benchmark on Hopper and Stampede use the Lustre file system, their stripe settings to achieve the highest performance are different. The tuned parameters can be different on the same platform and at the concurrency for different benchmarks. For instance, the VPIC-IO and VORPAL-IO benchmarks obtain the highest I/O rates with different MPI-IO collective buffering settings and HDF5 alignment settings, whereas their Lustre settings are the same. Similarly, the same benchmark at different concurrencies on the same platform have different tunable parameters. For example, at 128-cores (not shown in the table), VPIC-IO achieves tuned performance with 48 Lustre stripes and 32 MB stripe size, whereas at 2048-cores, VPIC-IO uses 128 stripes with 64 MB stripe size. We analyze these observations further in the following sections.

5.1.3 Partial Tuning

The auto-tuning framework returns a set of tuned parameters across all layers of the I/O stack. In order to assess the impact of each of these parameters (as opposed to the fully tuned set), we devised a set of "Partial Tuning" experiments. These experiments start with the default values for each parameter setting and then iteratively set each parameter to the tuned value returned from the GA result. For example, when we set the Lustre stripe settings (Lustre.Only case in the Figure 7), we disable HDF5 settings. However, on Hopper and Stampede, when we set the Lustre stripe parameters only, the implementation of MPI-IO on these machines also set the MPI-IO collective buffering parameters by default. In our measurements of Lustre.Only setting, we did not isolate the impact of these default MPI-IO collective buffering parameters.

Figure 7 shows the results of these "Partial Tuning" runs at 2048-core scale. There are some gaps between the bars grouped under a particular benchmark. These gaps indicate untested configurations. For example, in evaluating GCRM-IO partial tuning performance, the tuned parameters are not available for MPI-IO collective buffering.

From Figure 7, we can observe that on Hopper and Stampede, Lustre striping (Lustre.Only) has the highest impact on I/O tuning. Readers should note that this setting includes MPI-IO settings that the corresponding platforms set by default. MPI-IO collective buffering (MPI-IO.Collective–.Buffering.Only) also has more performance impact on Hopper and Intrepid than on Stampede for the VPIC-IO and VORPAL-IO benchmarks. HDF5 alignment tuning also has a noticeable impact on performance in most cases. We applied HDF5 chunk size setting only for the GCRM-IO benchmark and it has impact on Hopper, but little on Intrepid or Stampede. Overall, a strategy involving tuning of all parameters outperforms partial tuning in all cases.

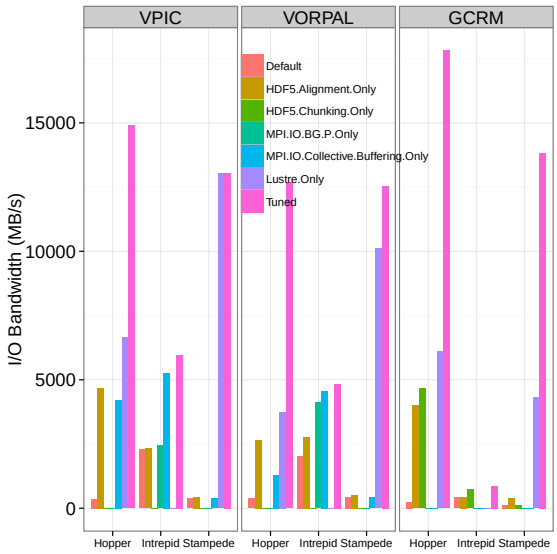


Figure 7: Summary of performance with default, partial runs, and tuned settings for all benchmarks running on 2048 cores

5.2 Tuned I/O performance across platforms

Figure 8(a) shows the distribution of speedups with tuned parameters across Hopper, Intrepid, and Stampede systems representing three different architectures. The speedups are color-coded by I/O benchmark. Overall, the auto-tuning system achieved improved performance on all platforms for all benchmarks. We can observe that the speedups on Hopper and Intrepid are lower than those on Stampede. The speedups on Hopper range from 3.10 to 74.12, with an average of 28.55. Speedups on Intrepid range from 1.30 to 7.05 with an average of 2.76. Speedups on Stampede ranges from

4.85 to 107.73, with an average of 31.39. As mentioned earlier, higher speedups on Stampede are due to poor default performance. In contrast, lower speedups on Intrepid can be attributed to higher default performance. The tuned raw I/O rates on Stampede are similar to those on Hopper.

The aim of this section is to highlight how the auto-tuning framework can deduce high performance configurations for the same application at the same concurrency, but running on different platforms. We highlight this capability by choosing the VPIC-IO benchmark running on 2048 cores on Hopper and Intrepid, and provide some insights on the configuration returned by the GA.

We consider the effect of choosing the collective buffer size parameter for VPIC-IO as illustrated by Figures 9 and 10. On Hopper (Figure 9), multiple buffer size values (equal to the Lustre stripe sizes) obtain good I/O performance, and on average the 32 MB buffer size achieves the best I/O rate (although 64 MB has achieved the peak performance, but at other concurrencies and in this concurrency 32 MB has consistently shown high performance). In the VPIC-IO benchmark, each MPI process writes eight variables and the size of each variable is equal to 32 MB. When the Lustre stripe size is equal to 32 MB, it obtains the best performance on Hopper. The powers-of-two fractions and multiples of 32 MB also obtain reasonably good performance. On Intrepid (Figure 10), we obtain the best performance when the collective buffer size is 128 MB. From Table 5, we can see that the number of *pset* nodes from the tuned parameters is 512, i.e., four MPI processes are being served by one collective buffer. When VPIC-IO writes 32 MB per process, a total of 128 MB data gets collected at the collective buffer node (aggregator) and this node writes data to the file system as one I/O request, which we believe aligns well with the GPFS file system to achieve the best performance. We note that the framework is able to derive these meaningful configurations without detailed prior knowledge of platform specific features.

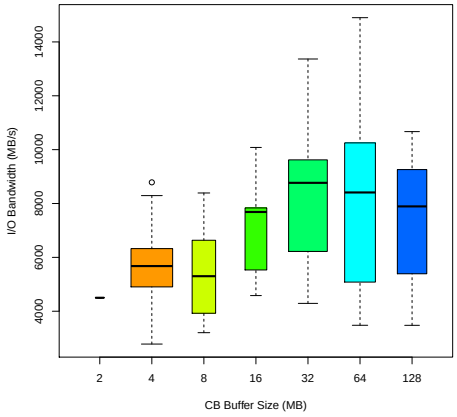


Figure 9: The effect of Hopper's CB Buffer Size on performance of VPIC-IO at 2048 cores

5.3 Tuned I/O for different benchmarks

Figure 8(b) presents the speedup numbers with respect to different I/O benchmarks. Speedups for VPIC range from

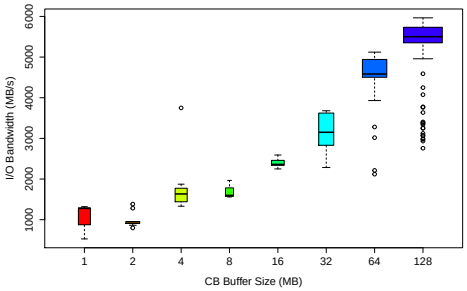


Figure 10: The effect of Intrepid's CB Buffer Size on performance of VPIC-IO at 2048 cores

1.70 to 50.60, with an average of 16.04. Speedups for VORPAL range from 1.30 to 38.00 with an average of 13.69. Speedups for GCRM ranges from 2.10 to 107.73 with an average of 33.50.

We now discuss the configurations returned by the auto-tuning framework for different applications, while holding the platform and concurrency constant. We highlight the VORPAL-IO and GCRM-IO applications, running on 2048 cores of Stampede, and consider tuned the Lustre configurations returned by the GA. Figures 11 and 12 show the impact of Lustre stripe size on the VORPAL-IO and GCRM-IO benchmarks. Both of these benchmarks obtain the highest performance using Lustre stripe count of 160. However, VORPAL-IO obtains the best performance using 2 MB stripe size, whereas GCRM-IO works well using 32 MB stripe size. We note that these different high performance configurations likely result from the different I/O patterns exercised by these benchmarks: VORPAL-IO uses MPI-IO in collective mode whereas GCRM-IO uses MPI-POSIX driver. We are conducting further analysis to better understand why these configurations in particular provide the best I/O performance. This result highlights a strength and a weakness of the auto-tuning approach: the auto-tuning process can produce a good configuration which performs well in practice, but is hard to reason about. On the other hand, it would be very hard for a human expert to propose this configuration in the first place since the interactions in the software stack are very complicated to analyze.

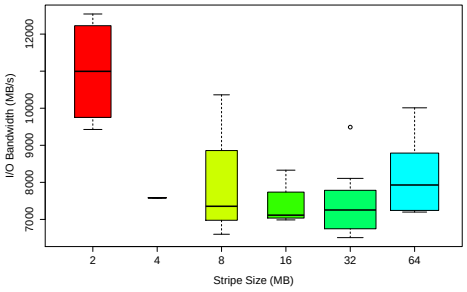
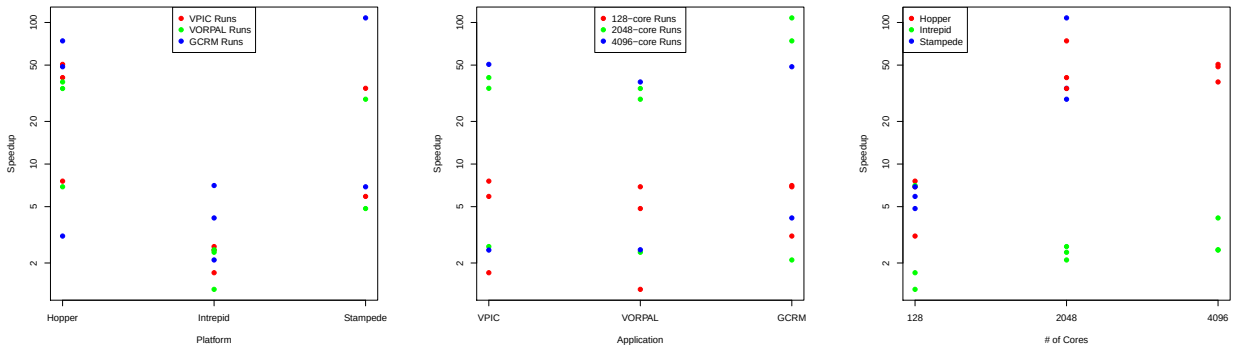


Figure 11: The effect of Lustre Stripe Size value on performance of VORPAL on 2048 cores of Stampede

5.4 Tuned I/O at different scales



(a) Speedups across platforms (b) Speedups across benchmarks (c) Speedups across scale

Figure 8: Speedups with respect to platforms, benchmarks, and scale of the experiments.

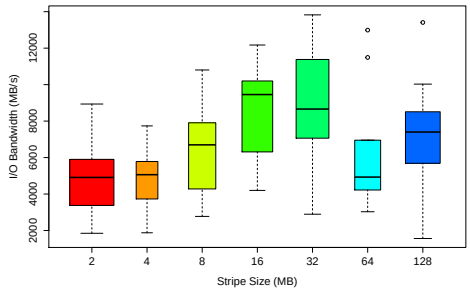


Figure 12: The effect of Lustre Stripe Size value on performance of GCRM on 2048 cores of Stampede

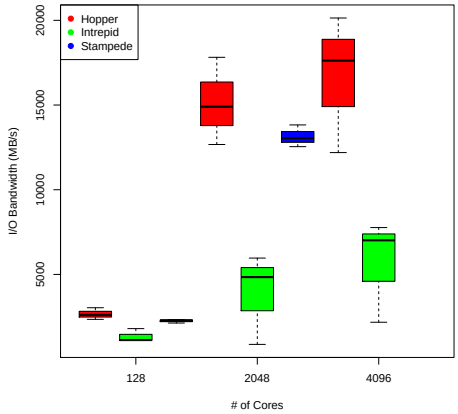


Figure 13: Raw Bandwidth plots and breakdown across scale

Figure 8(c) demonstrates the weak scaling performance obtained by our framework. We observe that the auto-tuning system obtains higher speedups on 2048 and 4096-core experiments. This shows that the default settings on all platforms fare reasonably well at a smaller scale. But, as the concurrency of the application increases, more resources are at stake and that presents more opportunities to optimize the stack.

Figure 13 shows another view of Figure 8(c) with raw I/O rates of benchmarks at various concurrencies grouped based on platform. Each box illustrates the range of I/O rates of the benchmarks. This also illustrates our observation above that auto-tuning is more beneficial at larger scale. This figure also shows that the Lustre-based platforms, i.e., Hopper and Stampede, can achieve higher I/O rates than the GPFS-based platform (Intrepid) with tuning at the concurrencies we experimented. We also show that tuning helps improving performance on BG/P based Intrepid.

6. RELATED WORK

Auto-tuning in computer science is a prevalent term for improving performance of computational kernels. There has been extensive research in developing optimized linear algebra libraries and matrix operation kernels using auto-tuning [30, 14, 16, 29, 32, 12, 31]. The search space in these efforts involves optimization of CPU cache and DRAM parameters along with code changes. All these auto-tuning techniques search various data structure and code transformations using performance models of processor architectures, computation kernels, and compilers. Our study focuses on auto-tuning the I/O subsystem for writing and reading data to a parallel file system in contrast to tuning computational kernels.

There are a few key challenges unique to the I/O auto-tuning problem. Each function evaluation for the I/O case takes on the order of minutes, as opposed to milli-seconds 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 while linear algebra tuning assumes a clean and isolated single node system. The interaction between various I/O parameters and how they impact performance are not very well studied, making interpreting tuned results a complex task.

We use genetic algorithms as a parameter space searching strategy. Heuristics and meta-heuristics have been studied

extensively for combinatorial optimization problems as well as code optimization [23] and parameter optimization [8] 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 [3, 13, 33]. A few recent studies have used genetic algorithms [28] and a combination of approximation algorithm with search space reduction techniques [17]. Both of these are again targeted to auto-tune 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 auto-tuning framework is designed towards 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.

There are a few research efforts to auto-tune and optimize resource provisioning and system design for storage system [1, 2, 24]. In contrast, our study focuses on tuning the parallel I/O stack on top of a working storage system.

Auto-tuning of parallel I/O has not been studied at the same level as the tuning for computation kernels. The Panda project [11, 10] studied automatic performance optimization for collective I/O operations where all the processes 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 and 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 [9]. The optimization approach proposed in this project were applicable to the Panda I/O library, which existed before MPI-IO and HDF5. The Panda I/O is not in use now and the Panda optimization strategy was not designed for current parallel file systems.

Yu et al. [35] characterize, tune, and optimize parallel I/O performance on the Lustre file system of Jaguar, a Cray XT supercomputer at Oak Ridge National Laboratory (ORNL). The authors tuned data sieving buffer size, I/O aggregator buffer size, and the number of I/O aggregator processes. This study did not propose an auto-tuning framework but manually ran a selected set of codes several times with different parameters. Howison et al. [15] also perform manual tuning of various benchmarks that select parameters for HDF5 (chunk size), MPI-IO (collective buffer size and the number of aggregator nodes), and Lustre parameters (stripe size and stripe count) on the Hopper supercomputer at NERSC. These two studies prove that tuning parallel I/O parameters can achieve better performance. In our study we develop an auto-tuning framework that can select tuning parameters.

You et al. [34] proposed an auto-tuning framework for the Lustre file system on the Cray XT5 systems at ORNL.

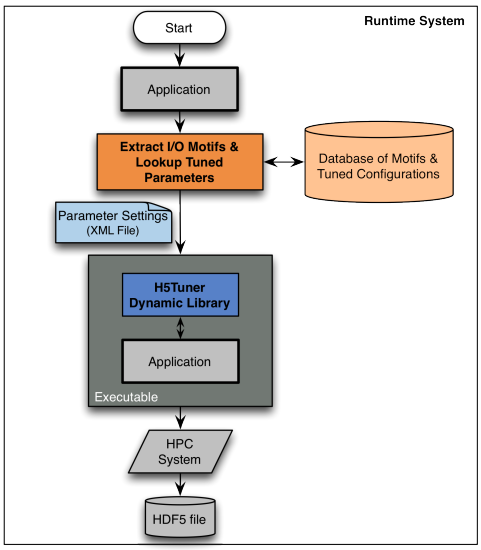


Figure 14: Proposed architecture for the Intelligent Runtime System

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 auto-tuning 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, genetic algorithms, etc. Developing a mathematical model for different systems based on queuing theory can be farther from the real system and may produce inaccurate performance results. In contrast, our framework searches for parameters on real system using search heuristics. A preliminary version of our auto-tuning framework appears in earlier work [4], where we primarily study the performance of our system at a small scale. In this paper, we do a more thorough analysis of the system on diverse platforms, applications, and concurrencies, and conduct an in-depth analysis of resulting configurations.

7. LIMITATIONS AND FUTURE WORK

In this paper, we have focused on developing and testing the auto-tuning system on multiple platforms using different I/O *benchmarks*. We have not addressed the issue of how one can generalize the results from running benchmarks to arbitrary applications. We believe that I/O motifs or patterns are the key to this generalization problem: in the future, we will characterize and enumerate prototypical motifs and use the current auto-tuning framework to populate a database of good configurations for these motifs. We will then implement an intelligent runtime system, which will be capable of extracting I/O motifs from arbitrary applications and consulting the performance database to suggest an optimal I/O strategy. Figure 14 illustrates our proposed architecture for an intelligent runtime system that could address this challenge.

The long runtime of the GA is a potential concern, especially for individual application developers. We believe that the GA runs (on a per-motif basis) can be incorporated into a "health-check" suite run by sysadmins on a system-wide

basis. Thereby, we can both incrementally populate the database of motifs and characterize the performance variability. Following results from our Partial Runs, we are considering designing a genetic algorithm with a custom mutation rate that would initially favor the sensitive parameters on each platform (Lustre or MPI-IO) and then focus on other layers of the I/O stack. We are also looking into machine learning based approaches (such as Gaussian Processes) to intelligently sample the search space, and further reduce the runtime.

Finally, runtime noise and dynamic interference from other users is a fact of life in production HPC facilities. While our auto-tuning framework has presented compelling results, we are assuming that the user will encounter a runtime workload which is comparable to the one encountered during the auto-tuning process. We believe that measuring noise and interference during the tuning process and deriving models for projecting their effect at runtime will be key in tackling this hard problem.

8. CONCLUSIONS

We have presented an auto-tuning framework for optimizing I/O performance of scientific applications. The 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. Perhaps most importantly, we believe that the auto-tuning 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.

9. ACKNOWLEDGMENTS

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy 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. B. Behzad and H. Luu were supported by NSF grant 0938064. The authors would like to acknowledge Wes Bethel, Mohamad Chaarawi, Bill Gropp, John Shalf, and Venkat Vishwanath for their support and guidance.

10. REFERENCES

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, Nov. 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[3] T. Bäck and H.-P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evol. Comput.*, 1(1):1–23, Mar. 1993.

[4] B. Behzad, J. Huchette, H. V. T. Luu, R. Aydt, S. Byna, Y. Yao, Q. Koziol, and Prabhat. A framework for auto-tuning HDF5 applications. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC '13, pages 127–128, New York, NY, USA, 2013. ACM.

[5] E. W. Bethel, J. M. Shalf, C. Siegerist, K. Stockinger, A. Adelmann, A. Gsell, B. Oswald, and T. Schietinger. Progress on H5Part: A Portable High Performance Parallel Data Interface for Electromagnetics Simulations. In *Proceedings of the 2007 IEEE Particle Accelerator Conference (PAC 07). 25-29 Jun 2007, Albuquerque, New Mexico. 22nd IEEE Particle Accelerator Conference, p.3396*, 2007.

[6] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):7, 2008.

[7] S. Byna, J. Chou, O. Rübel, Prabhat, and et al. 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, pages 59:1–59:12, 2012.

[8] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop*, HCW '00, pages 349–, Washington, DC, USA, 2000. IEEE Computer Society.

[9] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel I/O performance optimization using genetic algorithms. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 155 –162, jul 1998.

[10] Y. Chen, M. Winslett, Y. Cho, S. Kuo, and C. Y. Chen. Automatic Parallel I/O Performance Optimization in Panda. In *In Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 108–118, 1998.

[11] Y. Chen, M. Winslett, S.-w. Kuo, Y. Cho, M. Subramaniam, and K. Seamons. Performance modeling for the panda array I/O library. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, 1996.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. 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, pages 4:1–4:12, 2008.

[13] K. Deb, A. Anand, and D. Joshi. A computationally efficient evolutionary algorithm for real-parameter optimization. *Evol. Comput.*, 10(4):371–395, Dec.

2002.

[14] Frigo, Matteo, Johnson, and S. G. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[15] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf. Tuning HDF5 for Lustre File Systems. In *Proceedings of 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, Heraklion, Crete, Greece, Sept. 2010. LBNL-4803E.

[16] B. Jeff, A. Krste, C. Chee-Whye, and D. Jim. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 340–347, 1997.

[17] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. 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, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[18] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 40:1–40:12, New York, NY, USA, 2009. ACM.

[19] C. Nieter and J. R. Cary. VORPAL: a versatile plasma simulation code. *Journal of Computational Physics*, 196:448–472, 2004.

[20] C. S. Perone. Pyevolve: a Python open-source framework for genetic algorithms. *SIGEVOlution*, 4(1):12–20, 2009.

[21] D. Randall, M. Khairoutdinov, A. Arakawa, and W. Grabowski. Breaking the Cloud Parameterization Deadlock. *Bull. Amer. Meteor. Soc.*, 84(11):1547–1564, Nov. 2003.

[22] K. Schulz. Experiences from the Deployment of TACC's Stampede System, March 2013 2013.

[23] K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *Cluster Computing, 2008 IEEE International Conference on*, pages 421 –429, 29 2008-oct. 1 2008.

[24] J. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 21:1–21:16. USENIX Association, 2008.

[25] M. Sweet. Mini-XML, a small XML parsing library, 2003-2011.

[26] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–, Washington, DC, USA, 1999. IEEE Computer Society.

[27] The HDF Group. Hierarchical data format version 5, 2000-2010.

[28] A. Tiwari and J. K. Hollingsworth. Online Adaptive Code Generation and Tuning. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 879–892, Washington, DC, USA, 2011. IEEE Computer Society.

[29] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, 2005.

[30] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[31] S. Williams, K. Datta, J. Carter, L. Oliker, J. Shalf, K. A. Yelick, and D. Bailey. PERI: Autotuning memory intensive kernels for multicore. In *Journal of Physics, SciDAC PI Conference: Conference Series: 123012001*, 2008.

[32] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12, 2007.

[33] A. H. Wright. Genetic Algorithms for Real Parameter Optimization. In *Foundations of Genetic Algorithms*, pages 205–218. Morgan Kaufmann, 1991.

[34] H. You, Q. Liu, Z. Li, and S. Moore. The Design of an Auto-tuning I/O Framework on Cray XT5 System.

[35] W. Yu, J. Vetter, and H. Oral. Performance characterization and optimization of parallel I/O on the Cray XT. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –11, april 2008.

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.