Vidya: Performing Code-Block I/O Characterization for Data Access Optimization

Hariharan Devarajan, Anthony Kougkas, Prajwal Challa, Xian-He Sun
Illinois Institute of Technology, Department of Computer Science
{hdevarajan, akougkas, vchalla3}@hawk.iit.edu, sun@iit.edu

***Abstract*—Understanding, characterizing and tuning scientific applications' I/O behavior is an increasingly complicated process in HPC systems. Existing tools use either offline profiling or online analysis to get insights into the applications' I/O patterns. However, there is lack of a clear formula to characterize applications' I/O. Moreover, these tools are application specific and do not account for multi-tenant systems. This paper presents Vidya, an I/O profiling framework which can predict application's I/O intensity using a new formula called Code-Block I/O Characterization (CIOC). Using CIOC, developers and system architects can tune an application's I/O behavior and better match the underlying storage system to maximize performance. Evaluation results show that Vidya can predict an application's I/O intensity with a variance of 0.05%. Vidya can profile applications with a high accuracy of 98% while reducing profiling time by 9x. We further show how Vidya can optimize an application's I/O time by 3.7x.**

***Keywords*—I/O Systems, I/O profiling, I/O optimization, Code Characterization, High Performance Computing Systems**

I. Introduction

Modern applications are becoming incredibly sophisticated and require extensive tuning for efficient use of computing resources [1]. Understanding the behavior of applications' code is necessary to optimize its performance effectively. Additionally, to assess the impact of tuning efforts, developers need to monitor applications' behavior before and after they make the changes. Performance analysis tools such as OProfiler [2], Jumpshot [3], TAU [4], and STAT [5] are utilized to extract, model and tune the behavior of applications. These profiling tools fall into two major categories [6]: a) *monitoring tools*: expose performance measurements across hardware (e.g., CPU counters, memory usage, etc.,) and visually map them to an application's execution timeline, and b) *tracing tools*: interpret calls and individual events, displaying the results in a structured way inside a log (i.e., trace file). These tools facilitate application profiling and help identify potential performance bottlenecks. However, application performance tuning is burdensome involving a cyclic process of profiling the application, gathering and analyzing collected data, identifying code regions for improvement, and designing and applying optimizations. While profiling memory and communication patterns has been extensively explored [7], [8], the same cannot be said for profiling I/O behavior and subsystems.

In the area of storage systems, researchers are mainly focused on the following methodologies to extract applications' I/O behavior: static analysis tools, such as Darshan [9], which transparently reflects application-level behavior by capturing I/O calls in a per-process and per-file granularity; statistical methods, such as hidden Markov models (HMM) [10], and ARIMA models [11], which focus on spatial and/or temporal I/O behaviors requiring a large number of observations to accomplish good predictions [12]; dynamic pattern-based I/O predictions, such as Omnisc'IO [13], which uses a grammar-based model to predict when future I/O will occur. As I/O is now the biggest challenge in achieving high performance [14], I/O profiling is of the utmost importance in tuning applications on parallel and distributed systems at scale.

Using existing I/O profiling tools imposes several challenges and limitations in characterizing an application's I/O behavior: a) *Time*: static analysis tools pose a significant overhead in the tuning process as they require to execute the application at least once to capture the application's I/O behavior (i.e., offline tracing). This process is expensive in time and resources and can be prohibitive in larger scales. Moreover, erroneous profiling might occur due to performance variability in different scales [15] (i.e., profiled scale vs. actual execution scale). b) *Accuracy*: statistical prediction models capture an application's I/O behavior based on discovered repetitive I/O patterns. The sheer diversity of applications' characteristics, workloads, and complexity makes statistical methods less accurate especially for applications with irregular I/O patterns. c) *Granularity*: existing tools capture an application's I/O behavior in a procedural-level (i.e., which functions perform I/O). This granularity increases user intervention, in the form of manual code inspection, to further understand and analyze the I/O behavior of a given code. d) *Scope*: existing tools capture the I/O behavior on a "per-application" basis without considering system characteristics (i.e., system load, storage devices, networks, etc.), and cross-application I/O interference. This limited scope of analysis leads to skewed I/O profiling results as one application's I/O behavior is affected by another application and/or by the system itself. I/O profiling needs to be fast, light on resources, prediction-accurate, and detailed enough to guide I/O optimization techniques.

In this work, we present Vidya: an I/O profiling framework that can be used to predict the *I/O intensity* of a given application. Vidya performs source code analysis to extract I/O features and identify blocks of code[1] that are contributing the most to the I/O intensity of the application. In the context of this study, an application is I/O intensive when it spends more than 50% of the CPU cycles in performing I/O [16]. Vidya takes into account both the system's characteristics and the application's extracted I/O behavior to apply several code optimizations such as prefetching, caching, buffering, etc. Vidya uses several components to successfully profile an

[1] A code-block can be defined as either a file, a class, a function, a loop, or even a line of code.

application: a Feature Extractor, to capture I/O features from the source code, a Code-Block I/O Characterization (CIOC) formula, that captures the I/O intensity of each code-block, and a Code Optimizer, that can generate optimized code, inject it, and re-compile the application. Vidya offers both developers and system administrators a new, fast, and efficient mechanism to detect possible I/O bottlenecks in finer detail. Vidya's source code analysis approach avoids expensive I/O tracing while maintaining high accuracy in its predictions. Vidya can be used as a guideline to apply optimizations in job scheduling [17], I/O buffering [18], asynchronous I/O [19], and hybrid storage integration [20], [21]. The contributions of this work are:

a) Identifying a wide range of parameters, within a block of code, which contribute towards application's I/O intensity (III).

b) Introducing CIOC, a novel formula that characterizes I/O within and across applications (III-C2).

c) Presenting the design and implementation of Vidya, a modular framework which provides the mechanisms to extract the CIOC score and apply I/O optimizations (IV-A).

d) Evaluating Vidya with applications, improving profiling process by 9x and an I/O time reduction of 3.7x (V).

II. Background and Motivation

A. Modern HPC Applications

Growth in computational capabilities have led to the increase in the resolution of simulation models leading to an explosion in code length and complexity. As we move towards exascale computing, the cost of recording, tuning, and code-updating with advanced models will become ten times higher than the cost of supercomputer hardware [22]. To contain these costs, the exascale software ecosystem needs to address the following challenges: a) software strategies to mitigate high I/O latencies, b) automated fault tolerance, performance analysis, and verification, c) scalable I/O for mining of simulation data. I/O is the primary limiting factor that determines the performance of HPC applications. There is a growing need for understanding complex parallel I/O operations. Applications have become more multi-faceted as programmers and scientists use a variety of languages, libraries, data structures, and algorithms in a single environment. For instance, Montage [23], a mosaic building tool, has 23 million lines of code spanned over 2700 files along with 38 executables. Another example, Cubed-Sphere-Finite-Volume (CSFV) [24], NASA's climate and numerical weather prediction application, has more than a million lines of code in Fortran with 23 simulation kernels and 54 analysis kernels shared across 12 different teams.Moreover, hyper-scalers such as Google deal with an unprecedented scale of projects; Google has a code base of 2 billion lines spanning across all their applications, which is written in more than 50 different languages and frameworks, and shared with more than 2500 engineers [25]. Growth in the complexity of applications strangles the process of tuning. Hence, characterizing and I/O profiling these applications, an already complicated process, is crucial for avoiding performance bottlenecks.

Due to the increasing gap between storage and compute resources [26], researchers perform I/O optimizations by employing several techniques [22]. Dynamic runtime optimization [27] aims at detecting I/O bottlenecks during runtime and redirecting the application from the original to dynamically optimized code. Compiler-directed restructuring [28] aims to reduce the number of disk accesses by using the "disk reuse maximization" technique, a process that restructures the code at compile time. Profiling scientific workflows [29] characterizes workloads using representative benchmarks, and thus, guides researchers and developers as to how to design and build their applications. Lastly, Auto-Tuning [30], an area of self-tuning systems, utilizes a genetic algorithm to search through a large space of tunable parameters 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 I/O calls.

B. Motivation

Monitoring and profiling an application to understand its I/O behavior is a strenuous process. It involves several expensive steps: a) Understanding how the application works: in order to even start the profiling process, we need to know how to execute the application, what is the expected input and output, in what scale, what parameters to use, etc., b) Choosing the appropriate profiler: each profiler extracts certain features with constraints on type of language, detail of extraction, and accuracy of profiling, making it difficult to make the best choice. Additionally, for applications which have several smaller kernels with different languages, finding one profiling tool, which could handle all these kernels together, is not currently feasible. c) Performing the actual I/O profiling: all offline profiling tools require users to first link their application with the tool itself and then execute it to capture the I/O behavior. The execution of most scientific workloads can be a costly task. On the other hand, online profiling tools bind with the application's runtime and predict its I/O behavior based on past observations. Such methods are faster but are typically less accurate, especially for applications with irregular patterns. d) Analyzing the collected profiling data: this stage consists of multiple complex tasks like collecting several logs, analyzing the data, identifying possible bottlenecks, and manually applying potential fixes. It is clear that the above process requires expertise, time, and multiple iterations. We are strongly motivated to address these challenges by introducing Vidya which: a) automatically understands code through parsing without requiring user intervention, b) can capture I/O behavior across multiple source files, executables, and projects, c) does not require additional application execution (i.e., offline) to collect logs, traces, etc., and d) automatically pinpoints which parts of the code can be optimized to avoid performance bottlenecks.

III. Modeling I/O Intensity

A. Application Profiling

To characterize the I/O behavior from source code, we first need to extract and analyze the application's code base. Finding what contributes the most to the I/O intensity of a certain code block can lead to optimization opportunities. The goal of this study is to formally model the set of parameters in a source code that can lead to accurate I/O intensity predictions. To achieve this, we examine a scientific application using existing profiling tools. Specifically, we profile Montage [23], an astronomical image mosaic engine using IOSIG [31], an I/O signature tracing tool, and PAT [32], a flexible performance analysis framework designed for the Linux OS. Using these two tools on the Montage application, we managed to gather

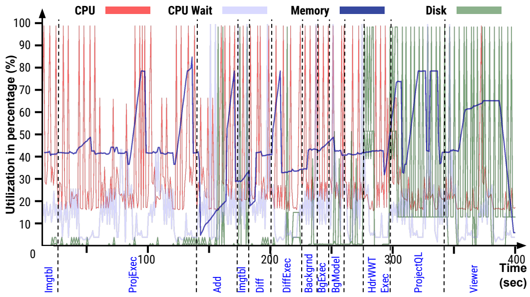


Fig. 1: Montage Analysis

system-level performance metrics including CPU, memory, disk, and network as well as a complete trace of all function calls, I/O calls, and execution timestamps. Lastly, we performed all profiling and analysis on Chameleon systems [33].

Montage workflow description: In our analysis, Montage creates a mosaic with ten astronomy images using 11 analysis executables (i.e., kernels), composed of 36 tasks. The workflow consists of several interdependent phases. In the first phase input files are used to generate multiple intermediate files which are then converted into a mosaic. Montage kernels are then used to compute the list of overlapping images which are analyzed for image-to-image difference in parameters and to generate multiple intermediate files. Lastly, a PNG image is created by combining these intermediate files.

Montage profiling and analysis: Figure 1 shows the behavior of the entire workflow from both system perspective and operations in Montage kernels. The mapping of the system behavior to the application execution timeline, based on I/O tracing, is crucial in understanding which part of the code contributes to the I/O intensity of the entire application. It can be seen that, `mImgtbl` spends most time in compute, `mProjExec` shows that I/O wait time is reduced when data is in memory, `mAdd` depicts a drop in memory after I/O is performed due to flushing of data to disk and finally, kernels such as `mViewer`, and mProjectQL show a lot of repetitive I/O patterns with negligible compute. After analyzing all gathered information regarding the execution of Montage on our testbed machine, we can broadly categorize Montage kernels into three groups: 1) *Compute-intensive:* execution time consists mostly of computation (`mImgtbl`, `mProjExec`, and `mDiff`). 2) *Data-intensive:* execution time consists mostly of I/O operations (`mHdrWWTExec`, `mProjectQL`, and `mViewer`). 3) *Balanced:* execution time including both computation and I/O operations, approximately running for the same period of time (`mAdd`, `mFitExec`, and `mDiffExec`). The data-intensive kernels are of the most relevance to our study as we aim to understand and identify parameters in source code that dictate the I/O behavior of the entire application. Therefore, we manually inspect the source code line by line while referencing the system status, execution, and I/O traces.

B. Parameters Affecting I/O Intensity

Performing source code analysis is a cumbersome process, but it can lead to useful insights as to what code characteristics contribute to an application's I/O behavior. There are some parameters which could be identified easily. These include: the number of I/O operations, the total size of all I/O operations combined, and the number of data sources (i.e., both input and output). These parameters are directly related to a kernel's I/O intensity. We analyzed all the data-intensive and balanced Montage kernels and found the above statement to be true. For instance, in `mHdrWWTExec`, the total amount of I/O was more than 288 MB performed in more than 982 I/O calls in file-per-process fashion. In contrast, `mDiff` only did a few MB in limited I/O calls in a shared file. This adds to our previous kernel categorization (i.e., `mHdrWWTExec` is data-intensive whereas `mDiff` is not).

On the contrary, there were some parameters which could not be identified easily. These include: I/O calls enclosed in a loop (i.e., count of iterative I/O calls in loop structures such as `for` or `while`), the size of data source (i.e., small or large files), and even the I/O interface (i.e., POSIX, MPI-IO, HDF5, etc.,). For example, `mProjectQL` stressed the I/O system even though it performed few MB of I/O. This was counter-intuitive from our previous observations. The explanation for this phenomenon is that the I/O call was enclosed within a loop of hundreds of iterations creating a repetitive I/O pattern. Another example is `mHdrWWTExec`, in which significant time difference was observed when opening a newly created or a large existing file. Lastly, `mViewer` randomly read small portions of multiple files to project the final image which is a known pain point in PFS caused by random small accesses.

Finally, throughout our Montage analysis, we found some code characteristics that can alleviate, to some extent, the I/O intensity of a code-block. These include: asynchronous I/O calls which can be overlapped with computation (i.e., hidden behind compute), and conditional I/O calls caused by code branching (i.e., I/O might not happen based on `if` or `switch` statements). For example, `mAdd` showed less I/O intensity due to certain I/O calls being skipped by an `if` statement. Executing `mAdd` with different input might result in an increase/decrease in the I/O intensity based on the evaluations of conditional statements. Moreover, several code-blocks in `mAdd` had little to no contribution to the I/O intensity due to the asynchronous nature of its I/O calls. Lastly, accessing data from memory (i.e., cached I/O calls) can decrease the percentage of CPU I/O wait time and thus decrease the I/O intensity of the code-block.

In general, applications might demonstrate performance variability based on the input configuration (i.e., number of processes, input files, etc.) and the underlying system specifications (i.e., number of PFS servers, storage device type such as HDD or SSD, etc.). Therefore, besides the application source code parameters, we have also identified system-based parameters that might affect the I/O intensity of a given code-block. We ran some of the Montage kernels several times on top of different storage mediums such as HDD, SSD, and NVMe, and found that the I/O intensity of each kernel slightly changed between runs. The main reason leading to this change is the storage medium characteristics such as bandwidth, latency, and sensitivity to concurrent accesses [18]. We list all parameters identified by our analysis of Montage into a comprehensive set shown in Table I.

C. Dataset and Regression Model

Understanding and analyzing Montage, a scientific simulation consisting of several kernels, each with different behavior, was an experience that motivated us to express all those

TABLE I: Parameters Affecting I/O Intensity

Parameter	Description
P_1	loop count containing I/O calls (i.e., number of iterations)
P_2	number of I/O operations (i.e., count of calls)
P_3	amount of I/O (i.e., size in bytes)
P_4	number of synchronous I/O operations
P_5	number of I/O operations enclosed by a conditional statement
P_6	number of I/O operations that use binary data format
P_7	number of flush operations
P_8	size of file opened
P_9	number of sources/destination files used
P_{10}	space-complexity of code
P_{11}	function stack size of the code
P_{12}	number of random file accesses
P_{13}	number of small file accesses
P_{14}	size of application (i.e. number of processes)
P_{15}	storage device characteristics (i.e. access concurrency, latency and bandwidth)

parameters as a model that can predict the I/O intensity of a given code-block for a certain system. To achieve this, we first collected data from several source codes, spanning a wide variety of applications' classes, devised a regression model, and defined a new formula that encapsulates the I/O intensity in a score. This score can be used to express how I/O intensive a code-block is (i.e., 0 - only compute, 1 - only I/O).

1) Data and Variables: We model all parameters identified in Section III-B into 16 variables. Each variable expresses the parameter's relative contribution to the I/O intensity. The first variable is defined as $X_1 = \frac{P_1}{\max P_1}$. For variables X_2-X_{13}, we define $X_i = \frac{P_i}{\sum P_i}$ (i.e., each parameter value within a code block over the total value of all code-blocks). We also define $X_{14} = \begin{bmatrix} 1 & P_{14} = P_{15} \\ \frac{|P_{14}-P_{15}|}{\max(P_{14},P_{15})} & else \end{bmatrix}$, which matches the application's size to the concurrency of the underlying storage device. Finally, for variables 15 and 16, we define $X_i = \begin{bmatrix} 1, & P_j > 0 \\ 0, otherwise \end{bmatrix}$ where P_j is P_{12} and P_{13} respectively. Furthermore, each variable's value is within [0,1] and measurements are normalized to avoid model skewness due to a variable's scale. Our dataset consists of data collected from a variety of applications: graph exploration kernels [34], sorting programs [35], machine learning kernels [36], I/O and CPU benchmarks [37]. The above variety describes in better detail different families of applications and their respective I/O behavior (i.e., different I/O access patterns, compute to I/O time ratios, number of data sources, etc). For instance, external sort applications access data sequentially whereas breadth-first search demonstrates a random access pattern. Moreover, algorithmic style and complexity differ among different applications. For example, graph algorithms are typically recursive whereas numerical analysis is iterative. Hence, extracting I/O behavior from a diverse set of applications leads to more accurate modeling of I/O intensity.

We divide each source code into blocks. Each code-block can be either a file, a class, a function, a loop, or even a line. We treat every block of code within each application as a record. Each record includes the values of all variables and the *I/O intensity* of the code-block. Note that we define I/O intensity as the ratio of I/O time to the overall execution time of the code-block. We run ten different applications from the above categories collecting measurements for each code-block. Our final dataset consists of 4200 records. Initially, we ran some descriptive statistics to check whether our dataset was in good shape for the model. We tried some simple descriptive statistics which revealed several observations. The distribution of variables X_{1-5}, X_{10}, X_{15}, and X_{16} is normal, and variables X_{6-9} and X_{11-14}, have a Gamma distribution. Also, to investigate the co-linearity between variables, we calculated the covariance matrix for all the variables. The Pearson correlation (p-values) for all variables was less than 0.05, and therefore, all variables have a small correlation coefficient but not significant enough to skew the final model. After this preliminary dataset analysis, we move to the linear regression model. The dataset is representative of a variety of applications and algorithms, and the variables we chose can capture the factors that affect the I/O intensity.

TABLE II: Linear Regression Model

Name	Coefficient	Std. Error	t-ratio
const	-1.99	0.16	-11.92
X_1	0.17	0.33	2.53
X_2	278.80	44.18	6.30
X_3	3706.47	196.81	18.83
X_4	-42612.30	14540.90	-2.93
X_5	Excluded		
X_6	Excluded		
X_7	-10487.80	2511.20	-4.17
X_8	Excluded		
X_9	809.04	93.55	8.64
X_{10}	183996.00	5843.16	31.49
X_{11}	Excluded		
X_{12}	227.98	18.43	12.36
X_{13}	6456.39	2257.85	2.86
X_{14}	0.78	0.10	7.24
X_{15}	Excluded		
X_{16}	Excluded		

Metric	Value
Mean dependent	-6.78
S.D. dep. var	1.69
Sum^2 resid	2675.76
S.E. of reg.	0.79
R^2	0.92
Adjusted R^2	0.91
$F(16, 4183)$	785.13
P-value(F)	0.00

2) Regression Model: In our model, the variable we want to predict is *I/O intensity*. We chose to run a linear regression model since it is simple enough, allows us to determine the overall fit of the model, and to quantify the relative contribution of each of the independent variables (i.e., X_{1-16}) to the dependent variable. During the preliminary analysis of our dataset, we made sure that all assumptions that are required by linear regression held. We do not list all eight assumptions here, but we will mention some. One assumption is that the dependent variable should be measured on a continuous scale, which holds for our I/O intensity variable. Another assumption is that two or more independent variables are either continuous or categorical, which is also true in our case. After checking those assumptions we run the linear regression with the stepwise method on this initial model:

$$Y_m(a, s) = \beta_0 + \sum_{i=1}^{a} \beta_i * X_i + \sum_{i=1}^{s} \beta_i * X_i \tag{1}$$

where Y is the dependent variable I/O intensity, m is the m^{th} code block, a are the application-based variables, s are the system-based variables, β are the coefficients of the regression and X_i is the value of the i^{th} variable. The model summary is shown in Table II.

As it can be seen, variables X_5, X_6, X_8, X_{11}, X_{15}, and X_{16} were excluded from the model. This is because they have a low t-ratio (i.e., the absolute value of t-stat is less than 2). A low t-ratio means higher probability of having zero values as their coefficients. The more significant a t-stat value is, the lower the std error for each predictor is, and a tighter confidence interval for the variable's value will result. Furthermore, out of the included variables, X_4, X_7, and X_{10} are the most significant variables for the model. This is due to their high absolute value of β coefficients. After this analysis, we define the *Code-Block I/O Characterization*

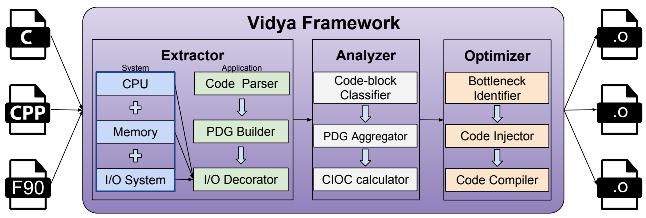


Fig. 2: Vidya framework architecture

(CIOC) formula as the I/O intensity of a code-block relative to that of the applications in the system; it is given by the final model as:

$$CIOC_b = \beta_0 + \beta_1 * X_1 + \beta_2 * X_2 + \beta_3 * X_3 + \beta_4 * X_4 + \beta_7 * X_7 + \beta_9 * X_9 + \beta_{10} * X_{10} + \beta_{12} * X_{12} + \beta_{13} * X_{13} + \beta_{14} * X_{14} \quad (2)$$

3) Model Analysis: There were a few important findings of the linear regression model output: first, the model shows a good *model fit* with the adjusted R^2 value at 92% and a high f-statistic score of 785.13. Additionally, the probability F statistic of 0.00 shows that the model has a reasonable level of confidence that the coefficients of the linear regression model will not range to zero. A high f-stat value also shows that at least one of the variables has the predictive power for determining the I/O intensity. Moreover, the QQ plots of the residual values are normally distributed for our model which shows a good predictive capability of our model.

IV. Vidya Design

A. Overview

Vidya is an I/O profiling framework that can be used to predict the *I/O intensity* of a given application. Vidya's design is inspired by the challenges mentioned in Section II-B. The Vidya framework consists of several tools and modules whose main responsibility is to capture CIOC from the source code and the underlying system. The framework is written in C++ and several scripts are written in Python and Bash. Using Vidya is a simple process: users are expected to provide access to their source code, and Vidya outputs the optimized executables. Our goal when designing Vidya was to build a system that can profile the I/O behavior fast, accurately, and in finer detail. Vidya does not require offline execution of the application to collect profiling data but rather a comprehensive source code analysis to predict the I/O intensity of every code-block and thus of the application. The Vidya framework facilitates the extraction of modeled variables, calculation of CIOC score, and optimization of code. Our prototype framework support C, C++, and Fortran source code. The architecture is presented in Figure 2. There are three main modules in the Vidya framework: Extractor, Analyzer, and Optimizer. All modules work together in harmony to achieve Vidya's objective: profile the I/O of an application, identify optimization opportunities, and apply fixes to the code.

B. Vidya Extractor

Vidya's Extractor derives characteristics from both the system and the application's source code. Its objective is to produce a program dependency graph (PDG) which is a representation of the program's runtime behavior. The algorithm to produce this graph is given in Procedure 1. The extractor collects the following system characteristics: CPU family, instruction length for common operations, number of cores, clock frequency, and cache size. Also, RAM architecture, number of memory banks, frequency, and memory capacity are collected. Lastly, mounting points, file system version, and controller concurrent lanes, bus bandwidth, type of storage device, cache size, device bandwidth, and access latency for a detailed view of the I/O subsystem. The extractor uses a collection of system tools and our scripts to collect the above information. Specifically, the extractor uses the following tools for the respective system component benchmarking: `cpuinfo`, `cpuid`, `meminfo`, `lshw`, `sar`, `mount`, `iostat`, and IOR benchmark [38]. The system specification is important to the profiling of an application since different systems will execute the application differently in terms of performance. Running the same code on a personal computer stresses the subsystems in a different volume than running it on a supercomputer. By knowing the target system, Vidya maps the source code instructions in a predicted execution plan and timeline. For instance, knowing the type of the storage device (i.e., HDD or SSD) can help predict the impact of interference when multiple processes perform I/O simultaneously [39]. Also, it helps to estimate the effect of random access pattern on the overall I/O time (i.e., SSDs are less sensitive to random access than HDDs).

Procedure 1 Program Dependency Graph Builder algorithm

```
Input: cpp file
Parse the file to generate list of nodes
for each node in list of nodes do
    if node is function or variable declaration or call then
        update function and variable maps
for each node in list of nodes do
    for all childnode in node do
        calculate childnode's compute
        if childnode is loop node then
            calculate loop variables
        else if childnode is function node then
            calculate function parameters
            if childnode is IO function then
                extract node IO attributes
                pass I/O information to System Profiler
                enrich node with I/O and system features
add virtual node as root node
```

Code Parser: This module utilizes LLVM's abstract syntax tree (AST) object to parse the code line by line and build a function map. This module extracts several code structures to output the control flow, variable scope and value, and inter-function dependencies. For instance, Figure 3 shows a sample code and its respective parsed output. In this case, the code parser highlights several pieces of structural information from the code such as functions (highlighted in blue), loops and branch statements (highlighted in pink), and I/O calls (highlighted in green).

PDG Builder: This module builds the procedural dependency graph (PDG) using several structures given from the code parser. Specifically, it traverses the function map line by line starting from the entry point (e.g., main function) while maintaining the variables' scopes and values. It uses a bottom-up recursion to build the code dependencies and, finally, output a graph that provides a more accurate view of the application. Figure 3 shows an example of the output produced by the PDG Builder. Each function, branch, and loop is represented by a node in the graph with their parent and children relationships.

Additionally, each node holds the information of the lines it contains as attributes.

I/O Decorator: This module enriches the PDG with the node's I/O features. Specifically, it derives information such as size of I/O, count of I/O calls, number of flush calls, etc. It supports various I/O interfaces as it interprets each API and extracts I/O features based on each implementation (i.e., POSIX, MPI-IO, and HDF5). Figure 3, shows the I/O information at the leaves of the tree. As shown in the figure, we have four I/O functions in the sample code. These are decorated as light blue on the PDG. These nodes contain I/O information such as amount of I/O, source, and, offset.

The extractor faces a significant challenge: several code constructs can be expressed in a different way depending on the language. For instance, `for` loops in C++ can be written in three different ways, or variables can be defined and declared in two different ways, etc. These choices increase the complexity of understanding the code accurately. However, the design of this component is modular and can be extended to other compiled programming languages. In summary, Vidya Extractor's mission is to parse the source code and understand its structure. Other languages, such as Java, offer capabilities to extract an abstract syntax tree (i.e., JavaParser.org, Oracle Java Tools - Parser, etc.), and thus one can similarly build a procedural dependency graph.

C. Vidya Analyzer

Vidya utilizes the enriched PDG from the Extractor to further perform code analysis. The analysis process starts by first classifying code as I/O, propagating all I/O features from the children to parents, and lastly performing CIOC calculations. The analysis is presented in Procedure 2.

Code-block Classifier: This module traverses the PDG produced by the extractor's PDG builder, and classifies the nodes of the graph (i.e., code-blocks) into two categories: compute or I/O blocks. This module marks a code-block as an I/O block when it sees that the block consumes, produces, or mutates data (i.e., *fread(), fwrite(), get(), put(), delete()*, etc.). The granularity of this process is at the node-level. The output is the same PDG but with the block markers. For instance, in Figure 3 node `3`, `5`, `8`, and, `11` are marked as I/O nodes.

PDG Aggregator: Once the I/O blocks are identified, this module performs an enrichment of the graph. When child nodes are marked individually as I/O, this module will aggregate all extracted I/O features into the parent node and does so recursively. Practically, this module calculates the total I/O amount, the total number of I/O operations, and other summary statistics for the entire application. This aggregation step provides a global view of the system as it aggregates the parameters across several such PDGs (per-application). The output is an enhanced PDG decorated with all aggregated values. Figure 3 shows an example of aggregation, as each node inherits its I/O features from its children and hence the root of the PDG consists of all I/O features in all its leaf nodes along with the effect of any branching and/or loop. The process of aggregation can be seen in Procedure 2 at line 7 and 9 where a node inherits its I/O features from its children.

CIOC Calculator: Once the aggregation step is completed, this module calculates CIOC score for all the nodes in the

```
void main(int argc, char *argv[]) {
  int loop_count = std::stoi(argv[1]);
  for (int i = 0; i < loop_count; i++) {
    if (myrank % 2 == 0)
      fwrite(write_buf, write_sz, write_cnt, input_fh);
    else
      fread(read_buf, read_sz, read_cnt, output_fh);
    checkpoint(i);
  }
  sort_temp();
  if (myrank == 0)
    fwrite(result_buf,result_sz,result_cnt,results_fh);
}
int checkpoint(int i) {
  for (int j = 0; j < i; j++)
    fwrite(temp_buf,temp_sz,temp_cnt,intermediate_fh);
}
int sort_temp() {
  std::sort(temp_results.begin(), temp_results.end());
}
```

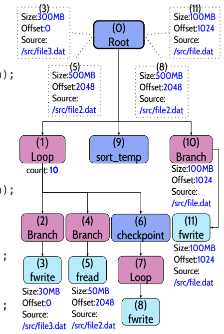


Fig. 3: PDG example.

Procedure 2 Vidya Analysis Algorithm

1: **procedure** PDG_ANALYSIS($node$)
2: $total\_node\_io \leftarrow FALSE$
3: **for** each childnode in node **do**
4: $childnode \leftarrow PDG\_Analysis(childnode)$
5: **if** $childnode\ has\ io$ **then**
6: $total\_node\_io \leftarrow TRUE$
7: append vars from characteristics of childnode
8: **if** $node\ has\ io$ **then**
9: $append\ vars\ from\ characteristics\ of\ node$
10: $total\_node\_io \leftarrow TRUE$
11: **if** $total\_node\_io = TRUE$ **then**
12: calculate CIOC score of node
13: **else**
14: $node's\ CIOC\ score \leftarrow 0$
15: **return** node

PDG. This CIOC calculation has two stages. First, calculation of the variables using the I/O features extracted. Second, prediction of the I/O intensity using CIOC score. The scoring on the graph is done bottom up to encompass the data intensity at all levels as seen in Procedure 2. For instance, in Figure 3 the scoring is done from node `11` to `10` and then these features are aggregated at `0`. Once the scoring is done, this module translates the CIOC score as metadata for each application, which is then written out as profiling logs.

The major challenge that the analyzer addresses is the granularity of the profiling analysis. Static analysis profilers such as Darshan can only address I/O profiling in function level (i.e., simply answer which functions perform I/O). In contrast, Vidya's analyzer module allows us to identify the I/O source in finer granularity (i.e., line, function, class, file, etc.). This granularity comes from the combination of code features that CIOC score encapsulates and the PDG itself.

D. Vidya Optimizer

The central question this module faces is what optimizations need to be applied and when. Vidya includes strategies which utilize the application's metadata to apply optimizations accurately. For instance, if an application's CIOC score is high due to read calls, Vidya identifies the event, and uses the prefetcher to pro-actively fetch data before the read operation. The prefetcher requires two pieces of metadata information from the PDG: a) the details of the read operation within the code-block (this is provided as the PDG keeps track of offset and size of each I/O that occurs on a file), and b) where to place data from the asynchronous prefetching call. The PDG maintains the order of I/O and non-I/O calls, and thus it can accurately predict the branch of code where this prefetching should occur. Based on these information, the

```cpp
void main(int argc, char *argv[]) {
  int loop_count = std::stoi(argv[1]);
  for (int i = 0; i < loop_count; i++) {
    std::sort(temp_results.begin(),
              temp_results.rbegin()-i);
    fread(read_buf, read_sz,
          read_cnt, input_fh);
  }
  if (myrank == 0)
    fwrite(result_buf,result_sz,
           result_cnt,results_fh);
}
```

(a) Before optimization

```cpp
void main(int argc, char *argv[]) {
  int loop_count = std::stoi(argv[1]);
  for (int i = 0; i < loop_count; i++) {
    vidya::async_prefetch(read_buf, read_sz,
                          read_cnt, input_fh);
    std::sort(temp_results.begin(),
              temp_results.rbegin()-i);
    vidya::buffer_read(read_buf, read_sz,
                       read_cnt, input_fh);
  }
  if (myrank == 0)
    fwrite(result_buf,result_sz,
           result_cnt,results_fh);
}
```

(b) After optimization

Fig. 4: Vidya Optimizer - Automatic Code Injection

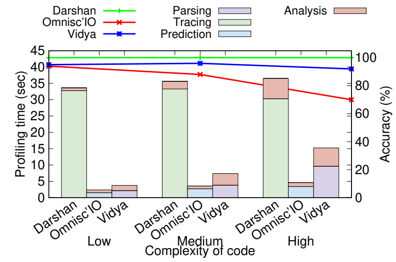


Fig. 5: Code Complexity

prefetcher sub-module injects prefetching calls (i.e. performing I/O asynchronously and buffering the data), and modifies the actual read call to use buffered I/O. Figure 4 demonstrates snippets of code and their optimized counterparts.

Bottleneck Identifier: This module utilizes the profiling logs, generated from the analyzer, to automatically understand the I/O behavior of applications. The CIOC score represents the I/O intensity of a code-block. Hence, the bottleneck identifier can pinpoint which blocks of code have the highest CIOC score, and by looking at the variables it can deduce the cause of that score. Once the cause of the score is found, this module marks the nodes with flags for what should be optimized. For example, when variable X_7 has a high score, it means that excessive flushing is taking place, and thus an optimization opportunity arises. This module will first check if *fopen()* was executed with the synchronous mode flag on. If so, it marks the code with a suggested optimization; in this case, it marks the removal of flush operations.

Code Injector: This module takes as input PDG nodes marked by the bottleneck identifier with potential optimizations. Based on the markers, this module performs code injections or modifications to perform the required optimization. Following our example from the previous module, if a node was flagged for removal of flushing operations, it will remove the lines of code that cause flushing. These updates in the code are done using LLVM's API (e.g., `remove_line`, `write_line`, etc.).

Code Compiler: Once the optimizations are injected, the modified source code is submitted for compilation. It compiles the given source code, along with its dependencies, using LLVM and produces the object files for all the applications.

API: Vidya offers a simple API. Vidya can be configured to either apply optimizations automatically or give some control to the user. Examples from Vidya's API include: *vidya::buffer_read()*: reads data from buffer, *vidya::async_prefetch()*: prefetches requested data asynchronously, *vidya::cache_to_buffer()* caches data into buffer, *vidya::evict_cache_line()* removes passed cache line, etc.

E. Design Considerations

Application input arguments during runtime: Input arguments can be passed to a program during execution. These inputs can not only alter programs execution flow but its behavior too. To solve this, Vidya maintains a heap of global variables and a function-level stack of local-variables. The program arguments are given as input to the Vidya profiler along with the source code which are then treated as inputs to the root function (e.g., `main()`) and used in the analysis of the source code.

I/O on pre-existing files: Many applications' I/O behavior is determined by the dataset size (i.e., reading and writing to big or small datasets). The dataset size can be dynamically used in a program and determine the I/O behavior of the entire application. Vidya uses its system profiler to not only collect system information but also information about the files that the application operates on. Specifically, based on the files opened, Vidya performs file stat operations to determine the file size during open(), and then based on application's I/O operations such as fwrite(), Vidya keeps track of file size in the PDG.

Code Branching: Most applications have conditional branching of code which leads to multiple execution paths. These execution paths determine the different behavior of the applications based on execution parameters. Vidya treats each branch as a possible path in the PDG. This allows Vidya to treat each portion of code separately and predict I/O intensity in all the possible branches of the application.

V. Evaluation

A. Platform and applications

All experiments were conducted on Chameleon systems [33]. More specifically, we used the bare metal configuration with 32 client nodes and 8 server nodes for Parallel File System. Each node has a dual Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz (i.e., a total of 48 cores per node), 128 GB RAM, 10Gbit Ethernet, and a local 200GB HDD. The list of applications used are CM1 [40], WRF [41] and Montage [23], which are real-world codes representative of an application running on current supercomputers. They have been used for NCSI's Kraken and NCSA's Blue Waters for CM1 and WRF ANL's Intrepid and Mira for Montage. Additionally, we use benchmarks such as IOR [38], which is designed to measure I/O performance at both the POSIX and MPI-IO level, and Graph500 [42]. Finally, we compare Vidya's results with Darshan [9], which analyzes applications based on their runtime behavior, and Omnisc'IO [13], which uses grammar-based models to predict future I/O.

B. Profiling Performance

In this series of tests, we evaluate the profiling performance of Vidya and compare the results with Darshan and Omnisc'IO.

1) Code complexity: In this first test, we use I/O benchmark (Synthetic workload generator) to test the time taken by all solutions to complete the profiling and analysis. The workload generator can tune the level of code complexity, which is a key factor in the speed of profiling. Specifically, the benchmark performs 4 GB of I/O (i.e., 64 files of 64 MB). All operations are performed on an HDD (i.e., 125MB/s read-write

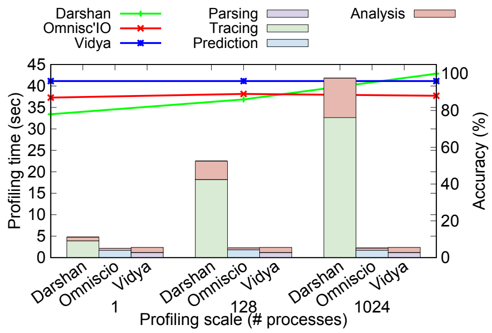


Fig. 6: Profiling Granularity

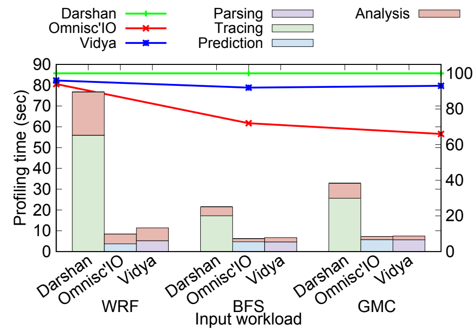


Fig. 7: Workflow Irregularity

bandwidth). We divide the benchmark into three complexity modes: *Low*, where each file is written serially one after another by a `for` loop inside a function using POSIX interface, *Medium*, where each file is written by separate functions using MPI Independent I/O, and *High*, where each file is grouped into categories and the I/O is performed by several classes utilizing the HDF5 [43] data format. Darshan follows a tracing approach whereas Omnisc'IO follows a predictive grammar-based approach. In contrast, Vidya implements a novel way to profile an application by parsing the source code and analyzing the syntax trees. The more complex a source code is, the more sensitive each tool can be regarding its profiling performance. In Figure 5, we present the results of profiling our benchmark. As it can be seen, Darshan first executes the benchmark while collecting traces and then analyzes the logs. The complexity of code does not affect Darshan's profiling performance. Moreover, since Darshan is collecting execution traces, the accuracy of profiling the I/O intensity is always 100%. Omnisc'IO trades accuracy for performance. It completes the profiling during runtime, adding only a small overhead. This results in performance gains of 15x, 11x, and 9x for Low, Medium, and High code complexity respectively. However, the accuracy of its predictions is 94% for Low code complexity and drops to only 70% for High, since the extent of the grammar it builds is directly proportional to the code complexity. On the other hand, Vidya's source code analysis approach balances profiling speed and profiling accuracy. In our test, Vidya achieved a profiling performance of 3.77 seconds for Low, 7.38 seconds for Medium, and 15.24 seconds for High code complexity, which is 8.9x, 4.8x, and 2.4x faster than Darshan respectively. Furthermore, Vidya was 31% more accurate than Omnisc'IO in its I/O intensity predictions, even though the later was still faster while profiling the benchmark. Note that, as the code complexity increases so does Vidya's profiling cost (i.e., parsing and analyzing). In summary, Vidya achieves high prediction accuracy and profiling performance.

2) Profiling Granularity: Profiling tools provide an understanding of the I/O behavior of an application. However, the scale with which we profile an application and the scale with which we run it might be different which might, in turn, lead to different profiling conclusions. There is a mismatch between tracing results and the actual I/O behavior due to the scale difference. Static analysis tools based on tracing, such as Darshan, are very susceptible to this issue. In this next test, let us assume that the actual execution scale is 1024. We change the granularity of profiling of CM1 from a single process, to 128, and 1024 processes, and we measure the overall profiling cost, expressed in time, and profiling accuracy, expressed in percentage. Note that, CM1 performs file per process hence this reflects a strong scaling test. As it can be seen in Figure 6, when profiling CM1 with 1 process, Darshan took 4.81 seconds, with 128 processes 22.48 seconds and with 1024 processes 41.809 seconds. The profiling result we got from analyzing each case was different. When profiling CM1 with a single process, the predicted I/O behavior was different from the actual behavior with an accuracy of only 78%. When we profile the application with the same number of processes compared to the actual execution (i.e., 1024 in this test), then Darshan produced 100% accurate results. Online profiling tools based on predictions, such as Omnisc'IO, do not suffer from this limitation. In our test, Omnisc'IO showed a stable profiling performance of 2.1 seconds with an accuracy of 88%. Similarly, Vidya, which relies on the analysis of the source code, takes into consideration the scale (i.e., the granularity of the test) and produces more accurate results without executing the application. It combines the speed of an online predictive tool such as Omnisc'IO with the accuracy of a tracing tool. In this test, Vidya demonstrated a profiling speed of 2.37 seconds, on average, and an accuracy of 96%.

3) Workflow irregularity: In this test, we evaluate how profiling performance is affected by the irregularity of I/O patterns. Specifically, for our driver applications for this test we use: a) WRF, which demonstrates a regular I/O access pattern, b) Graph500's breadth-first search (BFS), and c) Graph500's graph min cut (GMC) kernel, which both demonstrate a highly irregular I/O pattern. In Figure 6, we present the results of profiling of these applications. We execute all programs with 1024 MPI ranks, and we direct the I/O to our PFS of 8 servers. As it can be seen, Darshan takes a long time to profile the applications but maintains an accurate picture of the I/O behavior. Darshan is 100% accurate but is the slowest profiling solution. On the other hand, Omnisc'IO does not require any extra offline profiling processes; however, Omnisc'IO relies on the predictive power of its grammar which is strong for regular patterns but suffers on irregular applications. This behavior is reflected in our results where we can observe that the accuracy of Omnisc'IO drops to 66% for GMC which is known to have irregular patterns. Vidya, on the other hand, does not suffer by the irregularity of the input workload as it parses the code to understand the I/O. The profiling speed is not affected by this, and Vidya achieves similar numbers as Omnisc'IO. In summary, Vidya aims to optimize both speed and accuracy.

C. I/O Optimization using Profiling

In this last set of tests, we aim to evaluate the potential of an application profiler in optimizing I/O performance. Most of the optimization techniques rely on understanding the behavior of an application and thus correctly identifying optimization opportunities. In the first test, in Figure 8 (a), we first profile the application to identify when and how much to prefetch data to help reading operations. More accurate and complete the profiling improves our chances to prefetch data optimally via active storage [44], [45]. Darshan is the most accurate and optimized the I/O time by more than 4.3x. However, Darshan requires significant time to first profile the application. In contrast, Omnisc'IO only adds minor overheads in the execution time while building the grammar which is then used to predict when and what to prefetch. It optimizes the I/O performance of WRF by 2x. However, for BFS, which has irregular patterns, Omnisc'IO could not boost I/O time more than 20%. On the other hand, Vidya successfully identifies exactly what to prefetch and by adding slightly more time in source code analysis, it can offer almost the optimization boost of Darshan without the extra time to collect the traces. Specifically, it achieved **3.7x** performance gains and spent only a couple of seconds profiling. A similar outcome can be observed in the next and final test where we turn on or off the write-cache. Effectively, an accurate profiler will tell the optimization when to turn on the cache and avoid scenarios where the cache is trashed. Vidya outperformed both Darshan, by 2.7x, and Omnisc'IO, by 50%.

VI. Discussion

1) Measurement Vs Prediction: Application profilers can be classified based on their profiling approach: a) Measurement based: these execute the application to measure the application's behavior, and, b) Prediction-based: these try to predict the application's behavior without executing it. Clearly, it is a trade-off between accuracy and cost of profiling. Vidya aims to balance these two by trying to estimate the application's I/O behavior through its source-code instead of waiting for observations or performing online predictions. This trade-off is evident throughout the evaluation section.

2) Source Code Analysis Limitations: Source-code analysis is extremely powerful as shown in this work. But it suffers from some limitations.

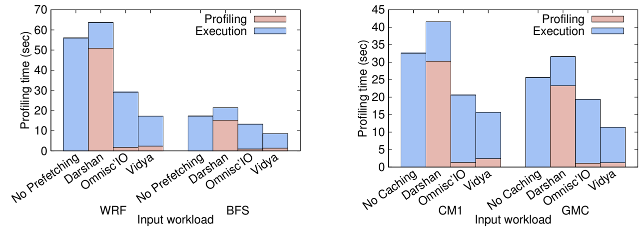
Dynamic runtime flows: Applications whose runtime dependency is based on external files is hard to detect and handle. These runtime flows can be handled in the PDG by simulating the code's execution. This is not only difficult to implement but also costly as it requires actual running of pieces of code.

Applications with automatic code generation: Applications which generate code dynamically during execution based on branches are hard to detect and simulate. Such applications form dynamic PDG on runtime, and therefore, any static analysis approach would be extremely error prone.

These cases are extremely hard for any source-code analysis approach to handle. We recommend solving these problems in two stages. a) Identification: Code patterns like dynamic code generations or branches dependent on data read from files can be detected in the code, and, b) Simulation: such pieces of code should be accurately simulated with extreme care.

VII. Related Work

Static I/O characterization tools: Carns et al. used Darshan [9] to analyze the production I/O activity on Intrepid.



(a) Prefetching On/Off (b) Write-cache On/Off

Fig. 8: Optimization

Darshan captures application-level access pattern information per-process and per-file granularity. Byna et al. have utilized tracing and characterization of I/O patterns as a means to improve MPI-IO prefetching [12]. Their method consists of running an application job once to generate a complete MPI-IO trace, post analyzing the trace to create an I/O signature, and then using the signature to guide prefetching on subsequent jobs. Both of these works suffer from these challenges: a) applications need to be run to get their I/O behavior which would not be possible for exascale systems, and, b) failure to have a global view in a multi-tenant system

Prediction Modeling: Sequitur is designed to build a grammar from a sequence of symbols and has been used mainly in the area of text compression [46], natural language processing, and macromolecular sequence modeling [47], which have repetitive periodic I/O. Tran has investigated prediction of temporal access pattern and Reed [11] using ARIMA time series to model inter-arrival time between I/O requests. Such statistical models need a large number of observations to converge to proper representation and, thus, right predictions. Dorier et al. [13] models the behavior of I/O in any HPC application and using this model it predicts the future I/O operations. These works cannot handle modern complex applications with irregular patterns as they depend on repetitive I/O behavior.

The HPC community has produced a wide variety of modern tools for generating traces of individual I/O operations including HPCT-IO [48], Jumpshot [3], [49], TAU [4], and STAT [5]. However, these tools focus primarily on in-depth analysis of individual application runs rather than long-running workload characterization.

VIII. Conclusions

In this paper, we propose Vidya, an I/O profiling framework that can be used to predict the *I/O intensity* of a given application. Additionally, we present a code-block formula for predicting I/O intensity of application called CIOC. We show how different code-block level parameters can affect I/O and how these parameters can be used to predict I/O intensity without executing the application. Results show that Vidya can make profiling of applications faster by 9x while having a high accuracy of 98%. Furthermore, we show, Vidya can be used to optimize applications up to 3.7x. To the best of our knowledge, Vidya is the first work that leverages program structure to predict I/O performance and optimize it. As a future step, we can use this approach to perform automated runtime I/O optimizations on applications at a system level.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grants no. CCF-1744317, CNS-1526887, and CNS-0751200.

REFERENCES

[1] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 155–164.

[2] W. Cohen, "Multiple Architecture Characterization of the Build Process with OProfile," http://oprofile.sourceforge.net, 2003, [Online; accessed April-2018].

[3] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[4] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[5] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2007, pp. 1–10.

[6] S. Benkner, F. Franchetti, H. M. Gerndt, and J. K. Hollingsworth, "Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401)," in *Dagstuhl Reports*, vol. 3. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[7] Y. Jin, "Numatop: A tool for memory access locality characterization and analysis," http://oprofile.sourceforge.net, 2013, [Online; accessed April-2018].

[8] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[9] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *International Conference on Cluster Computing and Workshops (CLUSTER)*. IEEE, 2009, pp. 1–10.

[10] J. Oly and D. A. Reed, "Markov model prediction of I/O requests for scientific applications," in *Proceedings of the 16th international conference on Supercomputing*. ACM, 2002, pp. 147–155.

[11] N. Tran and D. A. Reed, "Automatic ARIMA time series modeling for adaptive I/O prefetching," *IEEE Transactions on parallel and distributed systems*, vol. 15, no. 4, pp. 362–377, 2004.

[12] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proceedings of the ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 44.

[13] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "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*. IEEE Press, 2014, pp. 623–634.

[14] B. Dong, X. Li, L. Xiao, and L. Ruan, "A new file-specific stripe size selection method for highly concurrent data access," in *ACM/IEEE 13th International Conference on Grid Computing (GRID)*. IEEE, 2012, pp. 22–30.

[15] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the IO performance of petascale storage systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2010, pp. 1–12.

[16] T. Hey, S. Tansley, K. M. Tolle *et al.*, *The fourth paradigm: data-intensive scientific discovery*. Redmond, WA: Microsoft Research, 2009, vol. 1.

[17] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: A Heterogeneous-aware Multi-tiered Distributed I/O Buffering System," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 219–230. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208059

[18] ——, "Hermes: A Heterogeneous-aware Multi-tiered Distributed I/O Buffering System," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 219–230. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208059

[19] ——, "IRIS: I/O Redirection via Integrated Storage," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: ACM, 2018, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/3205289.3205322

[20] ——, "Enosis: Bridging the semantic gap between file-based and object-based data models," in *Data-Intensive Computing in the Clouds(Datacloud'17), 8th International Workshop on*. Denver, CO: ACM SIGHPC, 2017.

[21] ——, "Syndesis: Mapping objects to files for a unified data access system," in *Many-Task Computing on Clouds, Grids, and Supercomputers(MTAGS'17), 9th International Workshop on*. Denver, CO: ACM SIGHPC, 2017.

[22] A. Geist and R. Lucas, "Major computer science challenges at exascale," *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 427–436, 2009.

[23] IRSA, "Montage - An Astronomical Image Mosaic Engine," http://montage.ipac.caltech.edu/docs/m101tutorial.html, 2017, [Online; accessed April-2018].

[24] Rotman and GOCART, "Cubed-sphere finite-volume dynamic core(fvcore)," https://www.gfdl.noaa.gov/fv3/, 2001, [Online; accessed April-2018].

[25] C. Metz, "GOOGLE is 2 billion lines of code and it's all in one place," https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/, 2015, [Online; accessed April-2018].

[26] A. Kougkas, H. Eslami, X.-H. Sun, R. Thakur, and W. Gropp, "Rethinking key–value store for parallel i/o optimization," *The International Journal of High Performance Computing Applications*, vol. 31, no. 4, pp. 335–356, 2017.

[27] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu, "Design and implementation of a lightweight dynamic optimization system," *Journal of Instruction-Level Parallelism*, vol. 6, no. 4, pp. 332–341, 2004.

[28] M. T. Kandemir, S. W. Son, and M. Karaköy, "Improving I/O Performance of Applications through Compiler-Directed Code Restructuring." in *In Proc. 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 159–174.

[29] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

[30] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol, M. Snir *et al.*, "Taming parallel I/O complexity with auto-tuning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 68.

[31] Y. Yin, S. Byna, H. Song, X. H. Sun, and R. Thakur, "Boosting Application-Specific Parallel I/O Optimization Using IOSIG," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid)*, May 2012, pp. 196–203.

[32] Intel, "Performance Analysis Tool," https://github.com/intel-hadoop/PAT, 2015, [Online; accessed April-2018].

[33] Chameleon.org, "Chameleon system," https://www.chameleoncloud.org/about/chameleon/, 2017, [Online; accessed April-2018].

[34] H. Sato, "A Graph500 benchmark implementation," https://github.com/htsst/netalx, 2015, [Online; accessed April-2018].

[35] S. University of Connecticut, "External Sorting Library Project," http://lib-ex-sort.sourceforge.net/, 2010, [Online; accessed April-2018].

[36] N. Sarten, "A C++11 k-means clustering implementation," https://github.com/genbattle/dkm, 2017, [Online; accessed April-2018].

[37] CORAL, "LLNL,CORAL Benchmark Codes," https://asc.llnl.gov/CORAL-benchmarks/, 2014, [Online; accessed April-2018].

[38] LLNL, "Ior benchmark," https://goo.gl/YtW4NV, 2017.

[39] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun, "Leveraging burst buffer coordination to prevent i/o interference," in *e-Science (e-Science), 2016 IEEE 12th International Conference on*. IEEE, 2016, pp. 371–380.

[40] G. Bryan, "UCAR CM1 atmospheric simulation," 2016. [Online]. Available: http://www2.mmm.ucar.edu/people/bryan/cm1/

[41] W. Research and F. Model, "WRF," 2016. [Online]. Available: http://www.wrf-model.org/index.php

[42] G. . Benchmarks, "Graph500," 2017. [Online]. Available: https://graph500.org/

[43] M. Folk, A. Cheng, and K. Yates, "Hdf5: A file format and i/o library for high performance computing applications," in *Proceedings of Supercomputing*, vol. 99, 1999, pp. 5–33.

[44] H. Devarajan, A. Kougkas, X.-H. Sun, and H. Chen, "Open ethernet drive: Evolution of energy-efficient storage technology," in *Proceedings of the ACM SIGHPC Datacloud'17, 8th International Workshop on Data-Intensive Computing in the Clouds in conjunction with SC'17*, 2017.

[45] A. Kougkas, A. Fleck, and X.-H. Sun, "Towards energy efficient data management in hpc: the open ethernet drive approach," in *Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS), 2016 1st Joint International Workshop on*. IEEE, 2016, pp. 43–48.

[46] J. C. Kieffer and E.-H. Yang, "Grammar-based codes: a new class of universal lossless source codes," *IEEE Transactions on Information Theory*, vol. 46, no. 3, pp. 737–754, 2000.

[47] C. G. Nevill-Manning, "Inferring sequential structure," Ph.D. dissertation, University of Waikato, 1996.

[48] S. Seelam, I.-H. Chung, D.-Y. Hong, H.-F. Wen, and H. Yu, "Early experiences in application level I/O tracing on Blue Gene systems," in *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2008, pp. 1–8.

[49] A. Chan, W. Gropp, and E. Lusk, "An efficient format for nearly constant-time access to arbitrary time intervals in large trace files," *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, 2008.