The dissertation of Seong Jo Kim was read and approved\(^1\) by the following:

Mahmut T. Kandemir  
Professor of Computer Science and Engineering  
Dissertation Adviser  
Chair of Committee

Mary J. Irwin  
Professor of Computer Science and Engineering  
Evan Pugh Professor and A. Robert Noll Chair

Padma Raghavan  
Distinguished Professor of Computer Science and Engineering  
Associate Vice President for Research and Director of Strategic Initiatives  
Director of Institute of CyberScience

Dinghao Wu  
Assistant Professor of Information Science and Technology

Rajeev Thakur  
Special Member  
Senior Computer Scientist, Argonne National Laboratory  
Deputy Director of Mathematics and Computer Science Division, Argonne National Laboratory

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

\(^1\)Signatures on file in the Graduate School.
Abstract

Efficient execution of large-scale scientific applications requires high-performance computing systems designed to meet the I/O requirements. To achieve high-performance, such data-intensive scientific applications use multiple layers of I/O software stack that consists of high-level I/O libraries such as PnetCDF and HDF5, the MPI library, and parallel file systems. To design efficient parallel scientific applications, understanding the complicated flow of I/O operations and the involved interactions among the libraries is quintessential. Such comprehension helps identify I/O bottlenecks and thus exploits the potential performance in different layers of the storage hierarchy.

To trace the execution of I/O operations and to understand the complex interactions in the I/O stack, we have designed and implemented a parallel I/O profiling and visualization framework for high-performance storage systems, IOPro. IOPro automatically generates an instrumented I/O stack, runs applications on it, and visualizes detailed statistics in terms of user-specified metrics of interest. Next, we introduce a dynamic performance visualization and analysis framework for parallel I/O, called IOPin. IOPin performs the instrumentation with minimal overhead in the binary code of the I/O stack at runtime and provides the language independent instrumentation targeting specific applications written in C/C++ and Fortran. Furthermore, it requires neither source code modification nor recompilation of the application and the I/O software stack components. Lastly, we propose an automatic parallel I/O code generation and optimization framework for HPC applications, called IOGenie. Using a graphical user interface,
our tool takes high-level annotations for I/O as input, analyzes the given options, and generates optimized I/O code that effectively exercises the underlying I/O stack.

Overall, this thesis proposes three frameworks, IOPro, IOPin, and IOGenie. IO-Pro and IOPin help understand the complex interactions across different I/O layers from applications to the underlying parallel file systems, using two different approaches: static code instrumentation and runtime binary instrumentation. IOGenie helps users write data-intensive applications easily and effectively and enhances the quality of tool-generated code that exploits various optimizations on the underlying I/O software.
# Table of Contents

List of Tables ............................................................. vii
List of Figures ............................................................. viii
Acknowledgments ............................................................ x

Chapter 1. Introduction ................................................... 1

Chapter 2. IOPro: A Parallel I/O Profiling and Optimization Framework in HPC Systems ............................................. 7
  2.1 Introduction ......................................................... 8
  2.2 Background ........................................................ 10
    2.2.1 Challenges .................................................. 10
    2.2.2 Performance Metrics ....................................... 11
  2.3 High-level View of Instrumentation, Execution, and Visualization ........................................... 13
    2.3.1 Instrumentation Engine .................................... 13
    2.3.2 Execution Engine .......................................... 15
    2.3.3 Data Process Engine ...................................... 16
  2.4 Technical Details .................................................. 18
    2.4.1 Code Instrumentation ...................................... 18
    2.4.2 Configuration of Running Environments ............... 20
    2.4.3 Computation Methodology ................................ 21
    2.4.4 Query Model ............................................... 25
  2.5 Evaluation Results ................................................ 27
    2.5.1 FLASH I/O ................................................. 29
    2.5.2 S3D I/O .................................................... 37
    2.5.3 Case Study: Pinpointing I/O Interference in the Concurrent Execution of Multiple Applications ............ 44
  2.6 Conclusions ........................................................ 47

Chapter 3. IOPin: Runtime Profiling of Parallel I/O in HPC Systems ......................................................... 49
  3.1 Overview of Dynamic Instrumentation ............................ 50
  3.2 Backgrounds ....................................................... 52
    3.2.1 Overview of Dynamic instrumentation ............... 52
    3.2.2 Overview of Pin .......................................... 53
    3.2.3 Critical Path Detection Affecting Parallel I/O Performance 54
  3.3 Technical Details .................................................. 56
    3.3.1 Detailed Dynamic Instrumentation ....................... 56
  3.4 Evaluation .......................................................... 59
  3.5 Conclusions ........................................................ 62
# Chapter 4. IOGenie: Automatic Parallel I/O Code Generation and Optimization

## 4.1 Introduction

## 4.2 OVERVIEW

### 4.2.1 HPC Architecture and I/O Software Stack

### 4.2.2 Overview of IOGenie

## 4.3 OUR I/O MODEL

### 4.3.1 Data Set Definition Model

### 4.3.2 Data Access Model

## 4.4 FRAMEWORK

### 4.4.1 I/O Hints

### 4.4.2 I/O Options

### 4.4.3 Code Generation Engine

#### 4.4.3.1 Hint Parser

#### 4.4.3.2 Hint Analyzer

#### 4.4.3.3 I/O Optimizer

#### 4.4.3.4 Code Generator

## 4.5 EVALUATION

### 4.5.1 FLASH I/O

### 4.5.2 S3D I/O

### 4.5.3 Experimental Results

## 4.6 Conclusions

# Chapter 5. Related Work

## 5.0.1 Static/Dynamic Instrumentation

## 5.0.2 Tracing and debugging

## 5.0.3 Large-scale Distributed System Tracing

## 5.0.4 Code Generation

## 5.0.5 I/O Software Stack Optimizations

## 5.0.6 Caching and Prefetching

# Chapter 6. Conclusions and Future Work

# Bibliography
## List of Tables

2.1 Statistics that can be analyzed by IOPro. .................. 17  
2.2 Accepted query format. .................................. 22  
2.3 Overhead comparison. .................................. 28  
2.4 Baseline: S3D I/O detailed server I/O time and striped size of data. . . 46  
2.5 Running S3D I/O with the synthetic benchmark in interference. .... 46  
2.6 Running S3D I/O with the synthetic benchmark *without* interference. . 46
List of Figures

1.1 Parallel I/O software stack. ........................................... 2

2.1 Overview of IOPro. ..................................................... 13
2.2 The front-end (setup view) of the instrumented engine of IOPro. .... 14
2.3 The front-end (configuration view) of the execution engine. ........ 16
2.4 The front-end (query analyzer view) of data processing engine. ...... 17
2.5 Illustration showing how probes are inserted into the different layers of the I/O stack components by the instrumentation engine. .... 20
2.6 Computation of Latency and throughput. ............................. 21
2.7 Inclusive (dotted arrow) and exclusive (solid arrow) latency models. 24
2.8 Computation of inclusive latency. .................................... 25
2.9 Average execution time comparison. .................................. 28
2.10 Number of I/O calls issued to all servers from one process using the HDF5 interface. .................................................. 30
2.11 Inclusive latency values for the FLASH I/O benchmark. ............ 31
2.12 Total maximum and minimum latency from all processes. .......... 33
2.13 Maximum and minimum latency from the perspective of Process_16 for mpi_call_id ranging from 18 to 23. ............................... 35
2.14 Disk throughput for mpi_call_id 0 to 23 to write a checkpoint file from process 16 to 4 servers. ........................................ 36
2.15 Inclusive latency of the S3D I/O benchmark. .......................... 38
2.16 Inclusive latency from the perspective of Process_320 and Process_192. ................................................................. 39
2.17 Detailed latency from mpi_call_id 0 to 3 to create the first checkpoint file in Process_320. ............................................... 41
2.18 Disk throughput to server 0 from all 8 aggregator processes for mpi_call_id 0 to 3 to create the first checkpoint file. .................. 43
2.19 Comparisons of the execution time and the maximum I/O time in servers. 45

3.1 Overview of our dynamic instrumentation framework. ................ 51
3.2 The critical path affecting the application performance. .............. 55
3.3 Detailed illustration of how the trace information is passed. ........ 57
3.4 Comparison of S3D I/O execution time. ................................ 59
3.5 Comprehensive results drawn by IOPin. ................................ 60

4.1 Architecture of an HPC system. ...................................... 67
4.2 High-level view of IOGenie. .......................................... 68
4.3 Multidimensional arrays. ............................................. 69
4.4 Sample data access pattern. ......................................... 71
4.5 GUI view for giving the dataset hint. ................................ 75
4.6 GUI view for entering the data hint. .................................. 76
4.7 GUI view for specifying the I/O level. ................................ 77
4.8 Sample code generation by IOGenie, assuming an application written in C and PnetCDF as the high-level I/O library. 79
4.9 The block structure of FLASH mapped into memory and a checkpoint file. 82
4.10 S3D I/O data partitioning and mapping patterns. 85
4.11 Comparison of the average execution times of the hand-generated code versus the tool-generated code. 87
Acknowledgments

I would like to express my special appreciation and thanks to my advisor Professor Dr. Mahmut Kandemir, you have been a tremendous mentor for me. I would like to thank you for encouraging my research and for supporting me to grow as . Your advice on both research as well as on my career have been priceless. I would also like to thank my committee members, professor Dr. Mary Jane Irwin, professor Dr. Padma Raghavan, professor Dr. Dinghao Wu, and Dr Rajeev Thakur for serving as my committee members even at hardship. I also want to thank you for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions, thanks to you. I would especially like to thank my colleagues at Penn State University.

A special thanks to my family. Words cannot express how grateful I am to my mother, and father for all of the sacrifices that youve made on my behalf. Your prayer for me was what sustained me thus far. I would also like to thank all of my friends who supported me in praying, and incented me to strive towards my goal.
Chapter 1

Introduction

Users of HPC systems often find an interesting situation: it is not the CPU, memory, or network that restricts the performance of applications, but the storage systems. In fact, the prior research [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] shows that I/O behavior can be a dominant factor that determines the overall performance of many HPC applications from computational chemistry to biology to nuclear physics to financial services. Therefore, understanding the parallel I/O operations and the involved issues is a must to write efficient parallel I/O code to extract maximum performance from high-performance storage systems.

Unfortunately, understanding parallel I/O behavior and writing a efficient parallel I/O code is not trivial as it is a result of complex interactions between hardware and a number of software layers, collectively referred to as the I/O software stack, or simply I/O stack. Figure 1.1 illustrates a typical I/O stack used in many (if not most) HPC systems. Note that this figure is intended to present the software layers and a logical view; it is not meant to illustrate the physical connectivity and configuration of an I/O stack. At the lowest level is the storage hardware consisting of disks, SSDs, controllers, and interconnection network connecting multiple physical devices. At this level, data are accessed at the granularity of “blocks” across multiple physical devices such as in a RAID array. Above the storage hardware are the parallel file systems, such as Lustre [14],
GPFS [15], PanFS [16], and PVFS [17]. The roles of the parallel file system are to manage the data on the storage hardware, present the data as a directory hierarchy, and coordinate accesses to files and directories in a consistent fashion. The MPI-IO library [18], part of MPI-2 [19], sits, as a middleware, on top of the parallel file systems. It provides a standard I/O interface and a suite of optimizations including data caching and process coordination [1, 2, 3, 4, 5, 6].

While the MPI-IO interface is effective and advantageous because of its performance and portability, it does not support structured data abstraction for scientific applications. To provide that, high-level I/O libraries (e.g., Parallel netCDF [20] and HDF5 [21]) are added on top of MPI-IO. These high-level libraries allow application programmers to better describe how their applications access shared storage resources. Furthermore, they provide data abstractions that match the way the scientific applications view data. As shown in Figure 1.1, a parallel I/O application may directly call the...
MPI-IO library or a POSIX I/O function to access the disk-resident data sets. Alternatively, large-scale, data-intensive applications may exercise several layers of the I/O stack. Since the interactions among these layers are complex and unpredictable, understanding and characterizing those interactions must precede writing a parallel I/O code that efficiently utilizes these I/O software layers. For example, a high-level HDF5 code can be fragmented into different smaller calls, over the I/O stack, and each of these calls can experience a different latency. Similarly, two independent I/O calls can conflict in accessing the same storage device/component.

One approach to understanding I/O behavior is to let application programmers or scientists instrument I/O software stack manually. Unfortunately, this approach is extremely difficult and error-prone. In fact, instrumenting even a single I/O call may necessitate modifications to numerous files from the application to multiple I/O software layers below. Since many parallel scientific applications today are expected to run on large-scale systems with hundreds of thousands of processes in order to achieve better resolution, even collecting and analyzing trace information from them is laborious and burdensome.

Even if users or scientists understand numerous APIs and their complex interactions in the I/O stack, it will be still challenging for them to write a parallel I/O code since there exist various optimization opportunities across I/O libraries and parallel file systems, which helps improve I/O performance. In particular, it is not unusual for an average (non-I/O expert) user to make mistakes in orchestrating I/O accesses in the code. Further, such a user may not be able to take full advantage of I/O optimization
opportunities within and across different layers of the I/O stack. Writing correct and optimized I/O code is a serious challenge even for experienced programmers.

Motivated by these observations, we propose a parallel I/O profiling and visualization framework for high-performance storage systems, called IOPro and a runtime I/O profiling tool, called IOPin. We also propose an GUI-based parallel I/O code generation and optimization framework for HPC applications, called IOGenie. Instead of manually instrumenting source code of applications and other components of the I/O stack, IOPro takes as input the description of the target I/O stack and the application program, and automatically generates the instrumented I/O stack to trace the specified I/O operations, and compiles and builds it. Next, it runs the application with detailed configuration information for I/O servers (PVFS2 in our case) and an MPI process manager, mpiexec. It then collects and analyzes the trace log data and presents detailed statistics based on user-specified metrics of interest.

Unlike IOPro, IOPin provides dynamic binary instrumentation for the I/O stack. To implement our current prototype of IOPin, we leverage a lightweight binary instrumentation using Pin [22]. That is, our tool performs the instrumentation in the binary code of the MPI library and the underlying parallel file system, PVFS, at runtime. Therefore, in this scheme, our tool provides a language-independent instrumentation capability, targeting scientific applications written in C/C++ or Fortran. Furthermore, our tool requires neither source code modification nor recompilation of the applications and parallel I/O stack components.

Lastly, motivated by the observation indicating that writing correct and efficient parallel I/O code is very challenging, IOGenie automatically generates parallel I/O code
for any I/O stack, based on the user input parameters—high-level I/O hints and annotations. Specifically, it takes the I/O hints, analyzes the given hints, and formulates the optimized code at each I/O point that users specify in the application.

A unique aspect of our implementation of IOPro is that it provides an integrated profiling and analysis environment (IPAE) for the entire parallel I/O software stack. It can work with different I/O stacks and user-provided probe code. For instance, with the user-specified probes, it can trace parallel I/O in Blue Gene/P systems that deploy the I/O Forwarding Scalability Layer (IOFSL) [23]. It can also provide a reconfigurable setup for the I/O stack. Our proposed infrastructures, IOPro and IOPin, also provide a hierarchical view for parallel I/O. In our implementation, every MPI I/O call has a unique identification number in the MPI-IO layer and is passed to the underlying file system with trace information. This mechanism helps associate the MPI I/O call from the application with its subcalls in the file system layer systematically. In addition, our framework visualizes detailed I/O performance metrics for each I/O call, including latency, throughput, estimated energy consumption, and the number of I/O calls issued to and from servers and clients. Using our IOGenie, users can easily write a parallel I/O code in the data-intensive applications and also enhance the quality of the generated code that exploits various optimizations on the underlying I/O software.

We believe that our three infrastructures are a powerful and useful tool for scientists and application programmers as well as performance engineers. For the scientists and application programmers who do not have an in-depth knowledge of underlying complexities of emerging HPC systems, IOPro and/or IOPin can provide detailed I/O statistics that helps them understand the characteristics of I/O from the perspective of
the applications. Also, our automatic I/O code generator is expected to significantly improve programmers’ productivity. For the performance engineers, it enables customized instrumentation for more detailed performance measurements. Therefore, IOPro and IOPin can enable insights into the complex I/O interactions of scientific applications and provide an adaptive I/O strategy and IOGenie can improve programmers’ productivity and the code quality of scientific applications.

The rest of this dissertation is organized as follows. Chapter 2 explains our proposed parallel I/O profiling and visualization framework for high-performance storage system, IOPro. Chapter 3 elaborates on the runtime profiling approach for parallel I/O leveraged by Pin, and Chapter 4 discusses our automatic parallel I/O code generation and optimization tool for HPC applications. Related work is discussed in Chapter 5, followed by our concluding remarks and a brief discussion of the future work in Section 6.
Chapter 2

IOPro: A Parallel I/O Profiling and Optimization Framework in HPC Systems

Efficient execution of large-scale scientific applications requires high-performance computing systems designed to meet the I/O requirements. To achieve high-performance, such data-intensive parallel applications use a multi-layer layer I/O software stack, which consists of high-level I/O libraries such as PnetCDF and HDF5, the MPI library, and parallel file systems. To design efficient parallel scientific applications, understanding the complicated flow of I/O operations and the involved interactions among the libraries is quintessential. Such comprehension helps identify I/O bottlenecks and thus exploits the potential performance in different layers of the storage hierarchy. To trace the execution of I/O operations and to understand the complex interactions in the I/O stack, we have designed and implemented a GUI-based integrated profiling and analysis environment, IOPro. IOPro automatically generates an instrumented I/O stack, runs applications on it, and visualizes detailed statistics in terms of the user-specified metrics of interest. We present results from two real applications and demonstrate how our tool is used in practice. By generating on the end-to-end trace of the whole I/O stack and pinpointing I/O interference, IOPro aids in understanding I/O behavior and proving I/O solutions to improve the performance.
2.1 Introduction

Emerging data-intensive applications make significant demands on storage system performance. Although some of the most important issues exist in parallel I/O systems, including parallel I/O system components and architecture, parallel access patterns, and consistency semantics, I/O is a dominant factor that determines the overall performance of many HPC applications. Therefore, understanding the parallel I/O operations and the involved issues is critical to meet the requirements for a particular HPC system and/or decide I/O solutions to accommodate expected workloads.

Unfortunately, understanding parallel I/O behavior is not trivial as it is a result of complex interactions between hardware and a number of software layers, collectively referred to as the I/O software stack, as shown in Figure 1.1. Since the interactions in the I/O software stack are complex and unpredictable, understanding and characterizing those interactions must precede performance tuning and optimization for the HPC applications.

One approach to understanding I/O behavior is to let application programmers or scientists instrument the I/O software stack manually. Unfortunately, this approach is extremely difficult and error-prone. In fact, instrumenting even a single I/O call may necessitate modifications to numerous files from the application to multiple I/O software layers below. Worse, a high-level I/O call from the application program can be fragmented into multiple calls (subcalls) in the MPI library, which is severely challenging.
Since many parallel scientific applications today are expected to run on large-scale systems with hundreds of thousands of processes in order to achieve better resolution, even collecting and analyzing trace information from them is laborious and burdensome.

Motivated by these observations, we have developed a performance analysis and visualization framework for parallel I/O, called IOPro. Instead of manually instrumenting source code of applications and other components of the I/O stack, IOPro takes as input the description of the target I/O stack and the application program, automatically generates the instrumented I/O stack to trace the specified I/O operations, and compiles and builds it. Next, it runs the application with detailed configuration information for I/O servers (PVFS2 in our case) and an MPI process manager, mpiexec. Then, it collects and analyzes the trace log data and presents detailed statistics based on user-specified metrics of interest.

A unique aspect of our implementation is that it provides an integrated profiling and analysis environment (IPAE) for the entire parallel I/O software stack. It can work with different I/O stacks and user-provided probe code. For instance, with the user-specified probes, it can trace parallel I/O in Blue Gene/P systems that deploy the I/O Forwarding Scalability Layer (IOFSL) [23]. Also, it can provide a reconfigurable setup for the I/O stack. Last but not least, it provides a hierarchical view for parallel I/O. In our implementation, every MPI I/O call has a unique identification number in the MPI-IO layer and is passed to the underlying file system with trace information. This mechanism helps associate the MPI I/O call from the application with its subcalls in the file system layer systematically. In addition, our framework visualizes detailed I/O
performance metrics for each I/O call, including latency, throughput, estimated energy consumption, and the number of I/O calls issued to and from servers and clients.

We believe that IOPro is a powerful and useful tool for scientists and application programmers as well as performance engineers. For the scientists and application programmers who do not have an in-depth knowledge of underlying complexities of emerging HPC systems, it can provide detailed I/O statistics that helps them understand the characteristics of I/O from the perspective of the applications. By using the performance measurements of the underlying I/O stack, more optimized code can be implemented. For the performance engineers, it enables customized instrumentation for more detailed performance measurements. Therefore, IOPro can enable insights into the complex I/O interactions of scientific applications and provide an adaptive I/O strategy.

2.2 Background

In this section, we discuss the challenges in characterizing the I/O performance of modern HPC systems. We also explain the importance of the collected performance metrics and their usage to improve the I/O performance.

2.2.1 Challenges

Modern HPC systems comprise multiple entities such as high-level I/O libraries (e.g., PnetCDF and HDF5), the MPI library as a middleware, and POSIX on top of the underlying parallel file systems. When a scientific application runs on large-scale systems with hundreds of thousands of processes, its operation is often complex and difficult to understand. Frequently, application I/O calls can be optimized in the middle I/O layer to
achieve better performance. Also, the high-level I/O calls from applications can break
down into multiple calls in the MPI library, which make it extremely challenging to
understand and reason about.

Most of the previous research in this area focuses on presenting performance
metrics for the given applications. However, these statistics only reflect quantitative
information at each layer of the I/O stack rather than a deep understanding of the I/O
interaction and association from the application through the multiple libraries to the
underlying parallel file system. Instead, our scheme provides a qualitative approach
to associate high-level I/O from the application with the operations in the underlying
parallel file system by *automatically* injecting the probe code, and visualizes the user-
provided metrics of interest for better understanding. As a result, it helps scientists and
system engineers profile and improve the performance of applications running on deep
storage hierarchies. We want to emphasize that, while in principle a knowledgeable user
can manually instrument an I/O stack, in practice this is very difficult due to the complex
interactions between different layers, which makes it very challenging to pass/propagate
values/metrics of interest across the layers and accumulate results.

2.2.2 Performance Metrics

Depending on I/O demands and data access patterns, a given parallel applica-
tion may require bounded execution time, relatively low throughput, or both. In many
parallel applications, the requests from different processes are frequently interleaved and
merged into contiguous portions of the file to reduce the high I/O latency. When such
an optimization, broadly referred to as *collective I/O*, is used, all the joined processes
broadcast and exchange the information related to the I/O request. If the I/O access patterns of all processes are contiguous and can benefit from collective I/O, an aggregator process can access disk-resident data by two-phase I/O: (1) redistribution of data to the processes (communication phase) and (2) a single, large, contiguous access to data (I/O phase) in case of write operation. This method has the additional cost of interprocess communication, but it can significantly reduce the I/O time. Although collective I/O is performed to improve I/O latency, the performance of collective I/O can be significantly affected by the critical path from the process to the server. For example, if the process on the critical path has a small size of temporary buffer needed for two-phase I/O, frequently copies the data into the buffer, and communicates other processes for redistribution, it can degrade the performance. In this case, the critical path from the aggregator process to the server dominates the overall application performance. Also, the I/O server on the critical path can be a major bottleneck in certain situations such as explosion of I/O requests to the server, network hardware failure, or faulty I/O server. Since the I/O operations interfere with each other during the execution of multiple applications, it is also important to figure out how many I/O operations are issued and which server(s) the I/O requests from the applications target. In case of burst I/O to the server, by setting MPI hints the application can perform I/O operations without striping data to the bottleneck I/O server. Using our framework, therefore, users can easily/automatically generate the instrumented I/O stack to capture latency, throughput, and I/O call access information that affect the performance, and analyze those metrics by visualization.
Fig. 2.1. Overview of IOPro. It takes as input an application program and I/O stack information and builds an instrumented I/O stack to profile I/O operations, separately from the original I/O stack. After configuring the PFS server (PVFS in our case) and MPI program launcher, *mpiexec*, it runs the application program. The query analyzer then collects trace log files and returns the statistics based on the metrics of interest.

2.3 High-level View of Instrumentation, Execution, and Visualization

In this section, we first give a high-level view of IOPro. As shown in Figure 2.1, IOPro consists of three main components: *instrumentation engine*, *execution engine*, and *data processing engine*. Each of these components works with its corresponding front-end (i.e., setup view, configuration view, and query analyzer, respectively), as will be explained in the following subsections.

2.3.1 Instrumentation Engine

To provide an automated I/O tracing functionality for parallel applications, IOPro accepts the necessary information from the setup view (Figure 2.2). This information includes the directory locations of an application and the I/O software stack such as the parallel file system (e.g., PVFS), the MPI library (e.g., MPI-IO), and the high-level I/O library (e.g., HDF5). It also takes the location of trace log files generated by each
layer of the I/O stack. As shown in Figure 2.2, an instrumented file for a high-level I/O library is automatically chosen, depending on the selected high-level I/O library. In the example, H5FDmpio.c would be instrumented when targeting HDF5. In addition, the `make` option “

```bash
make -f Makefile.bb flash_benchmark_io
```

is given to compile the FLASH I/O benchmark [24]. Further, if desired, a trace option can be chosen here to track a specific operation and code range (i.e., write, read, or both) and application source code lines (1-5000 in this case). Note that the current implementation targets a user-level parallel file system. Unlike other system-level parallel file systems such as Lustre, GPFS, and PanFS, PVFS2 clients and servers can run at user-level\(^1\). Therefore, we can easily implement the functionality to trace and profile I/O operations in a hierarchical

\(^1\)PVFS2 also supports an optional kernel module that allows a file system to be mounted as in other file systems
fashion, without kernel modifications that are normally not allowed in system-level file systems.

As a backend of setup view, an instrumentation engine consists of a probe selector and a probe inserter. In this context, a probe is a piece of code being inserted into the application code and I/O software stack (e.g., in the source code of the high-level I/O library, the MPI library, and PVFS2), which helps us collect the requested statistics. Using the user-provided information from the setup, the instrumentation engine inserts the probe into the appropriate locations in the I/O stack automatically, and generates an instrumented version of PVFS2, the MPI-IO library as well as the high-level I/O library. More details are provided in Section 2.4.

2.3.2 Execution Engine

After creating the instrumented I/O stack and the application program successfully in the previous stage, the execution engine builds and compiles them. Also, as in Figure 2.3, the front-end (configuration view) in the execution engine takes information about the file systems, storage locations, endpoints that each server manages, metadata servers, and I/O servers. Using the user-provided information, it creates a global PVFS2 server configuration file (e.g., fs.conf). In general, PVFS2 servers are deployed using this global configuration file shared by all PVFS2 servers. Figure 2.3 shows an example of the front-end of the execution engine where the user-provided information is taken to run the application. In this example, bb18 is configured as a metadata server and bb05, bb06, bb07, and bb08 as I/O servers. The 512 MPI processes specified in the mpd.hosts
file that has node information would be launched by mpiexec to run the executable flash_benchmark_io.

We want to emphasize that the instrumented I/O stack is separately built from the non-instrumented one. Therefore, the application can run either on the instrumented I/O stack or on the non-instrumented (original) I/O stack by setting `LD_LIBRARY_PATH`.

### 2.3.3 Data Process Engine

After running the application with the user-provided information in the execution engine, the data process engine collects all trace log files from each layer of the target I/O stack. Table 2.1 lists a representative set of high-level metrics that can be profiled and visualized by our prototype of IOPro. Based on the user’s query taken in the front-end of the data process engine (Figure 2.4), the data process engine calculates the statistics.
Table 2.1. Statistics that can be analyzed by IOPro.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O latency experienced by each I/O call in each layer (MPI library, client, server, or disk) in I/O stack</td>
<td></td>
</tr>
<tr>
<td>Average I/O access latency in a given segment of the program</td>
<td></td>
</tr>
<tr>
<td>Throughput achieved by a given I/O read and write call</td>
<td></td>
</tr>
<tr>
<td>Disk power consumption incurred by each I/O call</td>
<td></td>
</tr>
<tr>
<td>Number of disk accesses made by each I/O call</td>
<td></td>
</tr>
<tr>
<td>Amount of time spent during inter-processor communication in executing a collective I/O call</td>
<td></td>
</tr>
<tr>
<td>Number of I/O nodes participating in each collective I/O</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2.4. The front-end (query analyzer view) of data processing engine.
using the collected trace log files, returns the performance metrics, and visualizes it for further investigation. The detailed query specification is discussed later in Section 2.4.4.

2.4 Technical Details

In this section, we go over the code instrumentation component of IOPro and the use of probes, the configuration of the servers, the role of the query analyzer, and various sample queries.

2.4.1 Code Instrumentation

Using the information provided in the setup view (Figure 2.2), IOPro automatically patches PVFS, the MPI library, and the high-level I/O libraries, such as PnetCDF and HDF5, as a preparation for code instrumentation. Using the probe library that maintains probe template codes for PVFS, MPI, PnetCDF, and HDF5, the instrumentation engine generates actual probes that contain the trace log file location. In this context, a *probe* is a piece of code inserted into the I/O software stack software to help collect the required statistics. IOPro then creates a probe location file from a provided template file (as in Listing 1) that specifies the appropriate location in the MPI library and PVFS where the probes should be inserted. The syntax given in Listing 1 is for illustrative purposes and is based on an initial prototype of IOPro that is currently under development. Probe selector, a sub-component of the instrumentation engine, parses the probe location file and extracts the location information for the probe code to be inserted. Using the extracted probe location information, the probe inserter automatically inserts the appropriate probes into the proper locations in the I/O stack.
Figure 2.5 illustrates how the instrumentation engine works. In this figure, IO-
CallIDs is a small array that contains information of each layer such as the MPI I/O
call ID, PVFS client ID, PVFS server ID, disk operation ID, I/O type, and the start
timestamp and the end timestamp of each layer. When IOCallIDs are passed from the
upper layer to the layers below, the inserted probes extract the information from them
and generate the trace log files with latency statistics at the boundary of each layer.

Note that a high-level MPI I/O call can be fragmented into multiple small subcalls.
For example, in two-phase I/O [6], which consists of an I/O phase and a communication
phase, tracing an I/O call across the boundaries of the layers in the I/O stack is not
trivial. In our implementation, each call has a unique identification number in the current
layer and passes it to the layers below. This helps us associate the high-level call with
its subcalls in a hierarchical fashion. It also helps analyze trace log data by combining
the statistics that come from different layers in a systematic way (for example, all the
variables that hold latency information at different layers are associated with one another
using these IDs).

In the PVFS server, a unique data structure, called flow_descriptor, maintains all
the information to perform requested I/O operations from the PVFS clients. This struc-
ture is used by our tool. In Figure 2.5, for example, the Server-start-probe inserted into
the PVFS server layer extracts the necessary information passed from the PVFS client
and packs it into the flow_descriptor. Since the flow_descriptor is passed to the entire
PVFS server, the probes inserted in the server can extract the necessary information
from it and manipulate the statistics to trace I/O calls without much difficulty.
Fig. 2.5. Illustration showing how probes are inserted into the different layers of the I/O stack components by the instrumentation engine. Left: I/O call flows when `MPI_File_write_all()` is issued. Right: the instrumented I/O stack.

### 2.4.2 Configuration of Running Environments

After PVFS is installed, the user specifies which nodes in the cluster will serve as metadata servers and I/O nodes. The user also determines how many MPI processes will be used to run the application. Unfortunately, manually configuring the PVFS servers and running the parallel application on them can be very tedious and challenging. Instead of manual configuration, our tool provides a simple mechanism to specify configuration of running environments (see Figure 2.3). It takes the configuration metrics for the servers such as metadata server(s) and I/O server(s) as well as a filename storing this configuration, protocol, port number, storage location, and a log filename for each server.
Fig. 2.6. Computation of latency and throughput. I/O latency computed at each layer is equal to the maximum value of the I/O latencies obtained from the layers below it. In contrast, I/O throughput is the sum of I/O throughput coming from the layers below.

It also takes a filename that specifies the host machine(s) from which the MPI job launcher, `mpiexec`, is launched and the number of processes (or clients) to be used for running the given application.

This simple configuration method also provides us with the flexibility of running a single application and/or multiple applications using different configuration options without rec Compilation of the instrumented I/O stack. For example, we can easily run the application program(s) on the instrumented I/O stack with different combinations of configurations such as (1) running the same application but varying the number of metadata server(s), I/O server(s), or PVFS clients; (2) running different applications on the same configuration; (3) different mixes of the previous two; and (4) running multiple applications on the same configuration and/or a different one.

### 2.4.3 Computation Methodology

After running the application program, the data process engine collects all trace log files from each layer of the I/O stack. Based on the user’s queries, it then processes
the trace log and returns the corresponding statistics. As shown in Figure 2.4, our current implementation provides functionalities to analyze latency, throughput, estimated energy consumption, and the number of calls issued from clients to servers. We want to emphasize however that, if desired, IOPro can be easily extended to accommodate additional/new statistics. Table 2.2 shows the input query formats accepted by the current implementation of our tool. The detailed description of our query will be given in Section 2.4.4.

<table>
<thead>
<tr>
<th>Latency Breakdown</th>
<th>process_id</th>
<th>mpi_call_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency Breakdown</td>
<td>process_id</td>
<td>mpi_call_id</td>
</tr>
<tr>
<td>Latency Operation List</td>
<td>process_id</td>
<td>mpi_call_id</td>
</tr>
<tr>
<td>Latency Operation Min</td>
<td>process_id</td>
<td>mpi_call_id</td>
</tr>
<tr>
<td>Latency Operation Max</td>
<td>process_id</td>
<td>mpi_call_id</td>
</tr>
<tr>
<td>Latency Operation Avg</td>
<td>process_id</td>
<td>mpi_call_id</td>
</tr>
<tr>
<td>Thru.</td>
<td>process_id</td>
<td>mpi_call_id</td>
</tr>
<tr>
<td>Energy</td>
<td>active_p</td>
<td>process_id</td>
</tr>
<tr>
<td>Call</td>
<td>process_id</td>
<td>mpi_call_id</td>
</tr>
</tbody>
</table>

Figure 2.6 illustrates the computation of latency and throughput. For each I/O call, the I/O latency value computed at each layer is the maximum value of the I/O latencies from the layers below it.

\[
\text{Latency}_i = \text{Max}(\text{Latency}_{i-1}A, \text{Latency}_{i-1}B, \text{Latency}_{i-1}C)
\] (2.1)

However, the computation of I/O throughput in Figure 2.6(b) is additive; in other words, I/O throughput at any layer is computed by summing the sizes of data coming from the
layers below below it.

\[
Throughput_i = \sum (Thpt_{i-1}A, Thpt_{i-1}B, Thpt_{i-1}C)
\]  

(2.2)

To compute (estimate) the energy consumption for I/O call, we employ the power model described in [25].

In our work, inclusive latency means the time spent in the current layer, which includes the latency in the layers below. Exclusive latency is the time spent in the current layer and excludes the sublayers. That is, it can be calculated from inclusive latency by subtracting the sublayer latency from the current layer. Figure 2.7 demonstrates how the inclusive and exclusive latencies are computed (the dotted arrows denote inclusive latency, and the solid arrows indicate exclusive latency). The figure also shows the employed tracing mechanism, which identifies and distinguishes I/O calls at each layer. Each layer generates a unique ID such as process_id, mpi_call_id, pvfs_call_id, and server_id when an I/O call is passed. This unique number is cumulatively carried down to the sublayers. All information for the I/O calls passed through the entire I/O stack is stored in the last layer. By matching and identifying these IDs, one can easily relate the high-level MPI I/O call to the subcalls.

Figure 2.8 shows the computation of inclusive latency in more detail. When, for instance, a collective I/O call is issued, this I/O call can be fragmented into multiple I/O calls in the MPI library if the size of requested I/O is larger than that of the buffer in the MPI library. For example, in the figure, mpi_call_id 0 is fragmented into two pvfs_call_id’s 0 and 1. In the PVFS client layer, each split I/O call has its own ID,
00 and 01 for the mpi_call_id 0, respectively. When these calls reach servers 0 and 1, the cumulative trace information is 000 and 001 for cumulative ID 00 (blue line), and 010 and 011 for ID 01 (red one). This relationship is maintained until the end of the I/O stack is reached. Therefore, for mpi_call_id 0, the inclusive latency computed at the PVFS client layer is

$$Latency_{client} = \sum (L_{00}, L_{01}),$$

and the inclusive latency at the PVFS server layer is

$$Latency_{server} = \sum (Max(L_{000}, L_{001}), Max(L_{010}, L_{011})), $$

where $L$ denotes latency. Exclusive latency, on the other hands, can be calculated as shown in Figure 2.7.
2.4.4 Query Model

As listed in Table 2.2, the current implementation of our tool provides four user metrics to be analyzed: latency, throughput, energy, and call information. Below, we discuss the details of our queries (metrics in square brackets are the user-provided input).

- **Latency Breakdown Inclusive** \([\text{process\_id}][\text{mpi\_call\_id}]\)

  This query returns the inclusive latency information given process\_id and mpi\_call\_id. For example, the query ‘Latency Breakdown Inclusive [0-1] [1-10]’ returns the inclusive latency for the mpi\_call\_id 1 to 10 issued from the process 0 and 1 to all servers in breakdown-fashion, as described in Section 2.4.3. This is also applied to compute exclusive latency.

- **Latency Operation List** \([\text{process\_id}][\text{mpi\_call\_id}][\text{pvfs\_call\_id}][\text{server\_id}]\)

  This returns all latency information, listing detailed latency statistics for all matching process-server combinations. For example, a query such as Latency Operation
List [0-4] [1-10] [-] [1-3] returns all combinations for mpi_call_id 1 to 10 issued from the process 0 to 4 to the server 1 to 3. In this case, all possible combinations are 15. In the parameter pvfs_call_id, “-” means all. By default, pvfs_call_id is set to “-” for simplicity since it is implicitly fragmented depending on the size of the I/O request.

- Latency Operation Max/Min [process_id] [mpi_call_id] [pvfs_call_id] [server_id]

This is similar to the list latency format except that it returns the maximum/minimum latency. For example, Latency Operation Max [0-4] [1-10] [-] [1-3] returns the maximum latency for mpi_call_id 1 to 10 issued from processes 0 to 4 to servers 1 to 3. Unlike list latency, this shows only the maximum latency among the given servers and the corresponding server number. Note that this query provides the latency statistics from the process’s and server’s points of view. More specifically, from the process’s point of view, we can easily identify in which server a given mpi_call_id experiences the maximum latency. From the server’s point of view, we can identify the process that has the most latency in that server. Also, unlike inclusive/exclusive latency, it presents detailed latency, not in breakdown fashion. For example, if an mpi_call_id 0 is split into ten subcalls, it returns the maximum latency among all ten individual subcalls.

- Latency Operation Avg [process_id] [mpi_call_id] [pvfs_call_id] [server_id]

This returns the average latency given the ranges of processes, MPI I/O calls, and servers for each MPI I/O call.
• Throughput [process_id] [mpi_call_id] [pvfs_call_id] [server_id]

This calculates disk throughput in each PVFS server for each mpi_call_id from the process's and server's points of view.

• Energy [active_power] [inactive_power] [process_id] [mpi_call_id] [pvfs_call_id] [server_id]

This calculates the estimated energy consumption in server class disks [25]. It also plots both the process's and the server's views. Here, active_power is the amount of power consumption (in watts) when the status of disk is active; inactive_power is the power consumption when the disk is not active.

• Call [process_id] [mpi_call_id] [pvfs_call_id] [server_id]

This returns statistics about the number of issued I/O calls from processes to servers. Using this query, one can detect which I/O server is suffering the most from the I/O requests.

2.5 Evaluation Results

Most of the GUI components of IOPro have been implemented in Java and the JFreeChart library [26]. Our implementation was evaluated on the Breadboard cluster [27] at Argonne National Laboratory. In our experiments, we built an I/O stack with pvfs-2.8.2, mpich2-1.2.1p1, pnetcdf-1.2.0, and hdf-1.8.5. Then, IOPro automatically generated an instrumented version of this I/O stack using IOPro. Note that we opted to use PVFS, the user-level parallel file system, so that we could easily implement a tracing and profiling mechanism without kernel modifications.
Fig. 2.9. Average execution time comparison. Two benchmarks run on 512 MPI processes with 1 metadata server and various number of I/O servers. It can be seen that, in both benchmarks, the overhead caused by our implementation is approximately 8.5%.

To evaluate the overhead caused by our implementation, we measured the average execution time after 20 iterations, running two I/O intensive benchmarks, S3D I/O and FLASH I/O. In each run, we dropped cache both in the servers and in compute nodes to minimize the effect on cache. Figure 2.9 compares the average execution time of two benchmarks running with 512 MPI processes and various number of I/O servers on a non-instrumented I/O stack and an instrumented one. The result shows that the overhead from all combinations is approximately 8.5%, on average. Table 2.3 presents the detailed statistics.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>I/O Server</th>
<th>Non-instrumented</th>
<th>Instrumented</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLASH I/O</td>
<td>4</td>
<td>49.66 sec</td>
<td>52.85 sec</td>
<td>6.4 %</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>31.32 sec</td>
<td>34.96 sec</td>
<td>11 %</td>
</tr>
<tr>
<td>S3D I/O</td>
<td>4</td>
<td>39.86 sec</td>
<td>43.16 sec</td>
<td>8.3 %</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>36.39 sec</td>
<td>39.40 sec</td>
<td>8.3 %</td>
</tr>
</tbody>
</table>
To demonstrate the capabilities of IOPro, below we present detailed results with two benchmarks.

2.5.1 FLASH I/O

FLASH I/O benchmark [28] is the I/O kernel of the FLASH application [24], a block-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed for studying nuclear flashes on neutron stars and white dwarfs. The computational domain is divided into small blocks that are distributed across different MPI processes. The FLASH block is a three-dimensional array and there are 80 blocks on each MPI process. Each block contains inner blocks with additional 4 element of guard cells to hold the state variables of the neighboring blocks. The inner block surrounded by guard cells has 24 data array variables, e.g., density, velocities, energy, and pressure. Every process writes these blocks into a checkpoint file using 24 collective I/Os, in a manner that the checkpoint file appears as the data for variable 0 up to variable 23. FLASH I/O generates one checkpoint file and two visualization files that contain centered and corner data. FLASH I/O works with both the PnetCDF and HDF5 interfaces to save data and metadata in each high-level I/O format.

In our evaluation, we ran the FLASH I/O benchmark on 1 metadata server, 4 I/O servers, and 512 MPI processes using the HDF5 interface. We configured a $8 \times 8 \times 8$ block size in X-Y-Z dimensions. In this experiment, FLASH I/O produces a 3.8 GB checkpoint file and two visualization files (329 MB and 466.8 MB, respectively). From Figure 2.10, we see that 32 collective I/O calls without fragmentation are evenly issued
Fig. 2.10. Number of I/O calls issued to all servers from one process using the HDF5 interface. We see that each server receives the same number of calls from that process to all servers from one process. The checkpoint file is generated by the first 24 I/O calls and two visualization files are created by the following 4 I/O calls, respectively.

Figure 2.11 illustrates the inclusive latency of the FLASH I/O benchmark from Process 0 to process 30 among all 512 processes. The total time in the figure presents the global time spent in each layer to run the application program, using different color legends. The context information gives the detailed latency statistics in the MPI I/O library, PVFS client, PVFS server, and server disk layers for each process. We observe that the latency in the MPI I/O library and the PVFS layer is unevenly distributed among the processes. For example, the most time spent in the MPI library for the I/O requests is approximately 11.51 seconds, in process 16, and the least is about 8.97 seconds in process 1. We observe that the overhead in the MPI library is relatively small. In MPI-IO, the default collective buffering scheme is set to automatic, that is, MPI-IO uses heuristics to determine whether it enables the optimization. Since FLASH I/O accesses noncontiguous data, rarely exploiting data exchanges and optimization for collective I/O,
MPI-IO disables collective buffering and automatically converts collective I/O requests to independent I/O requests. We find that the latency in the MPI I/O library increases about 14% because of the communication overhead when forcefully enabling collective I/O. Since all the joined processes exploit independent I/O for running the application, the completion time for each substantially differs.
Listing 1. A sample template file specifying probe locations. In this template file, five different probes are specified for the application, MPI I/O library, PVFS client, PVFS server, and disk layers, including probe names and location information to be inserted as well as file names to be instrumented in the I/O stack.
Fig. 2.12. Total maximum and minimum latency from all processes. In both figures, the time spent in the PVFS client layer is the same, but the time spent in the PVFS server and disk is different.
Figure 2.12 compares the maximum and the minimum latency for all I/O requests issued to all servers from Process 0 to Process 30. Unlike in Figure 2.12(a), we observe a bigger latency gap between the client layer and the server (the green portion) in Figure 2.12(b). We also notice the latency difference spent in the server and the disk between Figure 2.12(a) and Figure 2.12(b). If the data lies on the data block, but is smaller and not fit into it, it takes less time to write the smaller portion of the data.

Figure 2.13 plots more detailed latency statistics for mpi_call_id 18 through 23. The difference between the maximum and minimum latency of process 16 (in Figure 2.12) is caused by the latency in the server from those I/O calls, as shown in Figure 2.13. Note that the number on the server legend (the green bar) in Figure 2.13 is the server ID. The I/O request (18-0) from the process, for example, spends the maximum amount of time in server 0 and has minimum latency in server 3 even if it stripes 64 KB data over all servers.
Fig. 2.13. Maximum and minimum latency from the perspective of Process_16 for mpi_call_id ranging from 18 to 23.
Disk throughput from mpi_call_id 0 through 23 from process 16 is plotted in Figure 2.14.

![Figure 2.14.](image)

Here, we observe the I/O characteristics of FLASH I/O. Although FLASH I/O issues collective I/O requests to write checkpoint files, MPI-IO disables them and automatically converts them to independent I/O requests because the data is noncontiguous. We also notice that collective buffering rather degrades the performance of the FLASH I/O. Based on this observation, the optimized code can be implemented in a way to exploit the potential benefits of collective I/O. As seen in Figure 2.13, the latencies of the I/O calls in the specific servers are higher than the others. Therefore, the application and I/O stack can be tuned to reduce those variances.
2.5.2 S3D I/O

The S3D I/O benchmark is a parallel turbulent combustion application, named S3D [29], developed at Sandia National Laboratories. Using a direct numerical simulation, S3D solves the fully compressible Navier-Stoke, total energy, species, and mass continuity equations coupled with detailed chemistry. A checkpoint is performed at regular intervals; its data consists primarily of the solved variables in 8-byte, three-dimensional arrays. This checkpoint data can be used to obtain several physical quantities of interest. Therefore, most of the checkpoint data is maintained for later use. At each checkpoint, four global arrays—representing the variables of mass, velocity, pressure, and temperature—are written to files.

Among those four arrays, pressure and temperature are three-dimensional arrays while mass and velocity are four-dimensional. All four arrays share the same size for the lowest three spatial dimensions X, Y, and Z and are partitioned among the MPI processes along with X-Y-Z dimensions. For the three-dimensional arrays, the subarray of each process is mapped to the global array in block partitioning of X-Y-Z dimensions. For the four-dimensional arrays, the lowest X-Y-Z dimensions are partitioned as same as the three-dimensional arrays, but the fourth dimensions is not partitioned. For the arrays of mass and velocity, the length of fourth dimensions is 11 and 3, respectively.

S3D I/O supports MPI-IO, PnetCDF, and HDF5 interfaces. In our evaluation, we configured 1 metadata server and 8 I/O servers and ran S3D I/O on 512 MPI processes with the PnetCDF interface. We maintain the block size of the partitioned X-Y-Z dimensions as \(400 \times 200 \times 200\) in each process. With this configuration, S3D I/O produces
Fig. 2.15. Inclusive latency of the S3D I/O benchmark.

three checkpoint files, 1.9 GB each. The average execution time on the instrument I/O stack with 8 I/O servers is presented in Figure 2.9(b).

Figure 2.15 shows the inclusive latency generated by the query analyzer. For a collective write in S3D I/O, a subset of MPI tasks (called *aggregator*) in each compute node communicate with other processes to exchange data and write a large chunk of data into a temporary buffer. After that, the aggregator in each node ships the I/O request to the destination I/O servers. In our configuration, we have 8 aggregator processes to perform the actual I/O operation (Figure 2.15). We observe that each process spends about 33.4 seconds (on average) in the MPI I/O library and that most of the time spent in the server layer is for disk operations. We also notice a latency gap between the MPI I/O library and the PVFS client layer (the yellow portion in Figure 2.15). In S3D I/O, all the joined processes heavily exchange the data for optimization, such as two-phase I/O [6] and data sieving [5]. This optimization and synchronization results in the overhead in the MPI library.
Fig. 2.16. Inclusive latency from the perspective of Process_320 and Process_192. The total time difference for disk operation between them is mainly caused by mpi_call_id 4 (3.31 sec. \textit{vs.} 1.99 sec.) and 8 (4.14 \textit{vs} 3.03) in 2.16(a) and 2.16(b).
Figure 2.16(a) and Figure 2.16(b) plot inclusive latencies from the perspective of Process_320 and Process_192 that have a maximum and minimum latency in the disk, respectively. In both plots, the time spent in the MPI library for mpi_call_id 0, 4, and 8 is relatively longer than that for the other I/O calls. In general, S3D I/O produces three checkpoint files using 12 collective I/O calls, and these files are generated by call ids 0~3 (first file), 4~7 (second file), and 8~11 (third file). The first I/O call (0, 4, and 8) in each checkpoint file initially opens the checkpoint file and writes the mass variable to it. Recall that, among the four arrays of mass, velocity, pressure, and temperature, mass and velocity are four-dimensional arrays whose length of fourth dimensions are 11 and 3, respectively. Since the mass array is the largest, it takes longer to be written into each checkpoint file. In the same reason, the last I/O call (3, 7, and 11) to write velocity takes relative longer time than to write pressure and temperature. In Figure 2.15, the total time difference of disk operation between Process_320 (12.68 seconds) and Process_192 (10.51 seconds) is mainly caused by mpi_call_id 4 (3.31 vs. 1.99) and mpi_call_id 8 (4.14 vs 3.03) in Figure 2.16(a) and Figure 2.16(b).

Generated by using the max query format, Figure 2.17 presents detailed I/O information ranging from mpi_call_id 0 to 3 that create the first checkpoint file. Here, mpi_call_id 0 spends considerable time in the MPI I/O library to open the checkpoint file and write data into it. Since the size of the requested I/O to write the mass array is bigger than the buffer in the MPI library, this I/O call is split into multiple subcalls. In a typical collective I/O, all processes communicate with one another and exchange access information among all processes and reorganize I/O requests for better performance. After this step, all participating processes issue the I/O requests but cannot send the
Fig. 2.17. Detailed latency from mpi\_call\_id 0 to 3 to create the first checkpoint file in Process 320. The x-axis is a pair of (mpi\_call\_id - pvfs\_call\_id) and the y-axis is the execution time in log scale. Here, mpi\_call\_id 0 and mpi\_call\_id 3 are fragmented into 11 subcalls (pvfs\_call\_id) and 3 in the MPI library, respectively. The number on the PVFS server (green bar) in the figure indicates the server ID where the I/O call has the maximum value.

next I/O requests until all finish their I/O requests. In Figure 2.17, mpi\_call\_id 0 is fragmented into eleven subcalls from (0-0) to (0-10) when writing the mass array whose length of the fourth dimension is 11, and mpi\_call\_id 3 three subcalls (3-0), (3-1) and (3-2) to write velocity whose length of the fourth dimension is 3, respectively. The latency difference between the MPI library layer and the PVFS layer, in mpi\_call\_id 0, is caused by communications and data exchanges as well as by synchronizations among the split I/O requests.

Note that the inclusive latency is computed by summing all maximum values of the corresponding split calls from the I/O call in the given process(es). Further, the
maximum latency shows more detailed information for the split calls, if any, such as individual maximum values and the server ID having a maximum among the server’s given ranges. Therefore, the inclusive latency for mpi_call_id 0 is calculated by adding the maximum values of the split calls for this I/O call in Figure 2.17. Figure 2.18 plots the disk throughput from the perspective of the server 0. Among 8 aggregator processes, Process 192 has a maximum throughput (23.44 MB/sec, 24.39, 26.83, and 27.67, respectively) for (2-0), (3-0), (3-1), and (3-2).
Fig. 2.18. Disk throughput to server 0 from all 8 aggregator processes for mpi_call_id 0 to 3 to create the first checkpoint file. The x-axis is a pair of (mpi_call_id - pvfs_call_id). Here, call id pairs of (2-0), (3-0), (3-1), and (3-2) from the Process_192 have maximum disk throughput in the server 0.
Unlike FLASH I/O, all the joined processes heavily exchange data to do optimization before sending I/O requests to the PVFS server in S3D I/O. In addition to optimization, communication and synchronization among the processes cause the overhead in the MPI library. Based on this understanding, scientists and application programmers can customize the existing code to reduce the overhead, specifically in mpi_call_id 0, 4, and 8 at the application level. Also, performance engineers may improve the performance in the MPI I/O library and disk operation at the system level.

2.5.3 Case Study: Pinpointing I/O Interference in the Concurrent Execution of Multiple Applications

In HPC systems that share I/O system resources across processes, interference occurs when multiple applications access a storage resource, which in turn, causes substantial I/O performance degradation. To simulate this real case scenario and to profile detailed metrics in such a situation, first, we separately run two benchmarks, S3D I/O and a synthetic benchmark and measure the execution of each benchmark, as a baseline experiment. We run S3D I/O with the same configuration as in Section 2.5.2, using 512 MPI processes with 1 metadata server and 8 I/O servers. Here, the synthetic benchmark accesses data in row, column, and block fashion, and generates a 2 GB checkpoint file. At this time, we run the synthetic benchmark on 64 MPI processes, but it only stripes data over 1 I/O server, by setting MPI hints. After that, we run both benchmarks at the same time so that I/O operations are interfered in each other. S3D I/O accesses 8 I/O servers to write data and the synthetic benchmark only stripes 1 I/O server among
Fig. 2.19. Comparison of the execution time and the maximum I/O time in servers. In Figure 2.19(a), the execution time of S3D I/O and the synthetic benchmark is 39.63 seconds and 36.43, and, in Figure 2.19(b), 56.18 and 61.11, respectively. In both the experiments, the corresponding detailed I/O server time and striped data size are described in Table 2.4 and Table 2.5.

8 I/O servers. The compute node was not overlapped when running concurrently in this experiment.

Figure 2.19 compares the execution time and the I/O time in the server when each benchmark runs separately. In Figure 2.19(a), considered as the baseline, the execution time and the maximum server I/O time are 39.63 seconds and 33.29 in S3D I/O and 36.43 and 35.64 in the synthetic benchmark, respectively. Table 2.4 presents detailed metrics in S3D I/O. When I/O operations are interfered, as shown in Figure 2.19(b), the execution time increases up to 56.2 seconds in S3D I/O and 61.1 in the synthetic benchmark (see Table 2.5). Therefore, the overhead of the execution time caused by I/O interference are 42% and 68%, respectively. In this scenario, the data from S3D I/O are evenly striped to 8 I/O servers, about 732 MB each. At the same time, the synthetic benchmark accesses one of the 8 I/O servers to write a 2 GB checkpoint file. This I/O
Table 2.4. Baseline: S3D I/O detailed server I/O time and striped size of data.

<table>
<thead>
<tr>
<th>Server</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>33.29 sec</td>
<td>33.28</td>
<td>33.27</td>
<td>33.28</td>
<td>33.28</td>
<td>33.27</td>
<td>33.29</td>
<td>33.28</td>
</tr>
</tbody>
</table>

Table 2.5. Running S3D I/O with the synthetic benchmark in interference.

<table>
<thead>
<tr>
<th>Server</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>46.31 sec</td>
<td>46.22</td>
<td>46.26</td>
<td>46.27</td>
<td>46.24</td>
<td>46.21</td>
<td>46.15</td>
<td>46.20</td>
</tr>
</tbody>
</table>

A server is a bottleneck and causes the degradation of the overall I/O performance in both applications.

Table 2.6. Running S3D I/O with the synthetic benchmark \textit{without} interference.

<table>
<thead>
<tr>
<th>Server</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>35.64 sec</td>
<td>32.74</td>
<td>32.72</td>
<td>32.72</td>
<td>32.72</td>
<td>32.74</td>
<td>32.73</td>
<td>32.73</td>
</tr>
<tr>
<td>Size</td>
<td>2 GB</td>
<td>837 MB</td>
<td>837 MB</td>
<td>837 MB</td>
<td>837 MB</td>
<td>837 MB</td>
<td>837 MB</td>
<td>837 MB</td>
</tr>
</tbody>
</table>

Based on this observation, a new I/O strategy can be adopted to prevent the interference. By setting up the MPI hints not to stripe the bottleneck I/O server, thus striping data only to 7 I/O servers, the execution time S3D I/O are 42.35 seconds, and the I/O time and striped data size in server is presented in Table 2.6. Note that the execution time of S3D I/O with 7 I/O servers increases about 7% compared to the baseline. By ensuring that the two applications do not interfere with each other however, one can eliminate performance degradation.
2.6 Conclusions

Performance analysis and visualization is an important step in understanding I/O behavior, which is a result of complex interactions in the I/O stack. Performing manual code instrumentation is often difficult and extremely error-prone. Even building the I/O stack and configuring the running environment for application benchmarks is not trivial because of the scale of the current HPC systems. Moreover, collecting and analyzing trace data from them is challenging and daunting task. To alleviate these difficulties, we have developed a parallel I/O profiling and visualizing framework, IOPro. Our tracing utility uses existing MPI I/O function calls and therefore adds minimum overhead to the execution time of applications. Our framework provides multiple metrics to analyze and investigate detailed I/O behavior, including latency, throughput, energy consumption, and call information. The results from these metrics contribute to evaluating and explaining the parallel I/O behavior.

We used two application benchmarks, S3D I/O and FLASH I/O, to evaluate our implementation of IOPro. Our experiments demonstrate different I/O behaviors in each application: S3D I/O exchanges data among the joined processes to do optimization and synchronization in the MPI library whereas FLASH I/O rarely does such optimization. Although both applications issue collective I/O requests to write the checkpoint files, the characteristics of I/O are different in each benchmark. By using the performance information depending on the I/O behavior, the application programs can be optimized to improve the performance. Also, customized instrumentation can be performed to get more detailed performance statistics in the I/O stack.
Lastly, we show that, when multiple applications interfere each other due to sharing I/O system resources, our framework can be used to profile detailed performance metrics, aid in understanding complex I/O behavior, and detect the issue that degrades the performance. Based on the gleaned information, the user can then employ an appropriate solution.
Chapter 3

IOPin: Runtime Profiling of Parallel I/O in HPC Systems

Many I/O- and data-intensive scientific applications use parallel I/O software to access files in high performance. On modern parallel machines, the I/O software consists of several layers, including high-level libraries such as Parallel netCDF and HDF, middleware such as MPI-IO, and low-level POSIX interface supported by the file systems. For the I/O software developers, ensuring data flow is important among these software layers with performance close to the hardware limits. This task requires understanding the design of individual libraries and the characteristics of data flow among them. In this chapter, we discuss a dynamic instrumentation framework, called IOPin, that can be used to understand the complex interactions across different I/O layers from applications to the underlying parallel file systems. Instead of manually instrumenting applications and other components of the I/O stack, we leverage a lightweight binary instrumentation using probe mode in Pin [22] to implement our current prototype. That is, IOPin performs the instrumentation with minimal overhead in the binary code of the MPI library and the underlying parallel file system at runtime. Our preliminary experience indicates that the costs of using the proposed dynamic instrumentation is about 7% of the application execution time. Therefore, it provides the language-independent instrumentation targeting scientific applications written in C/C++ and Fortran. Furthermore, our tool
requires neither source code modification nor recompilation of the application and the I/O stack components.

A unique aspect of our runtime profiling framework is that it provides a hierarchical view for parallel I/O. As in IOPro, each MPI I/O call has a unique identification number in the MPI-IO layer and is passed to the underlying file system with trace information. This mechanism helps associate the MPI I/O call issued from the applications with its sub-calls in the PVFS layer in a systematic way. In addition, our tool provides detailed I/O performance metrics for each I/O call, including I/O latency at each I/O software stack layer, the number of disk accesses, disk throughput, the number of I/O calls issued to the PVFS server.

3.1 Overview of Dynamic Instrumentation

The main goal behind this work is to understand the I/O characteristics of parallel applications, by detecting a “critical I/O path” at runtime from the process to the parallel file system that affects the entire system performance. Based on the knowledge about I/O behavior, application programmers and scientists can optimize performance by redesigning applications or system architecture. Our current prototype exploits Pin [22], a lightweight binary instrumentation tool to instrument the binary code of the MPI library and PVFS. As a result, our tool does not require source code modification and recompilation of the I/O software stack components.

Figure 3.1 shows the overview of our Pin-based framework. This figure is intended to explain the flow of MPI I/O call and how the framework carries out the dynamic instrumentation when a collective write function is issued. In the figure, two Pin profiling
Fig. 3.1. Overview of our dynamic instrumentation framework. The client Pin process creates trace information for the MPI library and PVFS client at the boundary of each layer, and send it to the client log manager. The server Pin process produces trace information—the latency spent in the server, processed bytes, the number of disk accesses, and I/O throughput—and transmits it to the server log manager.

Processes on the client side and the server side generate trace log information at the border of each layer—the MPI library, PVFS client, and PVFS server. The log on the client side contains trace information of each layer such as rank, mpi_call_id, pvfs_call_id, I/O type (read/write), and latency spent in the MPI library and PVFS client. In the server log with these metrics, additional information is also sent to the server log manager, such as pvfs_server_id, latency in server, bytes to be read/written, the number of disk accesses, and disk throughput for the MPI I/O call at runtime.

Both log managers are implemented in SQLite [30], a software library that implements a SQL database engine. Each log manager sends the record information back to the corresponding Pin process that has a maximum latency for the I/O operation. Then, the Pin identifies the process that has a maximum I/O latency from it, and traces
and instruments only this process. This selective dynamic instrumentation not only reduces overheads, but also detects only one critical I/O path that affects the system performance effectively in the I/O stack.

3.2 Backgrounds

In this section, we briefly discuss dynamic binary instrumentation, Pin instrumentation, and parallel I/O operation widely exploited in HPC.

3.2.1 Overview of Dynamic instrumentation

Dynamic binary instrumentation (DBI) is a popular technique to analyze the software behavior at runtime through the injection of instrumentation code. The instrumentation code is executed in the application’s address space as a part of the normal instruction stream after injection and is entirely transparent to the applications. In this reason, DBI is widely used to implement the instrumentation platforms that provide an API to facilitate the development of instrumentation. Using the instrumentation platforms, a lot of analysis and profiling tools are developed for cache simulation [31], memory allocation error [32] and leak detection, security violation detection [33], and modeling of system performance [34].

Observing and analyzing the application behavior during execution makes it possible for software developers and scientists to gain insight into the characteristics and state of the application at various running points. Since the usability of DBI-based profiling and analysis tool heavily depends on its overhead caused by itself, the platform developer particularly focuses on improving the performance of DBI [35].
3.2.2 Overview of Pin

Pin is a software system that performs runtime binary instrumentation of Linux and Window applications. The goal of Pin is to provide an instrumentation platform for implementing a variety of program analysis tools for multiple architectures. Pin provides a rich API that observes all the architectural state of a process such as the contents of registers, memory, and control flows. Also, the Pin API makes it possible to write portable instrumentation tools (called Pintools). In Pin, user may add analysis routines to the application process, and write instrumentation routines to determine where the analysis routines are called. Pin also provides a limited ability to alter the program behavior by allowing an analysis routine to overwrite the registers and memory.

Instrumentation is performed by a just-in-time (JIT) compiler. The input to this compiler is not bytecode, but a native executable. Pin intercepts the execution of the first instruction of the executable and generates (“compiles”) new code for the straight-line code sequence starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Every time JIT fetches some code, the Pintool has the opportunity to instrument it before it is translated for execution. The translated code and its instrumentation is saved in a code cache for future execution of the same sequence of instrumentation to improve performance. Our initial evaluation for the parallel I/O application in JIT mode shows that the overhead ranges from 38.7% to 78% of the application execution time.
Application binary is also instrumented in Pin probe mode. Probe mode is a method of using Pin to insert probes at the start of specified routines. Here, a probe is a jump instruction that overwrites an original instruction in the application. Before the probe is inserted, the first few instructions of the specified routine are relocated. Pin copies and translates the original bytes of the application binary and then the probe redirects the flow of control to the replacement function. After instrumentation the control flow returns to the original function. Therefore, in probe mode, the application and the replacement routine are run natively. This improves performance, but it puts more responsibility on the tool writer. Many of the Pin APIs that are available in JIT mode are not applicable in probe mode. In this work, IOPin is implemented in probe mode.

3.2.3 Critical Path Detection Affecting Parallel I/O Performance

On a modern cluster today that is a generic client/server architecture for scientific applications, the entire application performance heavily depends on the parallel I/O performance in the system. In many parallel applications, to reduce the high I/O latency, the requests from different processes are frequently interleaved and merged into contiguous portions of the file. When such optimization, broadly referred to as collective I/O, is used, all the joined processes broadcast and exchange the information related to the I/O request. If the I/O access pattern of all processes is contiguous and can have benefits from collective I/O, the aggregator process in Figure 3.2 can access the data by two-phase I/O — (1) redistribution of data to the processes (communication phase) and (2) a single, large, contiguous access to data (I/O phase) in case of write
Fig. 3.2. The critical path affecting the application performance. Among the number of M processes and N servers, a MPI I/O call from the $i^{th}$ process (aggregator) to the $j^{th}$ server dominates the entire application performance for the given collective I/O call.

This method slightly adds the cost of interprocess communication among the joined processes, but it can significantly reduce the I/O time. Although collective I/O is performed to improve I/O latency, the performance of collective I/O can be significantly affected by the critical path from the process to the server for the given I/O. If the process on the critical path has a small size of temporary buffer needed for two-phase I/O and frequently copies the data into the buffer, and communicates other processes for redistribution, it can degrade the performance. In this case, the critical path from the aggregator process $i$ to the server $j$ can dominate the overall application performance in Figure 3.2. Also, the I/O server on the critical path can be bottleneck in certain reasons such as network problem, explosion of I/O requests to the server, or faulty I/O
server. Our implementation may perform optimization at process level by changing two user-controllable parameters specified in MPI-IO hints for collective I/O at runtime: the number of processes that perform I/O operation in the I/O phase and the temporary buffer size (4 Mbytes by default) needed for two-phase I/O on each process.

3.3 Technical Details

We provide here details about dynamic code instrumentation and computation methodology for latency and throughput.

3.3.1 Detailed Dynamic Instrumentation

Figure 3.3 illustrates in detail how our implementation performs dynamic instrumentation. When an MPI I/O function call is issued from the high-level I/O library or application, the Pin process on the client side generates trace information, including rank, mpi_call_id, pvfs_call_id, I/O types (read/write), and timestamp in the MPI library. By definition, the MPI I/O function call is replaced with PVFS_sys_io function in the MPI library with additional arguments (PVFS_IO_WRITE and PVFS_HINT_NULL) to be issued to PVFS client. Here, the Pin process packs the trace information into a PVFS_hints structure and replaces the last argument, PVFS_HINT_NULL (initially set to NULL by default), with the Pin-customized hint in the PVFS_sys_io(). In the PVFS client, the Pin process extracts the trace information from hints and stores the trace information in the buffer to calculate latency later. The Pin-defined hint is encapsulated into a state machine control block (smcb) structure and passed to the PVFS server.
Fig. 3.3. Detailed illustration of how the trace information is passed. The Pin process creates a PVFS hints structure that contains rank, mpi_call_id, and pvfs_call_id. It then replaces PVFS_HINT_NULL in PVFS_sys_io() with the Pin-generated PVFS hints.
At the starting point of server, the Pin process searches a customized PVFS_hints from the first argument (*smcb) and extracts the trace information. For each I/O operation, PVFS server maintains a flow_descriptor structure from smcb. This flow_descriptor includes all information about the corresponding I/O request and flows until the end of the I/O operation. Since the Pin-customized hint containing the trace information exists in flow_descriptor, the server Pin process can extract it from hints in flow_descriptor at any point in the server without complexity.

At the entry point of disk write operation, **trove_write_callback_fn()**, the Pin process acquires the address that points to the flow_descriptor from the first argument (void *user_ptr) in the function. It then finds the PVFS_hints from it and stores disk I/O information, including the bytes processed, the number of disk accesses at the end of the disk operation with the corresponding rank, and the id information extracted from hints.

At the exit point of the server, Pin produces the log information with necessary information, e.g., rank, mpi_call_id, pvfs_call_id, I/O type, bytes processed for the corresponding MPI I/O operation, the number of accesses to disk, latency spent in the server, and disk throughput. This server log information is sent to the server log manager. Again, the Pin process on the client side generates a log at the exit point of the layer and sends it to the client log manager.

The client log manager sends the record information back to the client Pin process that has a maximum latency for the I/O operation. The client Pin detects the MPI process that has the maximum I/O latency, and traces and instruments only this process. The server side Pin also identifies the I/O server that spends the longest time to handle
Fig. 3.4. Comparison of S3D I/O execution time. This figure shows the execution time running on un-instrumented I/O stack and Pin-instrumented. The overhead caused by Pin-instrumentation in probe mode is about 7%, on average.

the I/O request. Our selective dynamic instrumentation not only reduces the overhead, but also effectively detects only one “critical I/O path” to the server among hundreds of thousands processes that affects the system performance in the I/O software stack.

At the end of the execution, by simply associating the mpi_call_id and pvfs_call_id in the client with the one in the server, the entire I/O path from the MPI library to PVFS server can be traced with the performance metrics. The detailed computation methodology for the performance metrics is explained in Section 2.4.3.

3.4 Evaluation

Our dynamic instrumentation framework for the parallel I/O application is evaluated on the Breadboard [27] cluster at Argonne National Laboratory (ANL). Each
(a) Execution time of S3D I/O.  
(b) I/O throughput of S3D I/O.

Fig. 3.5. Comprehensive results drawn by IOPin. In (a), for each mpi_call_id, the latency spent in MPI, client, and server is plotted in order. The latencies for some mpi_call_id’s in client and server are barely seen because they are less than 0.1 sec. The figure on the server bar implies the number of fragmented calls (sub-calls). In (b), the throughput of mpi_call_id 0, 4, and 8 is plotted cumulatively even if they are split into 3 sub-calls.

node of this cluster consists 8 quad-core Intel Xeon Processors and 16 GB main memory. Therefore, each physical node can support 32 MPI processes. We evaluated our implementation running on 1 metadata server, 8 I/O servers, and 256 processes. In our evaluation, we use pnetcdf-1.2.0 as a high-level I/O library, mpich2-1.4 as a middleware, and pvfs-2.8.2 as a parallel file system. To demonstrate the effectiveness of the framework, we tested a I/O-intensive benchmark, S3D-IO [29].

S3D I/O is the I/O kernel of S3D application, a parallel turbulent combustion application using a direct numerical simulation solver developed at Sandia National Laboratories (SNL). A checkpoint is performed at regular intervals; its data consists primarily of the solved variables in 8-byte, three-dimensional arrays. At each checkpoint, four
global arrays—representing the variables of mass, velocity, pressure, and temperature—are written to files. All four arrays share the same size for the lowest three spatial dimensions X, Y, and Z and are partitioned among the MPI processes along with X-Y-Z dimensions. In our evaluation, we maintain the block size of the partitioned X-Y-Z dimension as $200 \times 200 \times 200$ in each process. With the PnetCDF interface, it produces three checkpoint files, 976.6MB each.

Figure 3.4 compares the execution time of S3D I/O when running on un-instrumented I/O stack and dynamically instrumented I/O stack. We observe that, with the process counts of 32, 64, 128, and 256, the average overhead incurred by our proposed dynamic instrumentation is about 7%.

Plotted in Figure 3.5(a) is the latency spent in the MPI library, PVFS client, and PVFS server from the perspective of one of the aggregator processes on a critical I/O path among 256 processes. Note that a large fraction of the time spent in the server is for disk operations even though not shown here. In S3D I/O, three checkpoint files are produced by 12 collective I/O calls, and each checkpoint file is generated by 0∼3, 4∼7, and 8∼11, respectively. For example, the first checkpoint file is opened by mpi_call_id 0. The four arrays of mass, velocity, pressure, and temperature are sequentially written by the mpi_call_id 0, 1, 2, and 3. We observe from Figure 3.5(a) the latency difference between the MPI library and the PVFS client. During the collective I/O operation in S3D I/O, all the joined processes heavily exchange data for optimizations such as data sieving [5] and two-phase I/O [6]. In addition, communication and synchronization among the processes cause the overhead in the MPI library. We also notice that the latency in the MPI library for mpi_call_id 0, 4, and 8 is longer than that of the others.
These calls are to open the individual checkpoint file and to write the mass value which is the largest array among the four. In our experiment, these calls are fragmented into 3 sub-calls to satisfy the I/O requests. The figure on the server bar in Figure 3.5(a) indicates the number of fragmented calls which is also the number of disk accesses.

Figure 3.5(b) plots the throughput of an individual I/O call from mpi_call_id 0 to 11. The first calls (0, 4, and 8) to create the individual checkpoint file are split into 3 sub-calls, respectively, and the throughput of those I/O calls is plotted cumulatively. We observe that the I/O throughput to for creating and writing the first file is higher than the others, on average, which needs to be further investigated.

Based on the understanding of I/O characteristic from the given applications, scientists and application programmers can customize the existing application code to better use the middleware. Also, performance engineers may reduce the the overhead caused by such optimizations in the MPI library.

3.5 Conclusions

Understanding I/O behavior is one of the most important steps for efficient execution of data-intensive scientific applications. The first step in understanding I/O behavior is to instrument the flow of an I/O call. Unfortunately, performing manual instrumentation is extremely difficult and error-prone since the characteristics of I/O are a result of complex interactions of both hardware and multiple layers of software components. Because of the scale of the current HPC systems, collecting and analyzing trace information are challenging and daunting tasks. To alleviate these difficulties, we propose a dynamic instrumentation framework working on the binary code of the MPI
library and PVFS. The tool inserts trace information into a PVFS_hints structure and passes it into the sub-layers at runtime. This method can provide a hierarchical view of the I/O call from the MPI library to the PVFS server without source code modification or recompilation of the I/O stack.

We used a scientific application benchmark, S3D I/O, to evaluate our proposed framework. Changing the number of processes to run S3D I/O, we made different experiments and observed that the overhead induced by our implementation is about 7% on average. Our tool provides several metrics to understand and analyze I/O behavior, such as the latency of each layer, the number of fragmented I/O calls and disk accesses, and I/O throughput. The results from these metrics contribute to evaluating and tuning the applications and I/O software stack.
Chapter 4

IOGenie: Automatic Parallel I/O Code Generation and Optimization framework for HPC Applications

In this chapter we introduce an automatic parallel I/O code generation framework, called IOGenie, which generates the efficient and optimized code for the scientific parallel applications written in C/C++ and Fortran. Using a graphical user interface, our tool takes high-level annotations for I/O as input, analyzes the given options, and generates optimized I/O code that effectively exercises the underlying I/O stack. This tool helps users write data-intensive applications easily and effectively. Also, it enhances the quality of tool-generated code that exploits various optimizations on the underlying I/O software. Our experience indicates that the overhead of running tool-generated applications is negligible.

4.1 Introduction

In response to the need for complex and highly scalable HPC applications, today’s parallel computing offers a comprehensible portfolio of computing and storage resources. In such platforms, design and implementation of data-intensive parallel applications should provide high levels of reliability, efficiency, availability, and scalability. Accordingly, novel approaches and numerous optimizations have been investigated at different layers such as intermediate libraries and parallel file systems to improve the I/O performance and overcome the I/O bottleneck. To map a given application onto
the target I/O system and coordinate its accesses to parallel file systems, different I/O software layers are developed and built in the I/O stack that consists of high-level I/O libraries such as PnetCDF and HDF5 as well as middleware like MPI-IO, as shown in Figure 1.1. These libraries/middleware are widely used in a variety of scientific domains to achieve high-performance.

Considering numerous APIs and their complex interactions among these libraries, it is difficult for users to understand their details and interactions. In addition, there exist various optimization opportunities across I/O libraries and parallel file systems, which helps improve I/O performance. Thus, writing a parallel I/O code that efficiently exploits these optimizations is a very challenging task. In particular, it is not unusual for an average (non-I/O expert) user to make mistakes in orchestrating I/O accesses in the code. Further, such a user may not be able to take full advantage of I/O optimization opportunities within and across different layers of the I/O stack. Writing correct and optimized I/O code is a serious challenge even for experienced programmers.

In this chapter, we discuss a GUI-based I/O code generation and optimization framework, called IOGenie, which automatically generates parallel I/O code for any I/O stack, based on the user-supplied input parameters—high-level hints and annotations. As shown in Figure 4.2, our prototype of IOGenie consists of two main components, namely, a graphical user interface (GUI) and code generation engine. The front-end GUI accepts high-level hints provided by users and the back-end code generation engine performs the actual code generation. The I/O hints taken from the GUI include information such as the I/O library to be used, datasets definition, data access patterns at each I/O point, and data layouts in the storage nodes. The code generation engine, then, analyzes the
given hints and formulates the optimized code at each I/O point that users specify in the application. Note that, unlike MPI hints to be used by users to optimize file access, the hint in this dissertation is an interface for users to pass general information to our tool. Also, when there is no confusion, the hint(s) and annotation(s) are used interchangeably.

The main goal of our tool is to improve programmers’ productivity and enhance the code quality of scientific applications. The GUI provides a convenient and user-friendly interface for users to enter hints. Also, the format of high-level hints supported by our tool provides a good abstraction. Thus, users can easily specify I/O operations to be performed in the application. Using the given hints, our tool then generates an optimized I/O code considering the I/O access patterns of the application as well as the potential optimizations presented by the underlying I/O libraries.

4.2 OVERVIEW

In this section, we briefly discuss an architecture model in HPC, a parallel I/O software stack, and parallel I/O operations widely exploited in HPC.

4.2.1 HPC Architecture and I/O Software Stack

Most large-scale data-intensive applications run on internet services (or “cloud computing”) or high-performance computing (HPC) systems. In this work, we assume data-intensive applications are written in Message Passing Interface (MPI) [19, 36], a dominant parallel programming model in large-scale. Also, our tool targets scientific applications running on HPC systems. Traditionally, HPC is defined by parallel scientific applications that is normally deployed in separate compute nodes and storage nodes with
interconnect network, as shown in Figure 4.1. In HPC, processes of an application run on the compute nodes in parallel. These processes heavily perform communications, computations, and synchronization with each other. During the execution, they issues I/O to the servers when I/O is required. The I/O requests are delivered to the I/O servers through the interconnect network. The servers, then, dispatch the requests to the disks attached to them. Again, the responses of the requests are sent back to the compute nodes from the I/O nodes through the network.

Parallel I/O operation places a tremendous burden on application programmers since it is difficult to be coordinated and optimized. To facilitate parallel I/O, scientific applications running on HPC systems exercise a number of intermediate layers, called I/O software stack, as presented in Figure 1.1.
4.2.2 Overview of IOGenie

The high-level view of IOGenie is presented in Figure 4.2. It comprises two main components including a front-end graphical user interface (GUI) and a back-end code generation engine. The GUI accepts as inputs application code without I/O, I/O level, and I/O hints. Here, the application code is in the form of a template that only contains computation code, not including I/O operations. The I/O level indicates the layer and the library that should perform I/O in the I/O software stack. Lastly, I/O hints specify the necessary I/O information, e.g., data set definition and data access pattern. This input information is utilized by IOGenie to guide I/O code generation.

More specifically, these input parameters are passed to the code generation engine that accommodates a hint parser, a hint analyzer, an I/O optimizer, and a code generator. First, the hint parser breaks the given hints down. Then, the hint analyzer resolves them. The I/O optimizer tries to improve the quality of the I/O operations by employing optimization techniques applicable in the selected I/O library. Finally, the
code generator brings out the application code that includes the generated I/O operation code as well as the original application code with computation. Note that the GUI interface is developed to simplify and facilitate the use of our tool.

4.3 OUR I/O MODEL

In this section, we introduce an I/O model that can be used to specify the types of data sets and data access patterns. Since the I/O model employed by IOGenie is unified, it covers most common types of data sets and access patterns for different I/O libraries. Also, it determines the formats of the I/O hints. Therefore, our tool can generate parallel I/O code for different I/O libraries from the same set of I/O hints. The I/O model supported by our tool accommodates two parts: 1) data set definition model and 2) data access model.
4.3.1 Data Set Definition Model

Multidimensional arrays are very common in scientific applications to store data. They are mapped into one dimensional machine memory layout. Mapping of multidimensional arrays into one dimensional memory dramatically affects the performance of the parallel applications.

The data set definition model in our tool supports the multidimensional array representation. This multidimensional array is the primary data set format supported by high-level I/O libraries such as PnetCDF and HDF5. A multidimensional array has two key components, namely, element type and data space. The element type defines the data type of each element in the array, whereas the data space specifies the number of dimensions and the length (extent) of each dimension of the array.

We use a tuple $\langle \text{datatype}, \text{dimension} \rangle$ to describe a multidimensional array, where datatypem is int, double, and other basic types, and dimension is itself a tuple of the form $\langle \text{dimension1}, \text{dimension2}, \ldots \rangle$. Figure 4.3 shows two examples of multidimensional arrays. Figure 4.3(a) is a two-dimensional array, and Figure 4.3(b) is a three-dimensional array. In this example, three dimensional array can be described as a tuple $\langle \text{int}, \langle 5, 5, 5 \rangle \rangle$. The data space can also be a supercube when the number of dimensions exceeds three.

We also support the concept of attribute to comply with metadata provided by high-level I/O libraries such as PnetCDF and HDF5. Metadata is used to describe the properties of the multidimensional array. It also provides additional information as to how to interpret the data stored in the array. For example, assuming the temperature
data is stored in a multidimensional array, one attribute can be defined to specify the unit of the temperature, e.g., Fahrenheit or Celsius.

### 4.3.2 Data Access Model

In addition to the mapping of multidimensional arrays to flat memory, the order in which array entries are accessed has an effect on the behavior of our target applications. For example, *strided* access to one-dimensional array reduces spacial locality and ultimately degrades utilization of memory bandwidth.

The data access model in this work is used to specify how a data set is accessed, that is, the portion of data accessed by each process. For simplicity and conciseness, the same access model is applied for both read and write. Our data access model supports subarray, one of the most common data access patterns in HPC applications. To describe a subarray, the length of each dimension and the offset coordinates (the smallest indices in each dimension of the data set) are required. In our model, we use a tuple.
\(< \text{offset}, \text{length} >\) to determine a subarray, where \text{offset} and \text{length} are also tuples, that is, \(< \text{offset}_1, \text{offset}_2, \ldots \rangle \) and \(< \text{length}_1, \text{length}_2, \ldots \rangle\), respectively.

There are two alternate methods for users to specify the data access patterns for the processes that the application runs on. If the number of processes is small and the access patterns to data are irregular from one another, users will specify them by providing specific information for each individual process. If the number of processes is large though, for example, if hundreds of thousands of processes are spawned in a large scientific application, entering the access information individually for each and every process may not be feasible. Luckily, in most of such applications, the data access patterns from the processes are usually regular. IOGenie, thus, employs a compact data access model so that users can specify the access patterns as a whole for all processes from a global view. The access pattern of each individual process is easily derived from the global compact specification.

Figure 4.4 shows two examples for irregular and regular access patterns from four processes, respectively. To describe the access patterns in Figure 4.4(a), the offset and the length of each dimension for all processes (\(P_1, P_2, P_3,\) and \(P_4\)) are necessary. For example, the access pattern of process \(P_2\) in Figure 4.4(a) can be represented using \(<< 0,3 >,< 1,1 >>\). In Figure 4.4(b), we can use a single tuple \(< \text{processes}, \text{offsets}, \text{lengths}, \text{strides} >\) to express the access patterns, where \text{processes} is the number of processes, \text{offsets} is the smallest indices of all processes, \text{lengths} is the length in each dimension (identical to all processes), and \text{strides} is the access distance between different processes in each dimension. In this case, the tuple is \(<< 2,2 >,< 1,1 >,< 2,2 >,< 3,3 >>\).
The I/O model discussed above is a general model that can be adapted to work with different I/O libraries. Note however that some libraries possess unique features that do not exist in other libraries. These featured data set definitions or access patterns can not be represented by the general I/O model. To address this limitation and make the functionality of our tool more complete, we design IOGenie to enable future extensions. For example, HDF5 supports “compound” data type similar to a structure in C or a common block in Fortran. It is a collection of one or more datatypes. Using our tool, users can describe more complex data structures and access patterns; each such composite type is a combination of two or more atomic types. We, thus, include I/O model to be compatible with this specific feature for HDF5 library.

4.4 FRAMEWORK

In this section, we discuss the front-end components of IOGenie, which consist of I/O hints and I/O options. We also elaborate on the back-end code generation engine that comprises the hint parser, hint analyzer, I/O optimizer, and code generator.

4.4.1 I/O Hints

Based on the general I/O model presented in Section 4.3, we next define the format of the I/O hints. According to the I/O model, there are two groups of I/O hints for data definition and data access.

Listing 2 presents the format of our data definition hints. The data definition group includes dimension hints, variable hints, and attribute hints that are used together to specify the data that will be accessed by the application. Specifically, the
dimension hint provides data space information. The variable hint defines the data type and the data space for each variable. The attribute hint adds additional information to interpret data. Lastly, read and write hints describe the portion of the data to be accessed. As presented in Listing 2, each hint is composed of an operator and several parameters. Operator indicates the type of I/O operations, and each parameter specifies its variable information. Note that the exact locations of these hints make differences in the sense that they determine the location to define specific data and to access them in the program. The directives starting with ‘#’ are used by the hint parser to locate these hints.

To specify these hints, users may choose to type them directly to the proper locations of the original source code. Alternately, users can give data set definitions and data write information through the GUI, as shown in Figures 4.5 and 4.6, respectively. Compared to the manual typing, this GUI-based input method can be safer and easier since it provides user-friendly and intuitive dialogues with clues. This latter option will be particularly helpful and efficient when users are not quite familiar with the formats of our I/O hints.

4.4.2 I/O Options

In addition to the I/O hints, additional information is needed to enable our framework to generate the desired I/O operations. If it is necessary to include additional guidances such as the I/O level, the I/O library (e.g., PnedCDF or HDF5), and the type of source code (e.g., C/C++ and Fortran), users can also specify these options using the GUI. Figure 4.7 shows the GUI view for entering I/O options.
Fig. 4.5. GUI view for giving the dataset hint. Users can enter dimensions, variables, and attributes to be accessed by the application.
The I/O level option determines the layer on which code-level I/O operations should be performed in the I/O stack. The I/O library option indicates the library for which I/O operation should be generated. Our prototype currently supports I/O operations written in C/C++ and Fortran. Using the language option, it provides code generation with the same library but written in different languages. Again, users can either directly enter these options into the source code or utilize the dialogue in the GUI to provide them. For example, if a user develops a C program that needs to handle data at a high abstraction level and the PnetCDF library is available, one may give the following options— the high-level I/O library with PnetCDF, and C as the programming language.

4.4.3 Code Generation Engine

The code generation engine is a core part (back-end) of our tool. It automatically generates parallel I/O code based on the I/O hints and annotations provided by the
users. As described in Figure 4.2, it consists of four main components: hint parser, hint analyzer, I/O optimizer, and code generator. The hint parser reads the hints that are inserted by the users. Then, the hint analyzer extracts necessary I/O information from the hints. Next, the I/O optimizer performs optimizations based on the extracted I/O information and the I/O library specified by the user. Finally, the code generator translates the results from the previous steps into parallel I/O code and inserts that code into the target application code.

4.4.3.1 Hint Parser

As presented in Listing 2, a number of hints are grouped together as hint sets if the I/O operations related to them are required at the same location in the application code. Each hint set has three parts, namely, set head, set body, and set tail. The hint parser recognizes the set body with the aid of set head and set tail. It then processes the hints one after another and translates them into internal hint representations.
4.4.3.2 Hint Analyzer

The hint analyzer takes the internal hint representations passed from the hint parser, and tries to make connections among different I/O hints. For example, the analyzer associates the hint that specifies variable access with the hint that indicates variable definition. One advantage of doing this is that the hint formats can be compacted so that users do not need to input redundant information repeatedly.

4.4.3.3 I/O Optimizer

The I/O optimizer tries to further improve the internal hint representations by taking into account the access patterns of all the processes and the available optimization techniques in the chosen library. Note that the optimizations in the I/O library do not necessarily have positive effects on the performance, depending on whether the benefits brought can offset the cost incurred. Usually, it is the user’s responsibility to decide whether to apply a specific optimization or not. Considering the situation that a user may be unfamiliar with the I/O library, these possible optimizations can be either neglected or inappropriately used. For example, in the I/O libraries such as PnetCDF, HDF5, and MPI-IO, choosing between collective I/O and independent I/O is not trivial. To solve this problem, the I/O optimizer takes advantage of the input I/O information and automatically decides whether to apply the available optimizations or not. Depending on the decision, the optimizer then reconstructs the internal hint representations to include optimization information that will be carried out by the code generator.
Fig. 4.8. Sample code generation by IOGenie, assuming an application written in C and PnetCDF as the high-level I/O library. Considering I/O hints, such as dimensions, variables, and attributes, starting with “#” in the left window, our tool generates the corresponding I/O code on the right.
4.4.3.4 Code Generator

The code generator translates the internal I/O representations into the target parallel I/O code and inserts it into the application source code. The generator checks the operations in the I/O hints along with associated parameters and replaces them with the corresponding APIs of the chosen I/O library and language. In addition, it generates optimized code by taking into account the optimization decisions from the I/O optimizer. For example, if the target I/O code is for PnetCDF and the optimizer has decided that collective I/O is more beneficial at a certain point in the application, then the data access APIs in the generated code will end up with "all" for collective I/O.

Figure 4.8 shows a sample I/O code generation performed by our framework. In this example, the application is written in C and utilizes PnetCDF as the high-level I/O library, on top of PVFS. Running on PnetCDF, an application first enters ‘define’ mode to describe all attributes, dimensions, types, and structures of variables. The program will then exit it and enter ‘data’ mode, where it performs I/O. As the first step, the necessary hints to define dimensions, variables, and attributes are taken on the left side window, through the GUI view shown in Figure 4.5. Then, our tool transforms the hints internally and generates the I/O code on the right side.

The goal of IOGenie is not only to improve the productivity of code generation, but also to ensure the quality of the generated I/O code. The latter one is achieved via the I/O optimizer component in the code generation engine as discussed above. An important point to note is that the optimizations in our optimizer are accomplished for
user level applications, not the I/O library. Thus, our tool will never modify the I/O libraries under any circumstance.

4.5 EVALUATION

Our implementation for automatic code generation and optimization is evaluated on the Breadboard cluster at Argonne National Laboratory [27]. This cluster is configured as follows. There are 1 metadata server and 8 I/O servers, each with 4 1GHz Dual-Core AMD Opteron processors and 4GB of RAM. Each of the compute nodes has 16 2.4GHz Intel Xeon processors with 24GB RAM and, thus, can host 64 MPI processes. We built an I/O stack that consists of PnetCDF-1.2.0, mpich2-1.4, and pvfs-2.8.2. We evaluated our implementation with various number of MPI processes, 64, 128, 256, and 512. In the evaluation, we used two parallel scientific applications that heavily issue I/O operations: FLASH I/O and S3D I/O.

4.5.1 FLASH I/O

The FLASH code [24] is an adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs. A typical large production run of FLASH will generate about 500GB of data, distributed between 1,000 plot files and 100 checkpoint files. Since I/O accounts for much of the running time, we only run the FLASH I/O benchmark, created to test I/O performance of FLASH independently of the entire code. The computational domain is divided into small blocks that are distributed across MPI processes. The FLASH block is a three-dimensional array and there are 80 blocks
The block structure of FLASH mapped into memory and a checkpoint file. The FLASH block is a three-dimensional array with an additional 4 elements as guard cells in each direction and there are 80 blocks in each MPI process. Each inner block surrounded by guard cells (in the middle figure) has 24 variables. Unlike the Flash block mapping into memory (in the middle figure), each variable has 80 FLASH blocks, and all 24 variables are contiguously mapped into a checkpoint file (in the rightmost). Using collective I/O, every process writes the contiguous $8 \times 8 \times 8$ blocks into a checkpoint file for all 24 variables.

on each MPI process. Each block contains inner blocks with additional four guard cells to hold the state variables of the neighboring blocks for use in the hydrodynamic algorithm. The inner block surrounded by guard cells has 24 data array variables, e.g., density, velocities, energy, and pressure. Every process writes these blocks to a checkpoint file using 24 collective I/O, in a fashion that the checkpoint file appears as the data for variable 0 up to variable 23. Since the number of blocks is fixed to 80 for each process, increasing the number of MPI processes linearly increases the I/O amount. FLASH I/O produces one checkpoint file and two visualization files that contain centered and corner data. The I/O time to create the checkpoint file dominates the entire benchmark. Figure 4.9 presents the block structure of FLASH and describes how the block is mapped into memory and a checkpoint file. Unlike the Flash block mapping into memory, each variable has 80 FLASH blocks and all 24 variables are contiguously mapped into the
checkpoint file. Note that the size of contiguous memory region per process is only the size of double, e.g., 8 bytes. However, the contiguous regions in the file are $8 \times 8 \times 8$ (for the dimensions $x$, $y$, and $z$) $\times$ double, that is, 4096 bytes. Since the access pattern of FLASH I/O is noncontiguous in memory and in file, it is a challenging application for parallel I/O systems.

### 4.5.2 S3D I/O

The S3D I/O benchmark is the I/O kernel of a parallel turbulent combustion application, named S3D [29], developed at Sandia National Laboratories. Using direct numerical simulation (DNS), S3D solves the fully compressible reacting Navier-Stokes, total energy, species and mass continuity equations coupled with detailed chemistry. The equations are solved on a conventional structured Cartesian mesh, and scalable parallelism is achieved through MPI and a domain decomposition strategy. A checkpoint is carried out at regular intervals to store the values of three-dimensional Cartesian mesh points to the variables in 8-byte three-dimensional arrays. This checkpoint data can be used to obtain several more derived physical quantities of interest. Therefore, most of checkpoint data is maintained for later use. At each checkpoint, four global arrays—representing the variables of mass, velocity, pressure, and temperature—are written to the files using four collective write operations for four arrays.
Listing 2. A sample format of data definition hints. The dimension hint provides data space information. The variable hint defines the data type and the data space. The attribute hint adds additional information to interpret data. Read and write hints describe the portion of the data to be read and written, respectively.

```plaintext
#begin_dimensions
name: len;
...
#end_dimensions

#begin_variables
name: data_type, num_dim, dim1_name, dim2_name,
dim3_name, ...;
...
#end_variables

#begin_attributes
name: assoc_var_name, data_type, value;
...
#end_attributes

#begin_write
array: process_id, var, dim1_start, dim1_count,
dim2_start, dim2_count, ..., buffer;
...
#end_write

#begin_read
array: process_id, var, dim1_start, dim1_count,
dim2_start, dim2_count, ..., buffer;
...
#end_read
```
Fig. 4.10.  S3D I/O data partitioning and mapping patterns [37]. For three-dimensional array in (a), the subarray of each process is mapped to the global array in block partitioning of x-y-z dimensions. For four-dimensional array in (b), the lowest x-y-z dimensions are partitioned the same way as the three-dimensional arrays, but the fourth dimension is not partitioned.
Figure 4.10 shows the data partitioning patterns and mapping of S3D arrays. Among four arrays, pressure and temperature are three-dimensional arrays as shown in Figure 4.10(a), whereas mass and velocity are four-dimensional as depicted in Figure 4.10(b). All arrays of four variables share the same size of three-dimensions, $x$, $y$, and $z$, and are assigned to MPI processes along the $x$-$y$-$z$ dimensions. For three-dimensional arrays, the subarray of each process is mapped to the global array in block partitioning of $x$-$y$-$z$ dimensions. For four-dimensional arrays, the lowest $x$-$y$-$z$ dimensions are partitioned similar to the three-dimensional arrays, but the fourth dimension is not partitioned. For the arrays of mass and velocity, the length of fourth dimension is 11 and 3, respectively.

4.5.3 Experimental Results

To demonstrate the capabilities and effectiveness of our implementation, we first ran the default versions of our two benchmarks that include originally hand-optimized I/O code and measured the execution time of each. After that, we ran them again, but with tool-generated I/O code at this time. To generate I/O operations for the benchmarks, we used the template files written in C and Fortran that do not contain I/O operations: `ncmpi_parallel_write.c` for FLASH I/O and `pnetcdf.m.f90` for S3D I/O.

Figure 4.11 compares the average execution times with the hand-generated I/O code and the IOGenie-generated I/O code for FLASH I/O and S3D I/O. We measured the average execution time after 20 iterations. In each run, we dropped cache in the
servers and the compute nodes to minimize the effect on cache. The results shown in Figure 4.11 for the I0Genie-generated versions include all overheads of our approach.

In Figure 4.11(a), the average execution time of FLASH I/O increases with an increase in the number of MPI processes. Recall that as the number of MPI processes is scaled up, the total amount data is scaled up as well. Accordingly, the execution time increases along with the increase in the number of processes. The FLASH I/O benchmark performs poorly under independent I/O due to the number of I/O requests and its access patterns, without optimizations. It issues a huge number of I/O requests, 80 (blocks) \( \times \) 8 (x-elements) \( \times \) 8 (y-elements) \( \times \) 8 (z-elements) \( \times \) 24 (variables), that is 983,040 requests per MPI process. What is worse, the access pattern of the FLASH code is noncontiguous in memory and in file. During each collective write, every MPI process writes contiguous blocks of data, appended to the data written by the previous ranked MPI process. As a result, a write request from one process is not overlapped or interleaved with a request from another process. This non-interleaved access pattern triggers
the independent I/O subroutine, instead of collective I/O subroutines, even if MPI collective I/O writes are explicitly called in the FLASH I/O benchmark. This behavior is governed by the \texttt{romio\_cb\_write} hint, by controlling whether collective buffering is applied to collective write operations. In MPI-IO, the value of \texttt{romio\_cb\_write} is set to \textit{automatic} by default. That is, MPI-IO uses heuristics to determine when to enable the optimization. We noticed that when the hint was set to \textit{enable} to use collective writes, the I/O performance of FLASH I/O was worse than the case where it is set to the default value. However, the performance is significantly improved with data sieving, combining I/O requests through buffering, by reducing the number of I/O requests. Therefore, our tool generates collective I/O operations for FLASH I/O based on the user specification and lets MPI-IO determine the I/O subroutines to use.

In our evaluation of S3D I/O, we keep the size of partitioned x-y-z dimensions as $200 \times 200 \times 200$ in each process. Each run produces three checkpoint files, each with a size of 976.6MB. Unlike FLASH I/O, as shown in Figure 4.11(b), the average execution time of S3D I/O decreases with an increase in the number of MPI processes. For a collective write in S3D I/O, a subset of MPI tasks (called \textit{aggregator}) in each client node performs two-phase I/O. The aggregator communicates with other processes to exchange data and writes a large chunk of data into a temporary buffer. After that, the aggregator process in each node ships the I/O request to the I/O servers. In our configuration, each client node can host 64 MPI processes. In Figure 4.11(b), thus, we have the aggregator processes with the number of 1, 2, 4, and 8, to do actual I/O operations. As we increase the number of MPI processes, the aggregate I/O amount proportionally increases as well. Since a large amount of I/O data is performed by the aggregator process as we
increase the number of MPI processes, the execution time decreases as can be observed in Figure 4.11(b). Observing that the two-phase I/O strategy can significantly improve the parallel I/O performance in S3D I/O, our tool generates I/O operation code to exploit it for S3D I/O.

4.6 Conclusions

Poor I/O performance has been widely recognized as the bottleneck in HPC domain. Despite the significant effort and advance to improve I/O performance in hardware system architecture and software libraries, scientific analyses and discoveries are daunted and discouraged without efficient use them. In this work, we propose an automatic parallel I/O code generation framework, called IOGenie, to generate the efficient and optimized code for the scientific parallel applications. Using a graphical user interface, our tool accepts high-level I/O hints provided by users, analyzes the given annotations, and generates optimized I/O code that effectively exercises the underlying I/O stack. In our evaluation, using two scientific data-intensive applications, FLASH I/O and S3D I/O, written in C and Fortran, respectively, our tool generates optimized I/O operations, depending on data access patterns of the applications. For the FLASH I/O benchmark whose data access pattern is noncontiguous, our tool generates I/O code to be suitable for data sieving, while it generates I/O operations to be appropriate to two-phase I/O for the contiguous data access in the S3D I/O benchmark. Also, our experiments indicate that the overhead of tool-generated code is negligible.

The contribution of this work is two-fold. One is to improve the productivities of programmers by supporting code generation in different I/O libraries. The other is to
enhance the quality of tool-generated I/O code by applying optimizations, considering data access patterns from the applications as well as the optimizations in the underlying I/O libraries.
Chapter 5

Related Work

There exist prior research in profiling performance and diagnosing the related problems in large-scale distributed systems. In this section, we discuss the work related to static/dynamic instrumentation, and tracing, profiling frameworks. we also discuss the previous work related to code generation and optimizations at various layers of an I/O software stack.

5.0.1 Static/Dynamic Instrumentation

Over the past decade a lot of static/dynamic code instrumentation tools have been developed and tested that target different machines and application domains. Static instrumentation generally inserts probe code into the program at compile time. Dynamic instrumentation, on the other hand, intercepts the execution of an executable at different points of execution and inserts instrumentation code at runtime. ATOM [38] statically instruments the binary executable through rewriting at compile time. FIT [39] is an ATOM-like static instrumentation tool but aims at retargetability rather than instrumentation optimization. HP’s Dynamo [40] monitors an executable’s behavior through interpretation and dynamically selects “hot instruction traces” from the running program. DynamoRIO [41] is a binary package with an interface for both dynamic instrumentation and optimization. PIN [22] is designed to provide a functionality simulator to the ATOM toolkit; but, unlike ATOM which instruments an executable
statically by rewriting it, PIN inserts the instrumentation code dynamically while the binary executable is executing. Dyninst [42] and Paradyn [43] are designed for dynamic instrumentation to reduce the overheads incurred during instrumentation.

5.0.2 Tracing and debugging

Tools such as CHARISMA [44], Pablo [45], and Tuning and Analysis Utilities (TAU) [46] collect and analyze file system traces [47]. Paraver [48] is designed to analyze MPI, OpenMP, Java, hardware counter profiles, and operating system activity. OpenSpeedShop [49] is targeted to support performance analysis of applications. Kojak [50] aims at the development of a generic automatic performance analysis environment for parallel programs, and Stack Trace Analysis Tool (STAT) [51] is designed to help debug large-scale parallel programs.

5.0.3 Large-scale Distributed System Tracing

To understand complex system behavior, Magpie [52] automatically extracts a system’s workload during execution and produces a workload model. This work has been extended to datacenters [53]. Fay [54] provides dynamic tracing of distribute systems for user- and kernel-mode operations in x86-64 Windows systems. Lee et al. [55] proposed the dynamic probe class library API for large-scale systems, extended by DynInst. Darshan [56] captures I/O behavior such as I/O rates, transaction sizes, and I/O library usage in HPC applications. Vampir [57] provides an analysis framework for MPI applications, and IOPin [58] performs the runtime profiling of parallel I/O operations in HPC systems.
5.0.4 Code Generation

Automated code generation has been the subject of many prior works. Budinsky et al. describe the architecture and implementation of a tool that automates the implementation of design patterns [59]. Given application-specific information for a given pattern, the tool generates all the pattern-prescribed code automatically. Domain-specific modeling [60] mainly aims to raise the level of abstraction and generate final products in a chosen programming language or other forms. Using high-level abstraction beyond programming, it specifies the solution in a language that directly uses concepts and rules from a specific problem domain. Model-driven engineering [61] offers a promising approach to alleviate the complexity of platforms and express domain concepts effectively by combining domain-specific modeling languages with an automated code transformation process.

5.0.5 I/O Software Stack Optimizations

Extensive prior research has focused on improving I/O performance at various levels in the I/O software stack. Parallel file systems such as Lustre [14], GPFS [15], PanFS [16], and PVFS [17], coordinate accesses to files and provide an interface to access contiguous regions of files in high-performance. In comparison, MPI-IO [18], part of MPI-2 [19] specification, provides the standard I/O interface in HPC and a variety of optimizations such as data sieving[5], collective I/O [5], and two-phase I/O [6].

Zhang et al. propose resonant I/O technique to rearrange the I/O requests from multiple MPI processes with the striping patterns so that non-sequential access changes
into sequential access [62]. This new collective I/O strategy demonstrate significant performance improvements. Chen et al. propose the layout-aware collective I/O scheme [63] that recognizes the underlying physical data layout and rearranges accesses for locality and concurrency. Song et al. propose to coordinate I/O servers to serve one application at a time to reduce the completion time, and at the same time to maintain the server utilization and fairness [10]. At the application layer, Kandaswamy et al. investigate the impact of I/O optimization techniques considering the specifics of the applications [64].

5.0.6 Caching and Prefetching

Many prior I/O related studies have focused on caching and prefetching in HPC systems. Liao et al. propose application-aware client-side file caching [65] for MPI-IO and extend it in [66] to enhance parallel I/O performance. Vilayannur et al. propose discretionary caching for parallel I/O that employs both compilation techniques and runtime support [67]. Eshel et al. design a scalable, high-performance, cluster file system cache for data-intensive applications [68]. Data is cached and updated using pNFS that performs parallel I/O between clients and servers. Client-side caching is supported in several parallel file systems such as GPFS [15] and PanFS [16].

Patterson et al. present informed prefetching and caching to allocate buffers dynamically using access patterns for I/O-intensive applications [69]. Voelker et al. propose cooperative prefetching and caching using optional program-provided hints to with disk-latency reduction technologies [70]. Patrick et al. implement and test a hint passing mechanism from an application to the underlying I/O stack [71]. The I/O stack in turn fuses these user-specified hints to perform cross-layer I/O optimizations.
Chapter 6

Conclusions and Future Work

Poor I/O performance has been widely recognized as the bottleneck in HPC domain. Despite the significant effort and advance to improve parallel I/O performance in hardware system architecture and software libraries, scientific analyses and discoveries are daunted and discouraged without efficient use of them. To optimize parallel I/O, performance analysis and visualization is an important step towards developing an understanding and characterizing I/O behavior, which is essentially a result of complex interactions in the software libraries and hardware components. Performing manual code instrumentation is often difficult and extremely error-prone. Even building the I/O stack and configuring the running environment for application benchmarks is not trivial because of the scale of the current HPC systems. Moreover, collecting and analyzing trace data from them is a challenging task. To alleviate these difficulties and help write the efficient and optimized parallel I/O code for scientific parallel applications, this dissertation proposes three complimentary infrastructures, namely, IOPro, IOPin, and IOGenie.

IOPro provides the integrated profiling and analysis environment for the entire I/O stack. It uses existing MPI I/O function calls and therefore adds minimum overhead to the execution time of applications. By leveraging runtime dynamic instrumentation, IOPin also provides profiling functionality for parallel I/O with minimum overhead.
These two infrastructures (IOPro and IOPin) provide multiple metrics to analyze and investigate detailed I/O behavior, including latency, throughput, energy consumption, and call information. The results from these metrics contribute to evaluating and explaining the parallel I/O behavior across the I/O stack. In addition, our automatic parallel I/O code generation framework, IOGenie, helps generate the efficient and optimized parallel I/O code automatically for a large set of scientific parallel applications. Using a graphical user interface, our tool accepts high-level I/O hints provided by users, analyzes the given annotations, and generates optimized I/O code that effectively exercises the underlying I/O stack.

We used two scientific data-intensive applications, S3D I/O and FLASH I/O, to evaluate the use and effectiveness of IOPro, IOPin, and IOGenie. Our experiments with IOPro and IOPin clearly demonstrate different I/O behaviors in each application: S3D I/O exchanges data among the joined processes to do optimization and synchronization in the MPI library whereas FLASH I/O rarely does such optimization. Although both applications issue collective I/O requests to write the checkpoint files, the characteristics of I/O are quite different in each benchmark. By using the performance information depending on the I/O behavior, the application programs can be optimized to improve the storage system performance. Also, customized instrumentation can be performed to obtain more detailed performance statistics from the underlying I/O stack. This dissertation also shows that, when multiple applications interfere each other due to sharing I/O system resources, IOPro can be used to profile detailed performance metrics, aid in understanding complex I/O behavior, and detect the issue that degrades the
performance. Based on the gleaned information, the user can then employ an appropriate solution.

In our evaluation, IOGenie generates optimized parallel I/O operations in our two benchmarks, depending on data access patterns of the applications. For the FLASH I/O benchmark whose data access pattern is noncontiguous, our tool generates I/O code for data sieving, while it generates I/O operations to be appropriate to two-phase I/O for the contiguous data access in the S3D I/O benchmark. Also, our experiments indicate that the overhead of tool-generated code is negligible.

Overall, the contribution of this dissertation is two-fold. First, it provides an infrastructure to help understand the complex I/O behavior by instrumentation and suggest I/O solutions to improve the performance using IOPro and IOPin. Second, it helps improve the programmers’ productivity by automating I/O code generation and embedding cross layer I/O stack optimization.

In the future, we plan to test our infrastructure under very large MPI process counts and to further explore the potential of our tool to provide better optimizations in parallel I/O. We also plan to extend the functionality of IOPro to adopt customized user’s probe to better provide customized performance metrics. By integrating dynamic instrumentation into IOPro, the extended infrastructure can provide both static and runtime instrumentation. In addition, by implanting IOGenie into it, our total I/O solution package helps understand and characterize parallel I/O behavior using both static and dynamic approaches, diagnose I/O issue involved in software and hardware, and write an efficient parallel I/O code that maximizes high-performance storage systems, in one spot.
Bibliography


Curriculum Vitae

Brief Biography

I received a Ph.D. and a Master’s degree in Computer Science and Engineering at Pennsylvania State University (Advisor: Mahmut Kandemir). In addition to these academic activities, I have 2+ years industry experience.

Publications

IOGenie: An Automatic Parallel I/O Code Generation and Optimization Framework for HPC Applications, Seong Jo Kim, Jun Liu, and Mahmut Kandemir (In progress)

IOPro: A Parallel I/O Profiling and Visualization Framework for High-Performance Storage Systems Seong Jo Kim, Yuanrui Zhang, Seung Woo Son, Mahmut Kandemir, Wei-keng Liao, Rajeev Thakur, and Alok Choudhary, In Journal of Supercomputing (JSC) (Under revision)

IOPin: Runtime Profiling Parallel I/O in HPC Systems, Seong Jo Kim, Seung Woo Son, Mahmut Kandemir, Wei-keng Liao, Rajeev Thakur, and Alok Choudhary, In Proceedings of the 7nd international workshop on Petascale data storage: held in conjunction with Supercomputing’12 (PDSW12)

An Evolutionary Path to Object Storage Access, David Goodell, Seong Jo Kim, Robert Latham, Mahmut Kandemir, and Robert Ross, In Proceedings of the 7nd international workshop on Petascale data storage: held in conjunction with Supercomputing’12 (PDSW12)

Automated Tracing of I/O Stack, Seong Jo Kim, Yuanrui Zhang, Seung Woo Son, Ramya Prabhakar, Mahmut Kandemir, Christina Patrick, Wei-keng Liao, and Alok Choudhary, In Proceedings of 17th European MPI Users’ Group Meeting (EuroMPI 2010)