Automated Tracing of I/O Stack*

Seong Jo Kim[1], Yuanrui Zhang[1], Seung Woo Son[2], Ramya Prabhakar[1], Mahmut Kandemir[1], Christina Patrick[1], Wei-keng Liao[3], and Alok Choudhary[3]

[1] Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA, 16802, USA
[2] Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL, 60439, USA
[3] Department of Electrical Engineering and Computer Science
Northwestern University, Evanston, IL 60208, USA

Abstract. Efficient execution of parallel scientific applications requires high-performance storage systems designed to meet their I/O requirements. Most high-performance I/O intensive applications access multiple layers of the storage stack during their disk operations. A typical I/O request from these applications may include accesses to high-level libraries such as MPI I/O, executing on clustered parallel file systems like PVFS2, which are in turn supported by native file systems like Linux. In order to design and implement parallel applications that exercise this I/O stack, it is important to understand the flow of I/O calls through the entire storage system. Such understanding helps in identifying the potential performance and power bottlenecks in different layers of the storage hierarchy. To trace the execution of the I/O calls and to understand the complex interactions of multiple user-libraries and file systems, we propose an automatic code instrumentation technique, which enables us to collect detailed statistics of the I/O stack. Our proposed I/O tracing tool traces the flow of I/O calls across different layers of an I/O stack, and can be configured to work with different file systems and user-libraries. It also analyzes the collected information to generate output in terms of different user-specified metrics of interest.

Keywords: Automated code instrumentation, Parallel I/O, MPI-IO, MPICH2, PVFS2.

1 Introduction

Emerging data-intensive applications make significant demands on storage system performance and, therefore, face what can be termed as *I/O Wall*, that is, I/O behavior is the primary factor that determines application performance. Clearly, unless the I/O wall is properly addressed, scientists and engineers will not be able to exploit the full potential of emerging parallel machines when

* This work is supported in part by NSF grants 0937949, 0621402, 0724599, 0821527, 0833126, 0720749, 0621443, 0724599, and 0833131 and DOE grants DEAC02-06CH11357, DE-FG02-08ER25848, DE-SC0002156, and DESC0001283.

R. Keller et al. (Eds.): EuroMPI 2010, LNCS 6305, pp. 72–81, 2010.

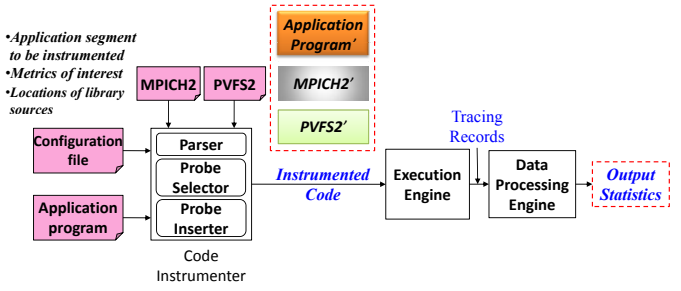

Fig. 1. Our automated I/O tracing tool takes as input the application program, I/O stack information and a configuration file which captures the metrics of interest, locations of target sources, and a description of the region of interest in the code. It automatically generates and runs instrumented code, and finally collects and analyzes all the statistics on the fly.

running large-scale parallel applications from bioinformatics, climate prediction, computational chemistry, and brain imaging domains.

The first step in addressing the I/O wall is to *understand* it. Unfortunately, this is not trivial as I/O behavior today is a result of complex interactions that take place among multiple software components, which can be referred to, collectively, as *I/O Stack*. For example, an I/O stack may contain an application program, a high-level library such as MPI-IO [8], a parallel file system such as PVFS [3], and a native file system such as Linux. A high-level I/O call in an application program flows through these layers in the I/O stack and, during this flow, it can be fragmented into multiple smaller calls (sub-calls) and the sub-calls originating from different high-level calls can contend for the same set of I/O resources such as storage caches, I/O network bandwidth, disk space, etc. Therefore, understanding the I/O wall means understanding the flow of I/O calls over the I/O stack.

To understand the behavior of an I/O stack, one option is to let the application programmer/user to instrument the I/O stack manually. Unfortunately, this approach (manual instrumentation) is very difficult in practice and extremely error prone. In fact, tracking even a single I/O call may necessitate modifications to numerous files and passing information between them.

Motivated by this observation, in this work, we explore automated instrumentation of the I/O stack. In this approach, as shown in Figure 1, instead of instrumenting the source code of applications and other components of the I/O stack manually, an application programmer specifies what portion of the application code is to be instrumented and what statistics are to be collected. The proposed tool takes this information as input along with the description of the target I/O stack and the source codes of the application program and other I/O stack software, and generates, as output, an instrumented version of the application code as well as instrumented versions of the other components

(software layers) in the I/O stack. All necessary instrumentation of the components in the I/O stack (application, libraries, file systems) are carried out automatically.

A unique aspect of our approach is that it can work with different I/O stacks and with different metrics of interest (e.g., I/O latency, I/O throughput, I/O power). Our experience with this tool is very encouraging so far. Specifically, using this tool, we automatically instrumented an I/O-intensive application and collected detailed performance and power statistics on the I/O stack.

Section 2 discusses the related work on code instrumentation and profiling. Section 3 explains the details of our proposed automated I/O tracing tool. An experimental evaluation of the tool is presented in Section 4. Finally, Section 5 concludes the paper with a summary of the planned future work.

2 Related Work

Over the past decade many code instrumentation tools that target different machines and applications have been developed and tested. ATOM [24] inserts probe code into the program at compile time. Dynamic code instrumentation [1,2,17], on the other hand, intercepts the execution of an executable at runtime to insert user-defined codes at different points of interest. HP's Dynamo [1] monitors an executable's behavior through interpretation and dynamically selects hot instruction traces from the running program.

Several techniques have been proposed in the literature to reduce instrumentation overheads. Dyninst and Paradyn use fast breakpoints to reduce the overheads incurred during instrumentation. They both are designed for dynamic instrumentation [12]. In comparison, FIT [5] is a static system that aims retargetability rather than instrumentation optimization. INS-OP [15] is also a dynamic instrumentation tool that applies transformations to reduce the overheads in the instrumentation code. In [27], Vijayakumar et al. propose an I/O tracing approach that combines aggressive trace compression. However, their strategy does not provide flexibility in terms of target metric specification. Tools such as CHARISMA [20], Pablo [23], and TAU (Tuning and Analysis Utilities) [19] are designed to collect and analyze file system traces [18]. For the MPI-based parallel applications, several tools, such as MPE (MPI Parallel Environment) [4] and mpiP [26], exist. mpiP is a lightweight profiling tool for identifying communication operations that do not scale well in the MPI-based applications. It reduces the amount of profile data and overheads by collecting only statistical information on MPI functions. Typically, the trace data generated by these profiling tools are visualized using tools such as Jumpshot [28], Nupshot [14], Upshot [11], and PerfExplorer [13].

Our work is different from these prior efforts as we use source code analysis to instrument the I/O stack automatically. Also, unlike some of the existing profiling and instrumentation tools, our approach is not specific to MPI or to a pre-determined metric; instead, it can target an entire I/O stack and work with different performance and power related metrics.

3 Our Approach

3.1 High-Level View of Automated Instrumentation

Our goal is to provide an automated I/O tracing functionality for parallel applications that exercise multiple layers of an I/O stack, with minimal impact to the performance. To this end, we have implemented in this work an automated I/O tracing tool that, as illustrated in Figure 1, comprises three major components: code instrumenter, execution engine, and data processing engine.

As shown in Figure 1, the code instrumenter consists of the parser, the probe selector, and the probe inserter. In this context, a *probe* is a piece of code being inserted into the application code and I/O stack software (e.g., in the source codes of MPI-I/O and PVFS2), which helps us collect the required statistics. The code instrumenter takes as input the application program, high level I/O metrics of interest written in our specification language, and the target I/O stack (which consists of the MPI library and PVFS2 in our current testbed).

The parser parses I/O metrics of interest from the configuration file, extracts all necessary information to instrument the I/O stack in a hierarchial fashion from top to bottom, and stores it to be used later by other components of the tool. After that, the probe selector chooses the most appropriate probes for the high-level metrics specified by the user. Finally, the probe inserter automatically inserts the necessary probes into the proper places in the I/O stack. Note that, depending on the target I/O metrics of interest, our tool may insert multiple probes in the code. Table 1 lists a representative set of high-level metrics that can be traced using our tool.

Table 1. Sample high-level metrics that can be traced and collected using the tool

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

The execution engine compiles and runs the *instrumented* I/O stack, and generates the required trace. Finally, the data processing engine analyzes the trace log files and returns statistics based on the user's queries. The collected statistics can be viewed from different perspectives. For example, the user can look at the I/O latency/power breakdown at each server or at each client. The amount of time spent by an I/O call at each layer of the target I/O stack can also be visualized.

3.2 Technical Details of Automated Instrumentation

In this section, we discuss details of the code instrumenter component of our tool. Let us assume, for the purpose of illustration, that the user is interested

```
-A [application.c, application]
-L [$MPICH2, $PVFS2, ClientMachineInfo, $Log]
-O [w]
-C [100-300]
-S [4, <3 max>, <3 max>, <3 max>, <3 max>, <3 max>]
-T [4, <3, mpi.0.log , mpi.1.log, mpi.2.log>, <3, client.0.log, client.1.log, client.2.log>,
   <3, server.0.log, server.1.log,server.2.log>, <3, disk.0.log, disk.1.log,disk.2.log>]
-Q [latency, inclusive, all, list:, list:*, list:*, list:*]
-P [App;common;App-probe1;-l main;before]
-P [App;common;App-probe2;-l MPI_Comm_rank;after]
-P [App;common;App-Start-probe;-l MPI_File_read;before]
-P [App;common;App-Start-probe4;-l MPI_File_write;before]
-P [MPI;latency;MPI-Start-probe;-n 74;$MPICH2/mpi/romio/mpi-io/read.c]
-P [MPI;latency;MPI-End-probe;-n 165;$MPICH2/mpi/romio/mpi-io/read.c]
-P [MPI;latency;MPI-Start-probe;-n 76;$MPICH2/mpi/romio/mpi-io/read_all.c]
-P [MPI;latency;MPI-End-probe;-n 118;$MPICH2/mpi/romio/mpi-io/read_all.c]
-P [MPI;latency;MPI-End-probe;-n 73;$MPICH2/mpi/romio/mpi-io/write.c]
-P [MPI;latency;MPI-End-probe;-n 168;$MPICH2/mpi/romio/mpi-io/write.c]
-P [MPI;latency;MPI-Start-probe;-n 75;$MPICH2/mpi/romio/mpi-io/write_all.c]
-P [MPI;latency;MPI-End-probe;-n 117;$MPICH2/mpi/romio/mpi-io/write_all.c]
-P [MPI;latency;MPI-probe;-n 62;$MPICH2/mpi/romio/mpi-io/adio/ad_pvfs2_read.c]
-P [MPI;latency;MPI-probe;-n 295;$MPICH2/mpi/romio/mpi-io/adio/ad_pvfs2_write.c]
-P [PVFSClient;latency;Client-Start-probe;-n 372;$PVFS2/client/sysint/sys-io.sm]
-P [PVFSClient;latency;Client-End-probe;-n 397;$PVFS2/client/sysint/sys-io.sm]
-P [PVFSClient;latency;Client-probe;-n 670;$PVFS2/client/sysint/sys-io.sm]
-P [PVFSServer;latency;.Server-Start-probe;-n 153;$PVFS2/server/io.sm]
-P [PVFSServer;latency;.Server-End-probe;-n 5448;$PVFS2/io/job/job.c]
-P [PVFSServer;latency;.Disk-start-probe;-n 1342;$PVFS2/io/flow/flowproto-bmi-trove/flowproto
    -multiqueue.c]
-P [PVFSServer;latency;.Disk-end-probe1;-n 1513;$PVFS2/io/flow/flowproto-bmi-trove/flowproto-
    multiqueue.c]
-P [PVFSServer;latency;.Disk-end-probe2;-n 1513;$PVFS2/io/flow/flowproto-bmi-trove/flowproto-
    multiqueue.c]
```

Fig. 2. An example configuration file

in collecting statistics about the execution *latency* of each I/O call in each layer of the I/O stack, that is, the amount of time spent by an I/O call in MPI-I/O, PVFS2 client, PVFS2 server, and disk layers. A sample configuration file that captures this request is given in Figure 2. This file is written in our specification language, and Table 2 describes the details of each parameter in the configuration file. Let us now explain the contents of this sample configuration file.

Table 2. Flags used in a configuration file

Parameter	Description
-A	Application file name or path
-L	Path for I/O libraries
-O	Operation of interest
-C	Code segment of interest to trace
-S	I/O stack specification
-T	Tracing file location generated by our tool
-Q	Metric of interest
-P	Probe name and inserting location

In this example, the user wants to collect the execution *latency* of MPI-IO write operations (indicated using -O[w]) that occur between lines 100 to 300 of an application program called, *application.c*. Also, the user specifies three I/O stack layers, which are MPI-IO, PVFS2 client, and PVFS2 server (below the application program). Finally, the user describes the trace log file names and their locations for the data processing engine. Based on the target metric of interest, that is *latency*, the most appropriate latency probes can be automatically inserted into the designated places in the probe specification.

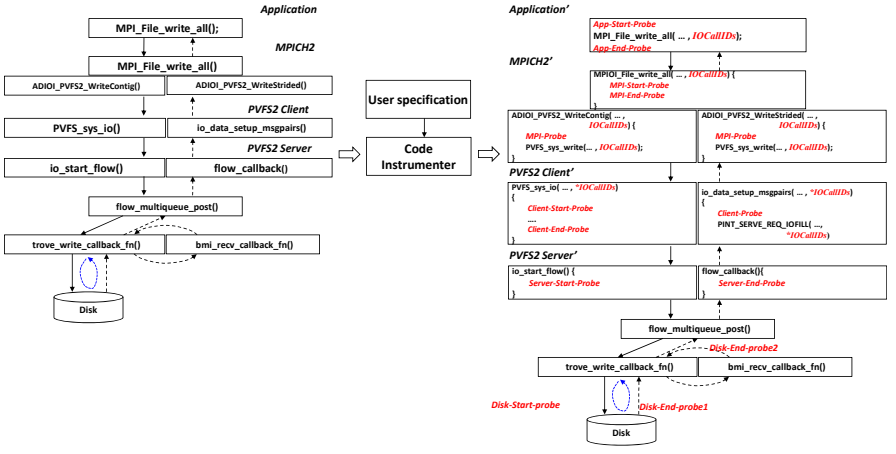

Fig. 3. Illustration of inserting probes into the application program and I/O stack components by our code instrumenter. Application′, MPICH2′, PVFS2 Client′, and PVFS2 Server′ represent the instrumented I/O stack.

Figure 3 illustrates how the code instrumenter works. It takes as an input the user configuration file along with MPI-IO and PVFS2. The parser parses this configuration file and extracts all the information required by the other components such as the probe inserter and the data processing engine. Based on the specified target metric, i.e., the execution *latency* of MPI *write* operations in all the layers including MPI-IO, PVFS2 client, PVFS2 server, and disk, the probe selector employs only the write-related latency probes, which helps to minimize the overheads associated with the instrumentation. Then, following the call sequence of MPI write function, from the MPI-IO library though the PVFS2 client to the PVFS2 server in Figure 3, the probe inserter selectively inserts the necessary probes into the start point and the end point of each layer described in the configuration file.

After the instrumentation, the probe inserter compiles the instrumented code. During the compilation, it also patches a small array structure, called *IOCallID*, to the MPI-IO and PVFS2 functions to be matched for tracing. IOCallIDs contain information about each layer such as the layer ID and the I/O type. When IOCallIDs are passed from the MPI-IO layer to the PVFS2 client layer, the inserted probe extracts the information from them and generates the log files with the latency statistics at the boundary of each layer.

Note that a high-level MPI-IO call can be fragmented into multiple small sub-calls. 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 layer boundaries in the I/O stack is not trivial. In our implementation, each call has a unique ID in the current layer and passes it to the layer below. This help us to connect the high-level call to its sub-calls in a hierarchical fashion. It also helps the data processing engine (see Figure 1) to combine the statistics coming from different layers in a

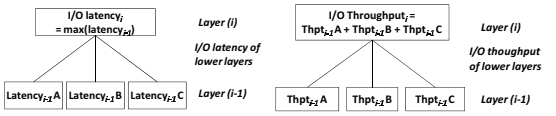

Fig. 4. Computation of I/O latency and I/O throughput metrics

systematic way (for example, all the variables that hold latency information at different layers are associated with each other using these IDs).

In the PVFS2 server layer, our tool uses a unique structure, called *flow_desciptor*, to perform the requested I/O operations from the PVFS2 client. The probe inserted into the start point in the server layer extracts the information in the IOCallIDs passed from the PVFS2 client and packs it into the flow_descriptor. Since the flow_descriptor is passed to the entire PVFS2 server, the probe in the server extracts the necessary information from it to collect the latency related statistics without much complexity.

The execution engine runs the instrumented code and generates the trace log files in each layer. Finally, the data processing engine analyzes all the trace log files and collects the execution I/O latency induced by each MPI operation in each layer. The I/O latency value computed at each layer is equal to the maximum value of the I/O latencies obtained from different layers below it. However, the computation of I/O throughput value is additive, i.e., the I/O throughput computed at any layer is the sum of I/O throughputs from different sub-layers below it. Figure 4 illustrates the computation of these metrics. To compute the I/O power, we use the power model described in [9].

4 Evaluation

To demonstrate the operation of our tracing tool, we ran a benchmark program using three PVFS2 servers and three PVFS2 clients on a Linux cluster that consists of 6 dual-core processor nodes, AMD Athlon MP2000+, connected through Ethernet and Myrinet. Each node of this system runs a copy of PVFS2 and MPICH2. To measure disk power consumption per I/O call, we used the disk energy model [9] based on the data sheets of the IBM Ultrastar 36Z15 disk [25]. Table 3 gives the important metrics used to calculate power consumption.

Table 3. Important disk parameters for power calculation

Parameter	Default Value
Disk drive module	IBM36Z15
Storage capacity (GB)	36.7
Maximum disk speed (RPM)	15000
Active power consumption (Watt)	13.5
Idle power consumption (Watt)	10.2

In our evaluation, we used the FLASH I/O benchmark [7] that simulates the I/O pattern of FLASH [29]. It creates the primary data structures in the FLASH code and generates three files: a checkpoint file, a plot file for center data, and a plot file for corner data, using two high-level I/O libraries: PnetCDF [16] and HDF5 [10]. The in-memory data structures are 3D sub-blocks of size 8x8x8 or 16x16x16. In the simulation, 80 of these blocks are held by each processor and are written to

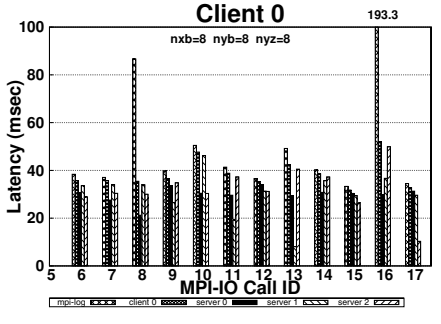

Fig. 5. Latency of Client 0 using PnetCDF

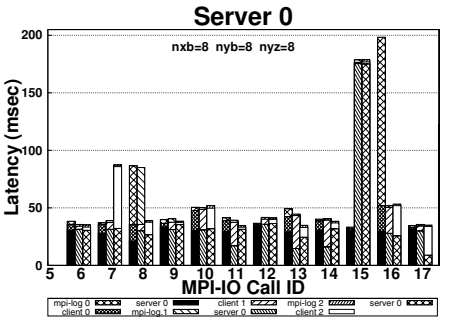

Fig. 6. Latency of Server 0 using PnetCDF

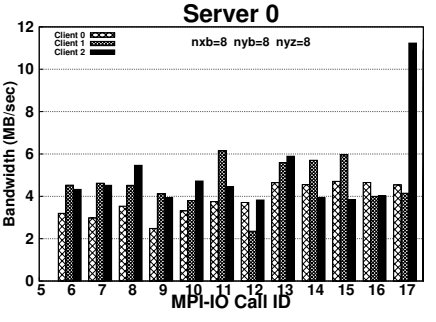

Fig. 7. Disk throughput of Server 0 using PnetCDF

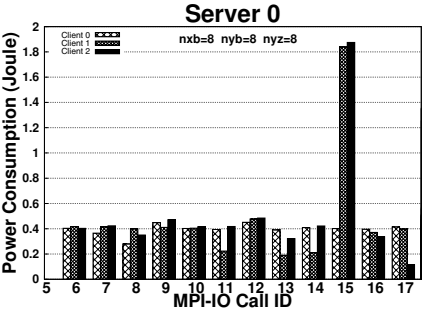

Fig. 8. Power consumption of Server 0 using PnetCDF

three files with 50 MPI_File_write_all function calls. We used the sub-blocks of size 8x8x8. nxb, nyb, and nyz in Figures 5 through 9 represent these sub-block sizes.

First, Figure 5 shows the I/O latencies experienced by each MPI-IO call from Client 0's perspective when PnetCDF is used. MPI-IO calls 6-17 are shown with latencies (in milliseconds) taken in mpi-log, Client 0, Server 0, Server 1, and Server 2, from left to right. We see that MPI-IO call 16 takes 193.3 milliseconds in mpi-log, but only 32 milliseconds in Server 1. In this experiment, some of the MPI-IO calls (0-5, 18-45) calls are directed to the metadata server to write header information of each file. These calls were not recorded in the I/O server log file. Figure 6, on the other hand, plots the latencies observed from Server 0's perspective. The three bars for every call ID represent cumulative latencies from each client (Client 0, Client 1 and Client 2 from left to right). Further, each bar also gives a breakdown of I/O latency (in milliseconds) taken for the request to be processed in the MPI-IO, PVFS client, and PVFS server layers, respectively. From this result, one can see, for example, that Client 1 and Client 2 spend less time than Client 0 in the Server 0 as far as call ID 16 is concerned. These two plots in Figure 5 and Figure 6 clearly demonstrate that our tool can be used to study the I/O latency breakdown, from both clients' and server's perspectives.

Figure 7 illustrates the I/O throughput in Server 0. One can observe from this plot the detailed I/O throughput patterns of different clients regarding this server. Comparing Figure 6 with Figure 7, one can also see that the bottleneck I/O call in the application code depends on whether I/O latency or I/O throughput is targeted. Figure 8, on the other hand, presents the power consumption results for Server 0. We see that most of the power is consumed by I/O call 15, and except for this call, power consumptions of Client 0 and Client 1 are similar on this server.

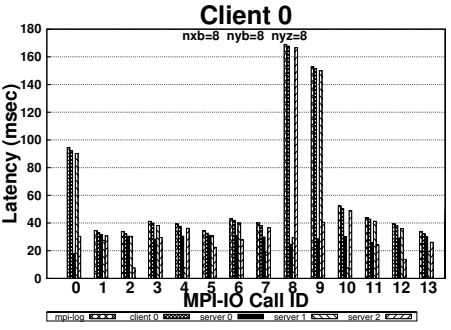

Fig. 9. Latency of Client 0 using HDF5

Our final set of results are given in Figure 9 and depict the I/O latency values observed from Client 0's viewpoint when using HDF5, instead of PnetCDF. Overall, these sample set of results clearly show that our tool can be used to collect and analyze detailed latency, throughput, and power statistics regarding the I/O stack.

5 Concluding Remarks and Future Work

Performing code instrumentation manually is often difficult and could be error-prone. Hence, we propose an automatic instrumentation technique that can be used to trace and analyze scientific applications using high level I/O libraries like PnetCDF, HDF5, or MPI I/O over file systems like PVFS2 and Linux. The tracing utility uses existing MPI I/O function calls and therefore adds minimum overhead to the application execution. It takes target high level metrics like I/O latency, I/O throughput and I/O power as well as a description of the target I/O stack as input and analyzes the collected information to generate output in terms of different user-specified metrics. As our future work, we plan to extend our analysis to other available I/O benchmarks, such as S3D-IO [22] and GCRM [21], to characterize their I/O behavior. We also plan to investigate techniques for dynamic code instrumentation that makes use of information available at runtime to generate/restructure code for data optimization.

References

1. Bala, V., et al.: Dynamo: A transparent dynamic optimization system. In: PLDI (2000)
2. Bruening, D.L.: Efficient, transparent, and comprehensive runtime code manipulation. PhD thesis, MIT, Cambridge, MA, USA (2004)
3. Carns, P.H., et al.: PVFS: A parallel file system for linux clusters. In: Proceedings of the Annual Linux Showcase and Conference (2000)
4. Chan, A., et al.: User's guide for MPE: Extensions for MPI programs (1998)

5. De Bus, B., et al.: The design and implementation of FIT: A flexible instrumentation toolkit. In: Proceedings of PASTE (2004)
6. del Rosario, J.M., et al.: Improved parallel I/O via a two-phase run-time access strategy. SIGARCH Comput. Archit. News 21(5), 31–38 (1993)
7. Fisher, R.T., et al.: Terascale turbulence computation using the flash3 application framework on the IBM Blue Gene/L system. IBM J. Res. Dev. 52(1/2) (2008)
8. Gropp, W., et al.: MPI — The Complete Reference: the MPI-2 Extensions, vol. 2. MIT Press, Cambridge (1998)
9. Gurumurthi, S., et al.: DRPM: Dynamic speed control for power management in server class disks. In: ISCA (2003)
10. HDF (Hierarchical Data Format) , `http://www.hdfgroup.org`
11. Herrarte, V., Lusk, E.: Studying parallel program behavior with upshot. Technical Report ANL–91/15, Argonne National Laboratory (1991)
12. Hollingsworth, J.K., et al.: MDL: A language and compiler for dynamic program instrumentation. In: Malyshkin, V.E. (ed.) PaCT 1997. LNCS, vol. 1277, Springer, Heidelberg (1997)
13. Huck, K.A., Malony, A.D.: PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In: SC (2005)
14. Karrels, E., Lusk, E.: Performance analysis of MPI programs. In: Workshop on Environments and Tools For Parallel Scientific Computing (1994)
15. Kumar, N., et al.: Low overhead program monitoring and profiling. In: Proceedings of PASTE (2005)
16. Li, J., et al.: Parallel netCDF: A high-performance scientific I/O interface. In: SC (2003)
17. Luk, C.-K., et al.: Pin: Building customized program analysis tools with dynamic instrumentation. In: PLDI (2005)
18. Moore, S., et al.: Review of performance analysis tools for MPI parallel programs. In: Cotronis, Y., Dongarra, J. (eds.) PVM/MPI 2001. LNCS, vol. 2131, p. 241. Springer, Heidelberg (2001)
19. Moore, S., et al.: A scalable approach to MPI application performance analysis. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 309–316. Springer, Heidelberg (2005)
20. Nieuwejaar, N., et al.: File-access characteristics of parallel scientific workloads. IEEE Transactions on Parallel and Distributed Systems 7, 1075–1089 (1996)
21. Randall, D.A.: Design and testing of a global cloud-resolving model (2009)
22. Sankaran, R., et al.: Direct numerical simulations of turbulent lean premixed combustion. Journal of Physics: Conference Series 46(1), 38 (2006)
23. Simitci, H.: Pablo MPI Instrumentation User's Guide. University of Illinois. Tech. Report (1996)
24. Srivastava, A., Eustace, A.: ATOM: A system for building customized program analysis tools. In: PLDI (1994)
25. Ultrastar, I.: 36Z15 Data Sheet (2010), `http://www.hitachigst.com/hdd/ultra/ul36z15.htm`
26. Vetter, J., Chambreau, C.: mpiP: Lightweight, scalable MPI profiling (2010), `http://mpip.sourceforge.net/`
27. Vijayakumar, K., et al.: Scalable I/O tracing and analysis. In: Supercomputing PDSW (2009)
28. Zaki, O., et al.: Toward scalable performance visualization with jumpshot. Int. J. High Perform. Comput. Appl. 13(3), 277–288 (1999)
29. Fryxell, B., et al.: FLASH: Adaptive Mesh Hydrodynamics Code. The Astrophysical Journal Supplement Series 131 (2000)