

# Characterizing the I/O Behavior of Scientific Applications on the Cray XT

Philip C. Roth

Computer Science and Mathematics Division  
Oak Ridge National Laboratory  
Oak Ridge, TN 37831  
rothpc@ornl.gov

## ABSTRACT

Scientific applications use input/output (I/O) for obtaining initial conditions and execution parameters, as a persistent way of saving program output, and for safeguarding against system unreliability. Although system sizes are expected to continue increasing, I/O performance is not expected to keep pace with system computation and communication performance. Understanding application I/O demands and system I/O capabilities is the first step toward bridging this gap between them. In this paper, we present our approach for characterizing the I/O demands of applications on the Cray XT. We also present preliminary case studies showing the use of our I/O characterization infrastructure with climate studies and combustion simulation programs.

## Categories and Subject Descriptors

C.4 [Performance of Systems]: *performance attributes, measurement techniques.*

## General Terms

Performance, Measurement.

## Keywords

Performance data collection, instrumentation, Cray XT.

## 1. INTRODUCTION

Scientific applications from areas like climate studies, fusion, and molecular dynamics use input/output (I/O) for several purposes, such as to obtain initial conditions and execution parameters, as a persistent way of saving program output, and to safeguard against system unreliability. This last purpose is becoming increasingly important: the desire to reach ever-increasing computational targets with high-performance computing (HPC) systems has produced a trend toward systems with an increasing number of components, and system reliability

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

© 2007 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.  
Supercomputing'07, Nov. 10-16, 2007, Reno, NV.  
Copyright 2007 ACM ISBN 978-1-59593-899-2/07/11...\$5.00

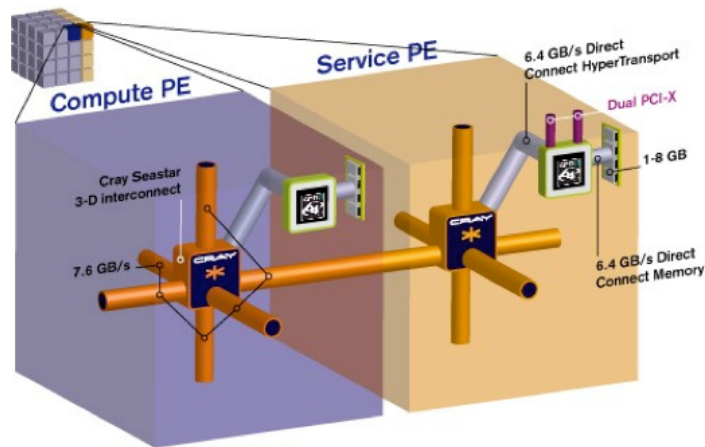
is expected to decrease as the number of components increases. Current HPC systems have several tens to hundreds of thousands of processor cores, but researchers are already bracing themselves for systems with millions of cores. Even with expected technological advances, this trend is expected to continue many years into the future. Because I/O in the form of checkpointing is the technique most often used to guard against system failures, and because I/O technology is not expected to keep pace with processor technology, I/O is an area of increasing concern for both producers and consumers of HPC systems.

Understanding application I/O demands and system I/O capabilities is the first step toward bridging the gap between the two. In response to the need for tools that provide insight into this gap, performance analysis tools like Paradyn [10] and the Tuning and Analysis Utilities (TAU) [15] support measurement and problem diagnosis of I/O performance. To support our investigation into the I/O behavior of scientific applications on leadership class systems, we have designed an I/O event tracing system and produced a prototype implementation for applications running on the Cray XT, the primary platform of the U.S. Department of Energy (DOE) Office of Science's Leadership Computing Facility at Oak Ridge National Laboratory (ORNL).

There are two primary contributions of this paper. First, we present our preliminary I/O characterization approach and its prototype implementation. Second, we present preliminary case studies showing the use of our I/O characterization approach with the Parallel Ocean Program (POP) [7] and the S3D combustion simulation program [4].

## 2. THE CRAY XT

The Cray XT is a parallel computing platform that features massive parallelism and high performance [1]. An XT system consists of processing elements (PEs) connected in a three-dimensional mesh or torus topology. Each PE contains an AMD Opteron processor, memory, and a Cray proprietary router Application-Specific Integrated Circuit (ASIC) called SeaStar (see Figure 1). Single- and multi-core processors are supported. The initial Cray XT systems (the XT3 and XT4) use only Opteron-based PEs, but the Cray XT5 also supports heterogeneous systems containing vector processors and Field Programmable Gate Arrays [5].



**Figure 1: Cray XT4 Processing Elements (Image courtesy Cray, Inc.)**

Cray XT PEs are partitioned into compute PEs and service PEs. Compute PEs run application processes, and use either a lightweight operating system kernel called Catamount [8] or Cray’s Compute Node Linux (CNL). Service PEs provide login and I/O services with a traditional Linux installation.

For this work, we used the Cray XT system from the DOE Leadership Computing Facility at ORNL. During the time of our experimentation, this system contained a combination of XT3 and XT4 cabinets. Also, the system has been converted from using the Catamount kernel on its compute nodes to CNL. At the time of our experimentation, this system used the Catamount kernel on its compute PEs and Lustre as its parallel file system.

### 3. THE IOT EVENT TRACING INFRASTRUCTURE

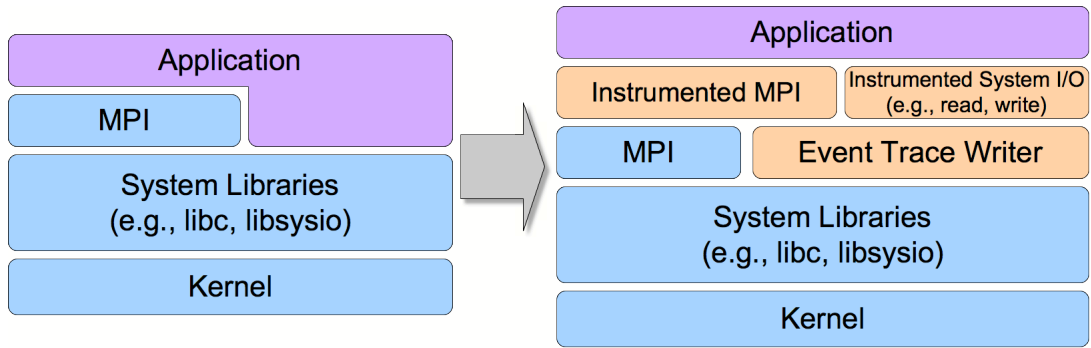
Event tracing is a well-established technique for performance data collection, and tools like TAU [15], Paraver [13], SCALASCA [2], and svPablo [6] have long supported collection and analysis of program event data, often including I/O events. For our application I/O characterization activity, we adopt a traditional event-based performance data collection approach that produces event trace files for ease of repeated post-mortem analysis and sharing with other researchers. We developed a prototype implementation of our event tracing infrastructure for MPI applications on the Cray XT; we call this prototype IOT. We intend IOT to be the first component of a more comprehensive performance data collection and analysis infrastructure for programs running on DOE leadership-class computing platforms.

Our data collection approach uses two components. The first component is a collection of functions that replace I/O and other interesting function calls (e.g., `open()` and `write()`) with an instrumented wrapper function. Each wrapper generates an event trace record for function entry, calls the real function that implements the desired functionality, generates an event trace record that captures the relevant details of the I/O operation, and then generates an event trace record for the function exit. At a

minimum, each event trace record includes a timestamp of when the operation occurred and the type of operation. Figure 2 shows where instrumented functions are interposed between an application process and the default runtime software stack. The second component is an event tracing support library that implements the needed functionality to produce event trace files in the Open Trace Format [12], a file format for expressing event traces that is supported by performance tools such as TAU, SCALASCA [2], and Vampir [11].

On many traditional UNIX and UNIX-like systems (e.g., Linux), we could use shared libraries to interpose our instrumented file I/O wrapper functions into the control flow between an application function that calls a file I/O function and the system’s implementation of that I/O function. In fact, Cray’s CNL supports shared libraries in order to ease the use of scripting languages such as Python in scientific applications. However, the Cray XT running Catamount does not support shared libraries and the default linking mode for XT systems running CNL is to produce statically linked executable files. Thus, we chose to use link-time function wrapping, facilitated by the GNU linker’s strong support for function wrapping. Using the `--wrap` command-line switch, this linker causes application calls to a function like `read()` to be calls to a function named `__wrap_read()` instead and exposes the original function with the name `__real_read()` instead. Our instrumented version of `__wrap_read()` uses the symbol `__real_read()` to access the system’s implementation of the `read()` function.

In addition to collecting event data for system file I/O functions, our event tracing software can collect event data for MPI [9] functions, including MPI-IO functions. Because a compliant MPI implementation includes support for the PMPI profiling interface, we use the PMPI interface for interposing instrumented MPI functions rather than the linker’s function wrapping facility to interpose our instrumented functions. For interposing our instrumented MPI wrapper functions, we use an automated wrapper generator script based on the generator used by the `mpiP` [16] lightweight MPI profiling tool.



**Figure 2: IOT Interposition of instrumented functions between an application process and the runtime software stack**

To simplify the use of our event trace capture software, we use custom versions of the Cray Fortran, C, and C++ compiler scripts that automatically include the correct linker switches and libraries to interpose our infrastructure libraries. For basic event tracing scenarios, the user need not modify their application source code to use our infrastructure; instead, the user modifies his makefiles to use the command `iot_ftn` instead of `ftn` to link their Fortran program.

Performance data collection using event tracing has the potential to generate massive volumes of performance data, e.g., if the events being traced occur frequently, are traced in a large number of processes, or generate a large amount of performance data each time they occur. Several techniques exist to manage the performance data volume produced by detailed event tracing. Dynamic control of the event tracing infrastructure can be used to enable and disable event tracing while a program runs. Such control may be explicitly specified using API functions provided by the tracing infrastructure, or implicitly enabled by the tracing infrastructure in response to excessive performance data volume. Sophisticated performance data collection tools like Paraver use pattern recognition to identify repetitive behavior and only keep event data for a limited, representative sequence of program events. Our IOT event tracing infrastructure currently uses simple event tracing control using explicit API functions coupled with OTF's compressed short format to manage performance data volume.

## 4. CASE STUDIES

As preliminary test cases for the prototype implementation of our I/O characterization approach, we have studied the I/O behavior of two scientific applications on the Cray XT at ORNL. At the time of our experimentation the ORNL Cray XT used the Catamount lightweight kernel on its compute nodes.

### 4.1 STATISTICS: THE PARALLEL OCEAN PROGRAM

The Parallel Ocean Program [7] (POP) is an ocean simulation program produced by researchers at Los Alamos National Laboratory. It serves as the ocean model in the Community

Climate System Model [3] (CCSM). It is implemented in Fortran 90 and uses MPI message passing for communicating data between parallel processes. It can use either netCDF or Fortran I/O functions for its output. The program performs I/O for reasons common to many scientific applications:

- To obtain simulation control parameters and initial conditions, such as topography grid data and forcing data used when POP is run in standalone mode (i.e., outside of the full CCSM);
- To save time-varying results such as movie frames and calculation history; and
- To save periodic checkpoint files.

In our experimentation, we used POP version 1.4.3 and the X1 benchmark problem with a grid spacing of one degree. To limit the time required for our program runs, we limited the number of simulation timesteps to forty timesteps; production runs involve many more timesteps. We also configured the program to output checkpoints every 10 timesteps, movie files every five timesteps, and no calculation history files.

POP implements its own collective I/O instead of using existing parallel I/O software like MPI-IO or parallel netCDF. Because performing I/O from too many processes can overwhelm the I/O capabilities of many systems, POP can be configured to limit the number of writer processes. For this study, we configured POP to use four output tasks. After we completed our study, climate community experts notified us that the parallel I/O feature of the POP version we used was suspected to be defective and that only one output task was usually used for production runs.

For our experiments, POP's I/O data volume was modest. The input activity consisted of reading approximately 7MB of horizontal and vertical grid data and approximately 490KB of topography data during program initialization. Output activity involved writing checkpoint files consisting of a 10KB text metadata file and a 346MB binary data file, and writing 3.9MB movie files using netCDF.

**Table 1: POP OTF trace file characteristics (all values in bytes)**

	Long			Short			Short-compressed			
	def	Fixed	Per-step	def	Fixed	Per-step	def	Fixed	Per-step	
Rank 0		314	41877	8479.4	283	23873	4833.15	158	6101	720.25
Rank !=0		314	424	1287.4	283	256	737.3	158	2266	66.6

POP output activity varied between the MPI rank 0 process and other writer processes. The rank 0 process alone writes the checkpoint metadata file; our IOT traces showed this process used seven write function calls to write this 10KB file. The rank 0 process also writes the entire netCDF movie file itself. The IOT event traces showed two write function calls each time a movie file was saved. All writer processes regardless of rank made write function calls to contribute to the checkpoint file. Each writer performed eighty write operations, each of approximately 980KB.

Our analysis of the IOT event trace files suggests several potential optimizations to improve POP I/O performance. First, the program could be modified to use a parallel I/O library for writing movie files to avoid serializing this activity. Second, the program’s checkpoint output activity could be adapted to use a parallel I/O library instead of its own collective communication and Fortran I/O operations. We stress, however, that any such changes must take into account any differences between the layout of data in memory versus the desired layout on disk, intended to support post-mortem analysis of the program’s results.

Table 1 describes the OTF event trace files produced when collecting event trace files describing the I/O activity of our four POP writer processes. The table shows the total performance data volume and the per-timestep data volumes for both MPI rank 0 and non-rank-0 processes. These data volumes reflect the data volume of all OTF local (per-process) metadata files but not the global metadata file. The event trace files include events for Fortran I/O operations but not the MPI communication performed by POP to gather program data to the writer processes. I/O operation event data in the generated trace file includes the operation timestamp, duration, and number of bytes read or written but not the number of bytes requested to be read or written. As expected, the OTF compressed short format is the most desirable output format. That this output format produced only 67 bytes per timestep is encouraging.

#### 4.2 VISUALIZATION: S3D

For another preliminary I/O characterization case study we used the combustion simulation program S3D [4]. Because our initial goal was to test the functionality of our approach, we applied our software to a small test case with eight application processes running for fifty simulation timesteps. Production S3D runs use thousands of processes and run for hundreds or thousands of timesteps.

In contrast to the POP case study where we analyzed IOT event trace files to obtain I/O operation statistics, for our S3D study we focused on event data visualization. A portion of the event trace corresponding to the writing of one checkpoint is shown in

the Vampir timeline display shown in Figure 3. In the figure, MPI events are shown in the timeline display, the lines indicating communication between processes are not shown for clarity.

Although our analysis of S3D’s I/O behavior is in its early stages, the Vampir timeline display reveals the general checkpoint I/O strategy used by the version of S3D we obtained. The MPI rank 0 process opens and reads data from a control file, then broadcasts a message describing the parameters to use for the checkpoint operation. The other MPI tasks, waiting for the broadcast, open a file for their checkpoint data (an event not clearly visible in the timeline visualization due to the display’s zoom factor). Once each process opens its checkpoint file, it writes its checkpoint data in several small write operations, at least as far as the system is concerned. The checkpoint finishes with a barrier operation, but a slow writer process causes most processes a delay before proceeding with the computation.

For our S3D test problem, these checkpoint files are each relatively small: only 16MB. However, because each process writes its own checkpoint file, the timeline visualization hints that runs of the version of S3D we used would present the metadata server with many nearly-simultaneous file create operations. This activity could be overwhelming for production runs with tens of thousands of processes. Spreading these file open operations across a longer time interval, and performing fewer large writes rather than several small writes are two potential strategies for improving the I/O performance of the S3D version we used, based on this preliminary event trace analysis.

Although our analysis of S3D’s I/O behavior has just begun, an early visualization of detailed event trace data has already enhanced our understanding of the S3D I/O strategy and suggested potential approaches for improving the I/O performance of this application.

### 5. SUMMARY

Understanding application I/O behavior is critical to overcoming gaps between the I/O demands of an application and the I/O capabilities of a system. We are developing an event tracing infrastructure for characterizing the I/O behavior of applications running on the Cray XT, a primary computing platform in the DOE Office of Science leadership computing efforts. We have begun to apply our prototype implementation to characterize the I/O behavior of two scientific applications of interest to the Office of Science, obtaining insight into possible optimizations for improving their I/O performance.

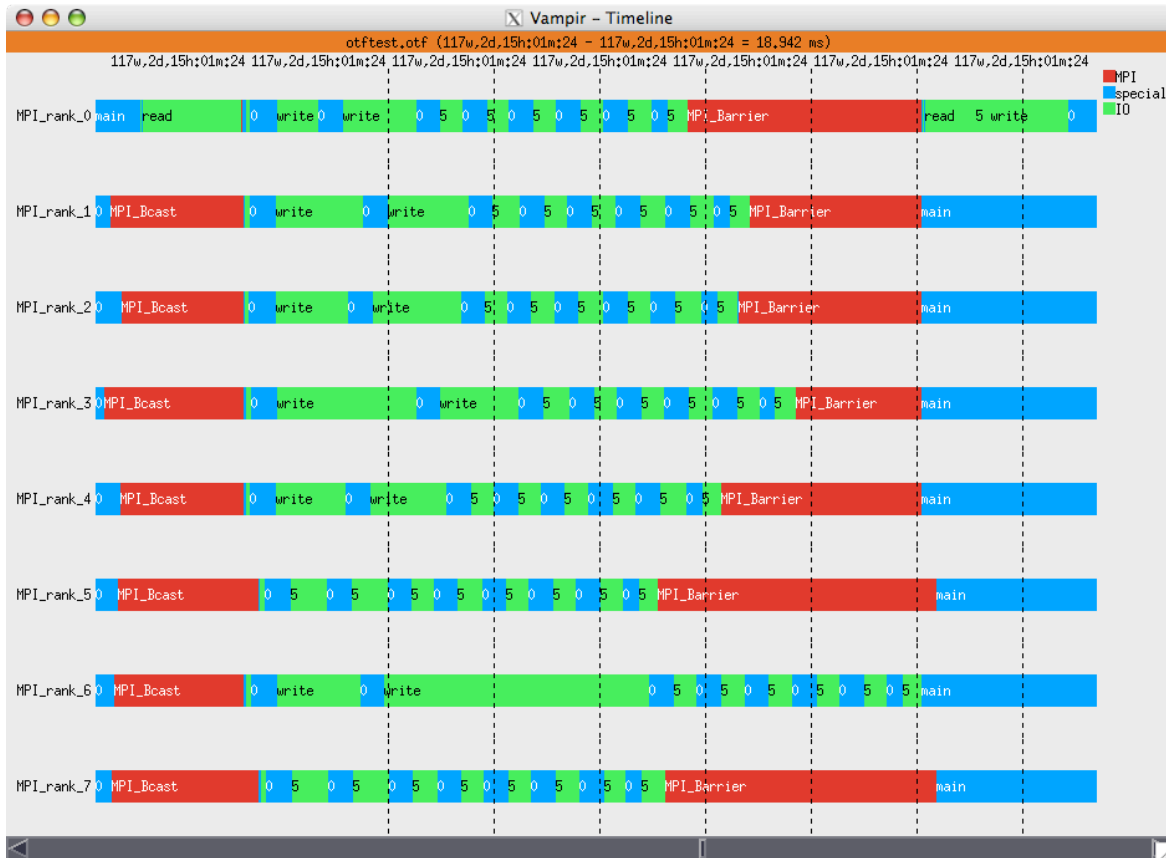


Figure 3: Vampir event trace timeline visualization showing one S3D checkpoint operation

In the future, we plan to use our I/O characterization software as the foundation for a suite of simple tools for I/O performance analysis, automated performance problem diagnosis, and automated performance tuning of application I/O behavior. We also plan to improve the scalability of our I/O performance data collection and analysis functionality using our MRNet [14] scalable tool infrastructure. Finally, we plan to continue our characterization work with applications beyond POP and S3D.

## 6. ACKNOWLEDGMENTS

Our thanks to Jeffrey S. Vetter, Weikuan Yu, and the other members of the ORNL Future Technologies Group for their constructive criticism of our work. Thanks also to Pat Worley for providing a mechanism for access to ORNL Leadership Computing Facility systems via the Performance Evaluation and Analysis Consortium End Station.

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract number DE-AC05-00OR22725.

## 7. REFERENCES

- [1] S.R. Alam, R.F. Barrett *et al.*, “An Evaluation of the ORNL Cray XT3,” *International Journal of High Performance Computing Applications*, 21(4), 2007 (to appear).
- [2] D. Becker, F. Wolf *et al.*, “Automated Trace-Based Performance Analysis of Metacomputing Applications,” Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2007.
- [3] M.B. Blackmon, B. Boville *et al.*, “The Community Climate System Model,” *BAMS*, 82(11):2357--76, 2001.
- [4] J.H. Chen and H.G. Im, “Stretch effects on the Burning Velocity of turbulent premixed hydrogen-Air Flames,” Proc. Comb. Inst, 2000, pp. 211-8.
- [5] Cray Inc., *Cray XT5 Family of Supercomputers*, <http://www.cray.com/products/xt5/index.html>, 2007.
- [6] L. DeRose and D.A. Reed, “SvPablo: A Multi-Language Architecture-Independent Performance Analysis System,” Proc. International Conference on Parallel Processing (ICPP'99), 1999, pp. 311–8.
- [7] P.W. Jones, P.H. Worley *et al.*, “Practical performance portability in the Parallel Ocean Program (POP),” *Concurrency and Computation: Experience and Practice*, 17(10):1317-27, 2005.
- [8] S.M. Kelly and R. Brightwell, “Software Architecture of the Light Weight Kernel, Catamount,” in *Cray User Group Technical Conference*. Albuquerque, NM, 2005
- [9] Message Passing Interface Forum, “MPI-2: A Message Passing Interface Standard,” *International Journal of Supercomputer Applications and High Performance Computing*, 12(1–2):1–299, 1998.

- [10] B.P. Miller, M.D. Callaghan *et al.*, “The Paradyn Parallel Performance Measurement Tools,” *IEEE Computer*, 28(11):37–46, 1995.
- [11] W.E. Nagel, A. Arnold *et al.*, “VAMPIR: Visualization and Analysis of MPI Resources,” *Supercomputer 63*, 12(1):69–80, 1996.
- [12] Paratools Inc., *Open Trace Format*, <http://www.paratools.com/otf.php>, 2007.
- [13] V. Pillet, J. Labarta *et al.*, “PARAVER: A Tool to Visualize and Analyze Parallel Code,” Proc. WoTUG-18: Transputer and Occam Developments, 1995, pp. 17–31.
- [14] P.C. Roth, D.C. Arnold, and B.P. Miller, “MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools,” Proc. SC2003, 2003.
- [15] S. Shende and A.D. Malony, “The TAU Parallel Performance System,” *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.
- [16] J.S. Vetter and M.O. McCracken, “Statistical Scalability Analysis of Communication Operations in Distributed Applications,” *Principles and Practice of Parallel Programming*, 36(7):123–32, 2001.