Characterizing I/O Performance Using the TAU Performance System

Sameer SHENDE[a,1], Allen D. MALONY[a], Wyatt SPEAR[a], and Karen SCHUCHARDT[b]
[a] *Performance Research Laboratory, University of Oregon*
[b] *Pacific Northwest National Laboratory, Washington*

Abstract. TAU is an integrated toolkit for performance instrumentation, measurement, and analysis. It provides a flexible, portable, and scalable set of technologies for performance evaluation on extreme-scale HPC systems. This paper describes alternatives for I/O instrumentation provided by TAU and the design and implementation of a new tool, *tau_gen_wrapper*, to wrap external libraries. It describes three instrumentation techniques - preprocessor based substitution, linker based instrumentation, and library preloading based replacement of routines. It demonstrates this wrapping technology in the context of intercepting the POSIX I/O library and its application to profiling I/O calls for the Global Cloud Resolution Model (GCRM) application on the Cray XE6 system. This scheme allows TAU to track I/O using linker level instrumentation for statically linked executables and attribute the I/O to specific code regions. It also addresses issues encountered in collecting the performance data from large core counts and representing this data to correctly identify sources of poor I/O performance.

Keywords. POSIX I/O, MPI-IO, TAU, Instrumentation, and GCRM.

Introduction

The ability to grow parallel applications to execute efficiently at extreme scales is most often coupled with the ability to increase the size of the problem being solved. Consequently, applications will see a steady increase in the volume of data they need to process and solution results they need to generate. While parallel efficiency has previously been mainly concerned with computing and memory optimization, it is clear the I/O performance is becoming a key bottleneck in many cases at the extreme scale. As the volume of data an application reads and writes increases, it is important to assess the scalability of I/O operations as a key contributor to overall application performance. Observing the performance of the I/O operations requires instrumentation to be inserted

[1] Corresponding Author.

in the I/O library layers of the software stack, including commonly used I/O interfaces such as POSIX I/O and MPI-IO. However, characterization of I/O performance must also be done with respect to application context to fully understand overall performance impact.

In this paper, we present extensions to the TAU Performance System® project to automate the creation of wrapper interposition libraries that intercept library calls and insert probes to trigger performance measurements. Section 1 describes six techniques for inserting instrumentation in I/O operations in an application and discusses in the advantages and disadvantages of each approach. TAU's library wrapping capability has proven to be an effective technique to characterize I/O performance, particularly for automating the instrumentation of I/O packages in cases where the source code may not be available for direct probe-based instrumentation. This technique is described in Section 2. In Section 3, we describe our work in applying TAU's I/O tracking features to measure the performance of the Global Cloud Resolving Model (GCRM) application on a Cray XE6 system. The paper concludes with directions for future work.

1. Library Wrapping and I/O Instrumentation

Many parallel applications are constructed using software library packages with interfaces callable from standard programming languages. Packages are often layered, internally calling other libraries to implement underlying functionality, which can be hidden to the user. Having an ability to intercept package calls at library routine interfaces enables performance tools to gather both semantic (contextual) and performance data for analysis purposes. I/O libraries represent a challenging case study for performance observation. During the compilation process, there are several phases of code transformation where instrumentation may be inserted to track I/O calls.

1.1. Pre-processor Based Instrumentation

Prior to compiling C and C++ code, compilers pre-process the source code and expand header files and macros before the code generation phase. This provides the tools an excellent opportunity for tools to intercept and replace I/O calls, such as POSIX I/O *open*, *close*, *read*, and *write*, with their instrumented counterparts. This can be done by re-defining a header file (*unistd.h*) that internally redefines the name of an

I/O routine as a macro that redirects all references to the given call with another. The compiler's pre-processor then replaces all references to the I/O call at the callsite in the source code with the corresponding call defined by the tool (e.g., *read* replaced by *tau_read*).

To use this approach, a tool simply adds an include directive to a directory that contains the tool's header file that performs the substitution. In addition to the instrumentation, a tool then implements the wrapper interposition library where each wrapper routine (*tau_read*) contains whatever measurement statements before and after calling the original call (*read*). Because the wrapper routine knows about the original routine's interface, it has access to and can examine the parameters that flow through the call, for instance, to assess the size of data arrays being passed. Once the tool wrapper library has been implemented, the tool is enabled by linking it to create the executable.

The above approach, described more fully in [1], works well for C and C++ programs where POSIX I/O calls are replaced explicitly during compilation. Unfortunately, this approach does not extend well to Fortran programs. Moreover, the instrumentation technique is limited to application code regions where the source code is available for recompiling. I/O library calls made from code where source code is not available (e.g., other libraries) will not be seen.

1.2. Source-Based Instrumentation

While the above technique is specific to C and C++, TAU does support instrumentation of Fortran I/O constructs by re-writing the source code. TAU's instrumentation tool (*tau_instrumentor*) examines the source code, its PDB file as generated by the Program Database Toolkit (PDT) [4], and re-writes the Fortran I/O calls in the instrumented source code. The user may specify I/O instrumentation requests via a selective instrumentation file that is passed to TAU's compiler scripts using environment variables. In this method, performance measurement code are inserted directly in the source code along with calls to track the sizes of arrays that are passed to the write and read calls.

This above approach leverages work described in [3] where in order to track memory leaks it was necessary to instrument the source code around memory allocation/deallocation calls. The technique works equally well for I/O. However, the dependency on source code being available is still a problem.

1.3. MPI-IO Instrumentation

The MPI message passing libraries provides a name-shifted interface that permits tools, including TAU [4], to intercept calls using the PMPI name-shifted interface. TAU additionally uses this support to create a wrapper library for MPI-IO calls (e.g., *MPI_File_read*) that internally calls the name-shifted interface (e.g., *PMPI_File_read*). Like before, the wrappers can examine the arguments that flow through the I/O calls to compute volume and bandwidth of individual I/O operations. In addition to TAU, this instrumentation technique is used in a wide variety of HPC tools including Scalasca[8], VampirTrace[9], Score-P [10], MPIP[12], and IPM[7]. However, library interposition through name-shifted interfaces is only available as a technique if such interfaces are implemented in the library. This is not the case with POSIX I/O.

1.4. Runtime Preloading of Instrumented Library

Many HPC operating systems such as Linux, Cray Compute Node Linux (CNL), IBM BlueGene Compute Node Kernel (CNK), Solaris permit pre-loading of a library in the address space of an executing application specifying a dynamic shared object (DSO) in an environment variable (*LD_PRELOAD*). It is possible to create a tool based on this technique that can intercept all I/O operations by means of a wrapper-library where the POSIX I/O calls are redefined to call the global routine (identified using the *dlsym* system call) internally. Preloading instrumented libraries is a powerful technique implemented by the runtime linker and is used in TAU [3], VampirTrace[9], and IOTrack[11]. While it can resolve all POSIX-IO calls and operates on un-instrumented executables, it only supports dynamic executables. Static executables are used by default on IBM BlueGene and Cray XE6 and XK6 systems, although dynamic executables may be created using the *–dynamic* command line flag. A different technique will be necessary to support static binaries.

1.5. Linker-Based Instrumentation

A linker can redirect references to a wrapped routine when it is invoked with a special flag on the command line (*-Wl,-wrap,function_name*). In this case, the application does need to be re-linked to use the wrapped library, but this instrumentation technique overcomes the limitation of the previous approach provided by the runtime linker and may be used with both static and dynamic executables. TAU has applied this

approach to instrument POSIX I/O calls by creating a wrapper library. Since the number of wrapped routines that may be present in a library might be potentially large, listing each routine on the linker's command line can interfere with predefined system limits for command line length. Instead, a linker may read a file that contains wrapped symbol names and expand these internally to construct the appropriate command line. TAU's compiler scripts have been updated to automatically add the necessary flags to the linker command line when the user sets a special I/O instrumentation flag (*-optTrackIO*) in the TAU_OPTIONS environment variable. We describe this approach in greater detail in Section 3 on GCRM profiling.

1.6. Instrumented External I/O Libraries

When the user needs to evaluate the time spent in un-instrumented I/O libraries (such as HDF5 [13]) and other system libraries, it is important to be able to generate custom user-directed wrapper libraries. These wrapper libraries may be pre-loaded at runtime or re-linked to create an instrumented binary using linker-based instrumentation as described above. However, manually building these libraries may prove to be cumbersome. In the next section, we describe a way to automate the creation of instrumented wrapper libraries in the TAU performance system.

2. Automating Generation of Wrapper Libraries

Wrapper libraries can greatly enhance the performance observation capabilities of a tool. Given the variety of interesting software packages one might want to observe during execution, it is important to facilitate the creation of wrapper libraries as much as possible. We created a TAU tool, *tau_gen_wrapper*, to do the following: take an interface declaration of a library in the form of a header file and generate a wrapper library for TAU instrumentation. It uses the PDT static analysis tool to parse the header file and generates for each routine, a complete representation of its signature. A signature consists of the return type, the routine name, and a list of arguments. Each argument, in turn, consists of the argument type and an optional argument name. The user may supply an optional selective instrumentation file that describes an exclude or include list of routines or files that is used to select the subset of routines that are wrapped.

The *tau_gen_wrapper* architecture is shown in Figure 1. Internally, it invokes the parser and the *tau_wrap* tool and builds the instrumented source code emitted by it. The wrapper generator tool allows the user to choose from the following instrumentation techniques for creating a wrapper library:

- Pre-processor based redirection of routines (Section 1.1)
- Runtime preloading of instrumented library using the runtime-linker (Section 1.4)
- Linker-based instrumentation (Section 1.5)

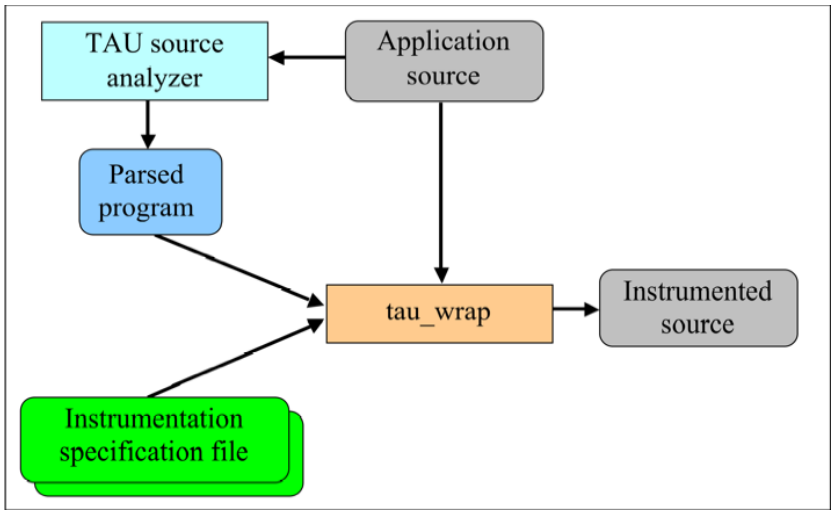

Figure 1. Architecture of *tau_gen_wrapper*, a tool that automates generation of wrapper libraries.

The wrapper generator tool has been used successfully to create wrappers for the CUDA [5] and HDF5 [13]. The next section describes the use of TAU's linker-based instrumentation capabilities for tracking POSIX I/O.

3. Profiling GCRM

The Global Cloud Resolving Model (GCRM) being developed by Randall et al [6] will model climate on the entire globe at a horizontal grid spacing of at least 4km and vertical dimension on the order of 256 layers resulting in over 10 billion cells. A single cell-based variable written in single precision will require approximately 43 GB of disk storage. Corner data will require 85 GB and edge data 128 GB. A single snapshot of history data will require 1.8 TB of storage as currently

configured. Climate scientists will want to write data as frequently as possible (down to the order of minutes) while maintaining an IO cost below 10% of the overall simulation. Obviously, the efficiency of the I/O is of critical importance.

Understanding and optimizing the behavior of the I/O system for an application is difficult for several reasons. First there are several layers in the I/O stack, some of which are proprietary software. Second, there are many options for controlling these layers varying from optional arguments, to hints to alternative APIs. Third, there are often multiple implementations of some of the layers. **Figure 2** shows the layers and alternatives considered for GCRM.

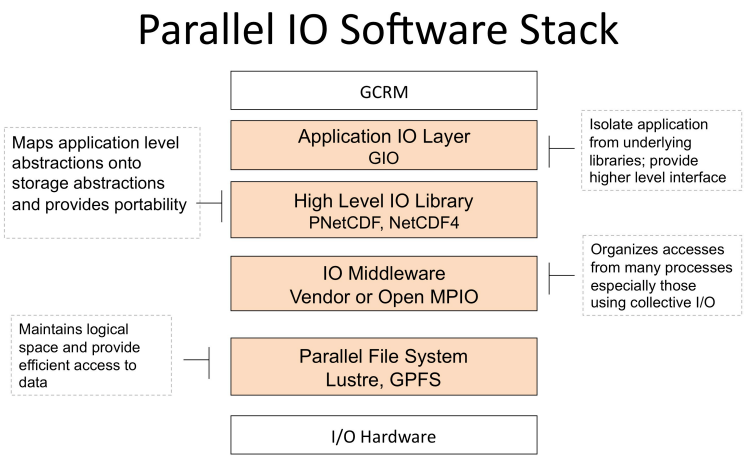

Figure 2. The GCRM I/O stack and some of the variations that make profiling a complex task.

It is still critical to be able to profile all the layers of the GCRM I/O in order to determine where the true bottlenecks reside. TAU provides the capabilities both to look deep into the various API layers and to organize and analyze the numerous configurations under evaluation.

Here we discuss our experiences applying TAU on the CRAY systems at NERSC using PNetCDF [14], the Cray MPI-IO library, and Lustre. We have ongoing efforts to examine the other variations shown. Our use of TAU was not limited to examining I/O, as we also were looking at the performance of the GCRM overall. In this respect, TAU supports breaking of the profile into multiple phases [4]. For our experiments,

performance was evaluated with respect to three phases: initialization, I/O, and the numerical model itself. This enabled us to look at performance of each phase in its context and see the relative cost of each phase.

Figure 3 shows the profile configuration used during initial performance analysis. Note that phases can be defined at the function level or through code ranges. Our main purpose for identifying the *Init* phase, was to ensure that one-time initialization costs would not skew the overall model profiling results. Selective loops within the numerical model were also profiled in more detail based on initial results.

```
BEGIN_INSTRUMENT_SECTION
loops routine="UTILITIES_ADVECTION_HORZ:DEL_DOT_XV"
loops routine="RRTMG_SW_SPCVRT::SPCVRT_SW"
loops routine="MULTIGRID_SOLVER_3D_LYR::MLG_RLX2_3D"
loops routine="MULTIGRID_SOLVER::RELAX"
static phase name="Init" file="ZGrd_main.pp.F90" line=42 to line=87
static phase  routine="ZGRD_ZGRD::ZGRD"
static phase  routine="GIO_DRIVER"
END_INSTRUMENT_SECTION
```

Figure 3. TAU selective instrumentation file that defines three phases: Init, I/O (GIO_DRIVER) and ZGrd (the numerical model)

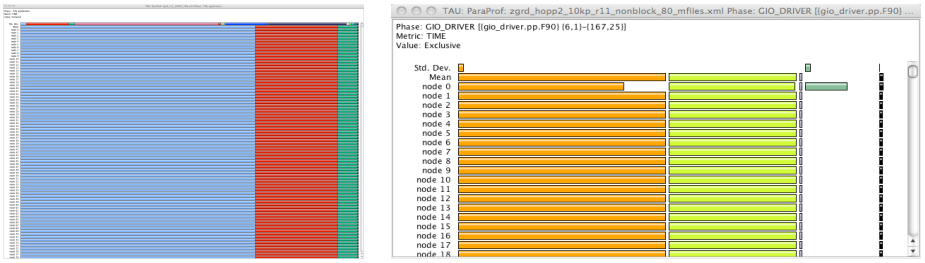

Figure 4. Right: The initial TAU profile when using TAU's phase capability. The first column represents the computational phase, the second column represents the initialization phase, and the third column represents the I/O phase. Left: I/O phase detail with columns *MPI_Write_all*, *MPI_File_open*, POSIX I/O write, and everything else. I/O aggregators are clearly identifiable.

Figure 4 (left) shows the phase profile summary screen within TAU for a typical GCRM run using the TAU profile described above. Profile runs tend to be of very short duration, which causes an over-representation of the *Init* phase. The interesting take-way from this particular run is that I/O is taking a reasonable amount (8%) time relative to the numerical model itself. Right clicking on the phase column and requesting the phase detail will result in a graph such as the I/O phase detail shown in

Figure 4 (right). Right clicking on a "node" label will generate a detailed profile of function times for that processor as shown in Figure 5.

In these early runs, we were clearly able to see that the collective open calls were taking significant time and that it was not due to the cost of POSIX open. Sharing these profiles with Cray engineers led to a detailed analysis of the cost of *MPI_file_open* and resulted in several changes to the Cray library that will be included with the next release.

A final important feature provided by the new TAU I/O profiling capability is a summary of the read and write sizes and read and write bandwidths per processor. An example is shown in Figure 6. Here we can see that we are successfully writing large chunks of data and that the mean is also large. Per-processor bandwidth and mean bandwidth can also be seen.

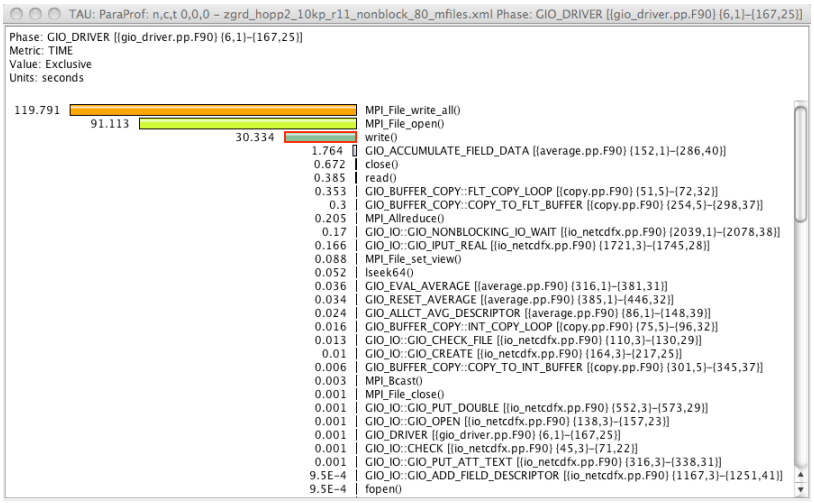

Figure 5. I/O phase function call detail for processor 0.

4. Conclusions

Understanding the performance of software packages in combination with the applications in which they are used requires an ability to capture important events and performance data at the library interfaces. In this paper, we presented several techniques for creating wrapper libraries with the TAU performance system. We then demonstrated these techniques for tracking I/O performed by the GCRM application on a Cray XE6 system. Our work has been instrumental in improving I/O performance in GCRM.

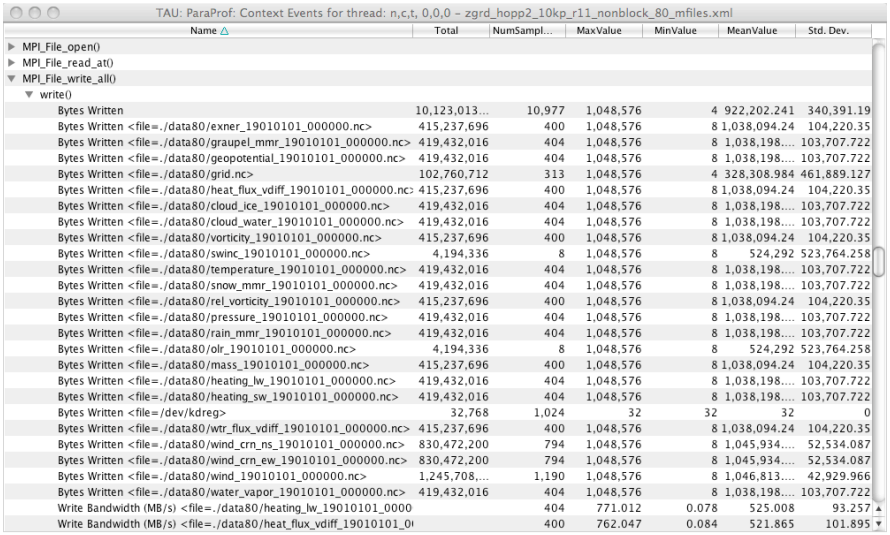

Figure 6. Processor 0 Context Event Window showing write sizes and bandwidths.

5. Acknowledgements

The research was conducted at the University of Oregon and Pacific Northwest National Laboratory (PNNL) under contract no. 113907 from Battelle Memorial Institute, PNNL, and grants DE-SC0001777, DE-FG02-08ER25846, and DE-SC0006723 from the Department of Energy, Office of Science. Resources from NERSC were utilized in this work.

References

[1] S. Shende, A. Malony, A. Morris, and D. Cronk, "Observing Parallel Phase and I/O Performance Using TAU," in Proc. DoD UGC Conference, IEEE Computer Society, 2009.
[2] S. Shende, A. D. Malony, S. Moore, and D. Cronk, "Memory Leak Detection in Fortran Applications using TAU," in Proc. DoD UGC Conference, 2007.
[3] S. Shende, A. D. Malony, A. Morris, and A. Wissink, "Simplifying Memory, I/O, and Communication Performance Assessment using TAU," in Proc. DoD UGC Conference, IEEE Computer Society, 2010.
[4] S. Shende, and A. D. Malony, "The TAU Parallel Performance System," in IJHPCA, Vol. 20 (2), pp. 287-311, Summer 2006, http://tau.uoregon.edu.
[5] A. D. Malony, S. Biersdorff, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, C. Lamb, "Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs," in ICPP 2011.
[6] Global Cloud Resolving Model, http://kiwi.atmos.colostate.edu/gcrm/, 2011.
[7] IPM, http://ipm-hpc.sourceforge.net/, 2011.
[8] Scalasca, http://www.scalasca.org, 2011.
[9] VampirTrace, www.tu-dresden.de/zih/vampirtrace, 2011.
[10] Score-P, www.score-p.org, 2011.
[11] IOTrack, http://www.pdc.kth.se/~pek/iotrack/, 2011.
[12] mpiP: Lightweight, Scalable MPI Profiling, http://mpip.sourceforge.net/, 2011.
[13] HDF Group, http://www.hdfgroup.org, 2011.
[14] Argonne National Laboratory, "Parallel-NetCDF: A High Performance API for NetCDF Access," http://trac.mcs.anl.gov/projects/parallel-netcdf, 2011.