Isolating Runtime Faults with Callstack Debugging using TAU

Sameer Shende, Allen D. Malony, John C. Linford
ParaTools, Inc.
Eugene, OR
{Sameer, malony, jlinford}@paratools.com

Andrew Wissink
U.S. Army Aeroflightdynamics Directorate
Ames Research Center
Moffett Field, CA
Andrew.m.wissink@us.army.mil

Stephen Adamec
University of Alabama at Birmingham, AL
Stephen.a.adamec.ctr@hpcmo.hpc.mil

Abstract—Traditional debugging tools do not fully support state inspection while examining failures in multi-language applications written in a combination of Python, C++, C, and Fortran. When an application experiences a runtime fault, such as numerical or memory error, it is difficult to relate the location of the fault to the original source code and examine the performance of the application. We present a tool that can help identify the nature and location of runtime errors in a multi-language program at the point of failure. This debugging tool, integrated in the TAU Performance System®, isolates the fault by capturing the signal associated with it and reports the program callstack. It captures the performance data at the point of failure, stores detailed information for each frame in the callstack, and generates a file that may be shipped back to the developers for further analysis. The tool works on parallel programs, providing feedback about every process regardless of whether it experienced the fault. This paper describes the tool and demonstrates its application to the multi-language CREATE-AV applications Kestrel and Helios. The tool is useful to both software developers and to users experiencing runtime software issues as the file output may be exchanged between the user and the development team without disclosing potentially sensitive application data.

***Index Terms*—TAU, Instrumentation, Python, Measurement, Debugging, Memory, I/O, Helios, Kestrel.**

I. Introduction

When application software experiences a runtime failure or performance problem, it is important for concise information about the error to be communicated to the development team. Although the next generation of HPC engineering simulation tools for the DoD use a multi-language implementation [1,2], with numeric kernels written in C, C++, or Fortran driven by scripting or interpreted languages like Python or Java, current debugging tools do not operate well in this multi-language environment. A multi-language paradigm facilitates modular software and a more straightforward exchange of software between different development groups. However, it introduces a number of complexities in terms of debugging and memory management.

Determining execution context in relation to program parts and the language features used to create them is non-trivial. At present, if software code experiences a runtime issue (e.g., a numerical or memory error), the program generates a core file which, while useful for debugging, contains limited information about the status of the execution, the memory profile, the process call stack, and the system resources in use. In order to resolve the problem, software developers need to understand the nature of the exception, where it occurred, how long the program executed, and the routines invoked prior to the error. A full view of the multi-language execution state, with the attribution of low-level faults to higher-level software context, is required to gain better insight into the layered workings of the application. The same reasoning extends to the importance of retaining other runtime program information that might be useful in understanding what led up to the error. In the case of parallel programs, for example, the ability to save parallel performance measurements after a long-running execution encounters an exception could be extremely valuable, instead of losing all performance information because the tools are unable to retain it. Just being able to properly manage an orderly shutdown of a faulty parallel application when only a few (or even one) process experiences the fault is a challenge. Also, exchanging large core files (generated by each core) with the developers is cumbersome, especially when working with parallel applications that use dynamic shared objects (DSOs). It is also of limited value when the fault occurs in a DSO where addresses are relative to an offset at which the shared library is loaded at runtime.

Another complexity in HPC software used by the DoD is the mechanism for reporting the software issue. Security requirements in the DoD often preclude the use of forums to

discuss software issues if geometry or input conditions are proprietary, sensitive, or classified. Software issues are generally independent of geometry or inputs but the development team must be able to reproduce the issue in order to effectively fix it, and this is difficult for cases with limited distribution rights.

This paper presents a diagnostic tool for runtime faults that aims to replace the standard core file output with more effective and concise information gathered during execution and at the time of the errant behavior, including data about the computational performance, memory usage, call stack, and I/O operation. The diagnostic file is self-contained and interchangeable between machines in the TAU standard profile format, so a user that experienced an execution problem can send the diagnostic file to the development team for further analysis to determine the source of the problem. The format for this file is human readable and open.

The next section presents the design approach and technical issues. Section 3 describes the development of the tool for multi-language applications based on a Python-driver model. We have successfully applied this tool to the Helios [1] and Kestrel [2] codes and we report our findings. Related work is presented in Section 4. We conclude the paper with a discussion of the next development and evaluation steps.

II. Design Approach

When an application experiences a runtime error such as a segmentation violation (e.g., caused by an access through a null pointer), execution of an illegal instruction, or floating-point exception (e.g., division by zero), it is critical to diagnose the problem with respect to the execution context. Merely reporting the text output of the execution is rarely sufficient to fix the problem, particularly in multi-language applications. The developers typically need to understand the nature of the exception, where it occurred, how long the program executed, and the routines invoked prior to the error. This requires a full view of the execution state to gain a better insight into the program's layered workings of the multi-language program. Additionally, system information such as the operating system kernel version, the extent of heap memory utilization, number of cores, and other application specific parameters are important in fully understanding the error.

The key to solving the problem of observing execution state is in capturing (unwinding) the call stack at the location of the error across the many layers of languages and libraries. The diagnostic tool we created integrates a call stack capture module in TAU to accomplish this [20]. TAU unwinds the calling stack of each thread of execution and records it in the profile file format with context specific information such as the calling routine name, file name, and source line number for each frame in the program's calling stack, when available. The tool operates at runtime, automatically interposing with the application to generate diagnostics. This involves registering signal handlers, examining the program callstack using GLIBC library *backtrace* API [15] at the point of failure, maintaining and updating address maps for DSOs as they are loaded and unloaded, and translating the address from each frame in the callstack using the GNU *Binutils* [13] to useful program information before the information is lost and the program terminates. However, a tool to recover state information from an errant parallel program must consider the cases where only a subset (perhaps just one) of the threads experiences a failure. Here, it is necessary to inform the other processes to gracefully terminate, capturing relevant diagnostic information as they do so. The design choices for this tool are described in [20].

III. Implementation

We have developed a callstack capture module in TAU that works with Python and can help us understand the I/O and memory usage in a mixed language application at the point of failure. The user activates this module by setting the *TAU_TRACK_SIGNALS* environment variable prior to executing the program using *tau_exec*. The *tau_exec* tool preloads a TAU DSO in the address space of the executing application and registers the handlers for capturing a signal at the point of failure. Additionally, a TAU Python wrapper helps to instrument all Python functions. Figure 1 shows how the application is invoked with the I/O and memory inspection command line options. No changes are required in the application source code or binaries. If and when a runtime fault occurs, TAU isolates it by capturing the callstacks for each rank of the MPI program. There are two distinct callstack entities – the system callstack and the callstack maintained by TAU. The TAU callstack represents the sequence of events that are recorded by entry and exit instrumentation in interval timers. Because TAU supports a wide variety of programming languages, this callstack is independent of the language and we can see a combination of events from C, C++, Fortran and Python in this view. The Python events represent routine names in the Python scripts that TAU gets from the interpreter. These events are typically not visible to traditional debuggers such as *gdb* [14]. By combining the two callstacks consistently within one tool, TAU permits debugging of system backtraces along with performance data generated by higher language level interval and atomic events providing a consistent view that will allow developers to get an accurate view of the program execution with or without fault diagnostic information.

TAU is integrated in the runtime layer of the *PToolsRTE* package [21] that is used by the CREATE-AV codes. This allows a user to source a single configuration file or load an environment module to set all paths needed for launching the codes using TAU, as shown in Figure 1.

Launching TAU's *ParaProf* profile browser allows the user to examine the callstacks. Figure 2 shows the system callstack at the point of failure in Kestrel initialization code. This data is captured in the BACKTRACE metadata fields. Figure 3 shows a small segment of the callstack, zoomed in for better readability. By clicking on an entry, TAU's source browser can show the source code at the location of the segmentation fault for any frame.

In addition to capturing the frames, the tool retains the performance data TAU has collected for the run. Figure 4 shows the exclusive and inclusive time spent in code regions. We see that the application executed on 8 processors for 302.9 seconds before terminating with the error. In addition to the code region performance data, the TAU callstack information reveals the Python routines that were called before the segmentation fault. Because I/O activity is also measured, the

message sizes for communication operations can be record. Figure 5 highlights these features for the Kestrel testcase.

```
[sameer@m0297 example]$ source $PET_HOME/pkgs/ptoolsrte/etc/ptoolsrte.cshrc
[sameer@m0297 example]$ which pyMPI
/mnt/cfs/pkgs/PTOOLS/pkgs/ptoolsrte-0.55/linux-redhat5.6-gnu-x86_64/pyMPI-2.5b0/bin/pyMPI
[sameer@m0297 example]$ which tau_exec
/mnt/cfs/pkgs/PTOOLS/pkgs/ptoolsrte-0.55/linux-redhat5.6-gnu-x86_64/tau-2.21.2/x86_64/bin/tau_exec
[sameer@m0297 example]$ setenv TAU_TRACK_SIGNALS 1
[sameer@m0297 example]$ setenv TAU_CALLPATH_DEPTH 100
[sameer@m0297 example]$ cat wrapper.py
import tau

def OurMain():
    import samarcrun

tau.run('OurMain()')

[sameer@m0297 example]$ mpirun -np 2 tau_exec -T icpc,python,mpi,pdt -io -memory pyMPI ./wrapper.py

initializing SAMARC
SAMINT::initialize()
building SAMARC initial AMR mesh
SAMINT::buildGrids()
printing output
SAMINT::getGlobalNumberPatches()
SAMINT::getLocalNumberPatches()
SAMINT::writePlotData()
-----------------------------------------------------------
          step:  1      time:  0.0
SAMINT::timestep()
initializing SAMARC
SAMINT::initialize()
building SAMARC initial AMR mesh
SAMINT::buildGrids()
printing output
SAMINT::getGlobalNumberPatches()
SAMINT::getLocalNumberPatches()
SAMINT::writePlotData()
-----------------------------------------------------------
          step:  1      time:  0.0
SAMINT::timestep()
TAU: Caught signal 8 (Floating point exception), dumping profile with stack trace: [rank=1, pid=25501, tid=0]...
TAU: Caught signal 8 (Floating point exception), dumping profile with stack trace: [rank=0, pid=25502, tid=0]...
--------------------------------------------------------------------------
mpirun has exited due to process rank 1 with PID 25428 on
node m0297.mana exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpirun (as reported here).
--------------------------------------------------------------------------
[sameer@m0297 example]$ paraprof --pack create_io_mem.ppk
Loading data...
Packing data...
[sameer@m0297 example]$
```

Fig. 1. Executing an application with the tau_exec tool captures the program state when a runtime fault occurs.

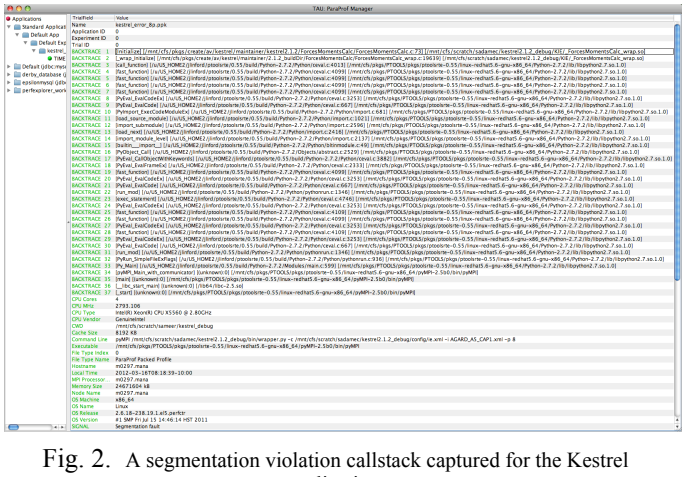

Fig. 2. A segmentation violation callstack captured for the Kestrel application.

```
BACKTRACE  1  [initialize] [/mnt/cfs/pkgs/create/av/kestrel/maintainer/kestrel2.1.2/ForcesMomentsCalc/ForcesMomentsCalc.c:73] [/mnt/
BACKTRACE  2  [_wrap_Initialize] [/mnt/cfs/pkgs/create/av/kestrel/maintainer/2.1.2_buildDir/ForcesMomentsCalc/ForcesMomentsCalc_wrap.c:19639]
```

Fig. 3. Zooming into a segment of the callstack shows the routine name and the source code location.

It is important to understand that a distinct callstack and performance profile is captured for each rank. Different ranks may have different faults (or none at all) and it is necessary to see all states. It is possible to analyze any particular rank, such as rank 7 in Figure 6. The location of the error can also be seen across all ranks in Figure 7 (the bargraph indicates the number of samples of the error event on each rank).

The same application can be re-executed to generate I/O and memory data using *the -io -memory* command line options for *tau_exec*. Figure 8 shows the peak heap memory usage (~1.3GB) for the application on rank 0. Figure 9 shows the TAU callstack with the I/O and memory statistics leading up to the point of failure on rank 0. The total memory allocated till the point of failure was ~3GB and the total de-allocated was 2.24GB.

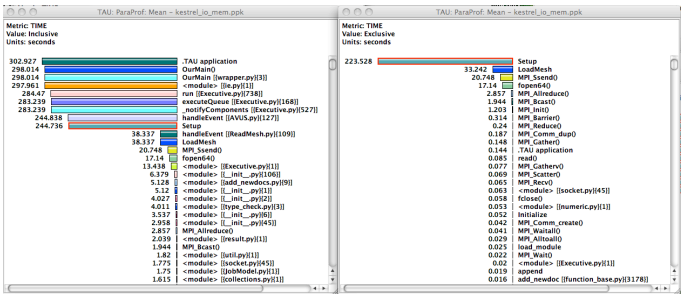

Fig. 4. Inclusive (left) and exclusive (right) time spent in Kestrel code regions.

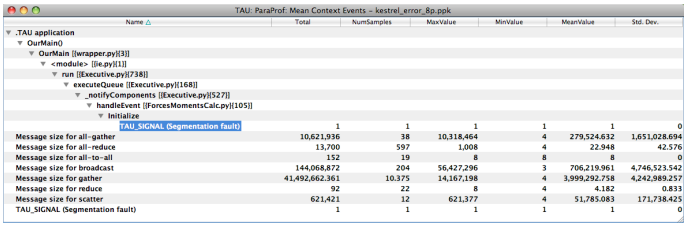

Fig. 5. TAU's callstack reveals the Python routines that lead to the segmentation fault.

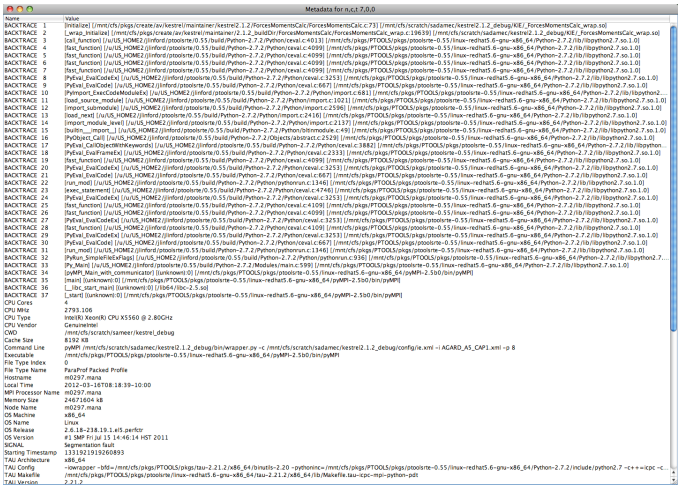

Fig. 6. System callstack leading up to the fault on the MPI rank 7.

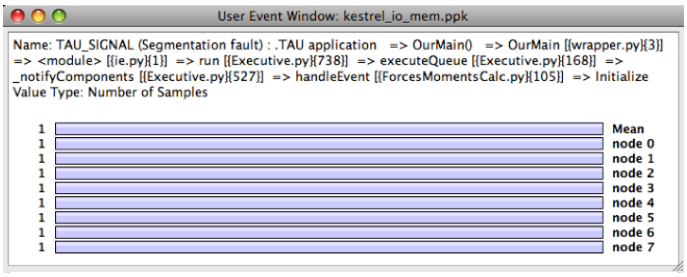

Fig. 7. Callstack and the location of the error shown for all MPI ranks.

TAU: ParaProf: Context Events for: n,c,t 0,0,0 – kestrel_io_mem.ppk

Name	Total	NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.
.TAU application						
Memory Utilization (heap, in KB)		47,690,869	1,319,090.592	0.023	275,509.524	38,071.609
malloc size (bytes)	3,012,988,071	23,869,030	496,523,280	1	126.23	113,322.074
free size (bytes)	2,248,780,956	23,821,839	496,523,280	1	94.4	108,796.005
Bytes Read	158,530,329	19,180	1,126,076	2	8,265.398	9,607.032
Read Bandwidth (MB/s)		19,156	10,982	0.051	1,209.371	277.131
Bytes Read <file=/mnt/cfs/pkgs/create/av	151,920,360	18,545	8,192	8,172	8,191.985	0.543
Read Bandwidth (MB/s) <file=/mnt/cfs/pk		18,545	2,048	327.68	1,238.092	153.524
Message size for all-reduce	13,700	597	1,008	4	22.948	42.576
Bytes Written	64,053	238	4,161	2	269.13	973.394
Write Bandwidth (MB/s)		212	1,040.25	0.085	46.81	174.011

Fig. 8. Memory and I/O diagnostic information for rank 0.

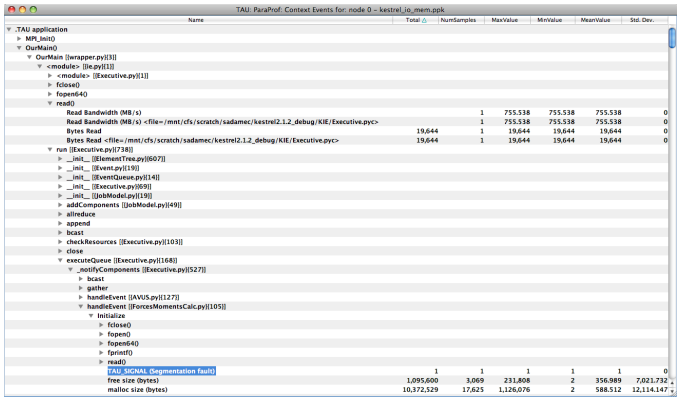

Fig. 9. Location of error with memory and I/O combined with the callstack data.

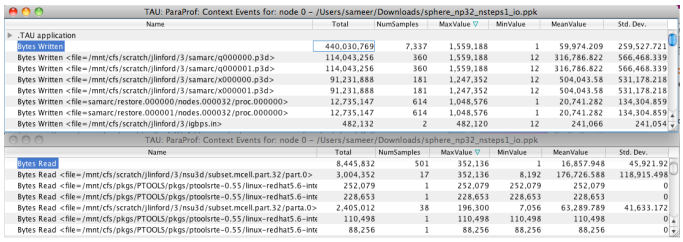

Fig. 10. I/O performed by MPI rank 0.

IV. Inclusion of Performance Analysis

The new debugging support in TAU discussed above works seamlessly with the performance measurement and analysis that TAU has traditionally been used for. Performance testing is not impeded in any way by having debugging enabled. To demonstrate this, we included performance evaluation of the CREATE-AV rotary-wing code Helios [1], which performs high-fidelity modeling of rotorcraft aero and structural dynamics. Helios consists of multiple components performing different parts of the multi-disciplinary application – computational fluid dynamics (CFD), computational structural dynamics (CSD), six degree of freedom dynamics (6DOF), etc. The different components are written in different languages – Fortran90, C, and C++ – which are integrated through a high-level Python-based infrastructure. Further details on the implementation and validation of Helios [1] and details of prior work of prior work integrating TAU with Helios [11] are outside the scope of this paper.

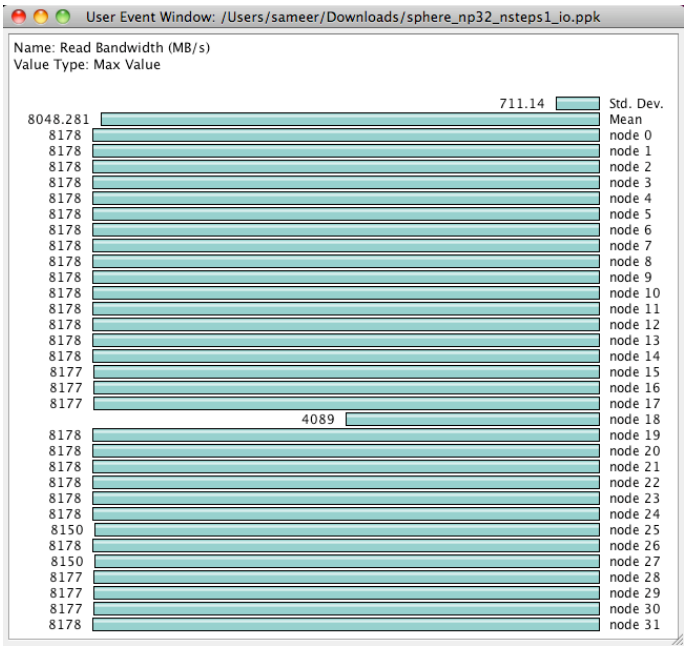

Fig. 11. Peak read bandwidth by MPI rank.

To demonstrate use of the new tool to assess multi-language performance, we executed Helios using *tau_exec*. Figure 10 shows the profile from Helios with TAU instrumentation at the Python, mpi4py, MPI, C++, C, and Fortran. It shows an integrated view of fault diagnostic information with the TAU callstack that shows Python entities clearly in the profile. PToolsRTE v0.55 supports both pyMPI and mpi4py based executions using TAU. While the earlier example with Kestrel shows the performance data truncated at a location of failure, the user may use TAU to investigate the performance characteristics of the test case that executes without an error as well, as shown in Figure 10. Here we see the I/O characteristics of the application executed on Mana, a Dell system at the MHPCC DSRC. It shows that MPI rank 0 wrote ~440MB and read ~8MB of data. It also shows the extent of I/O performed on each file. Figure 11 shows the peak read bandwidth obtained for all ranks of the application. Here we see that during the course of the entire execution, rank 18 had a peak bandwidth of 4089 MB/s compared to nearly 8178 MB/s for other ranks. Having access to such detailed information about each rank and file can help developers identify sources of performance bugs.

V. Related Work

Debugging mixed language parallel programs that use Python is a daunting task. Commercial debuggers such as TotalView [16], DDT [17], and open source debuggers such as gdb [14] excel at generating backtraces for compiled executions. It is difficult if not impossible at this time to stop a program at a breakpoint, and move up or down the frames traversing Python and C boundaries in the same debugger, and examining and invoking Python routines and data structures. Python level entities are visible to performance evaluation tools that operate at the Python interpreter level. By merging the backtrace operations traditionally in the debugging domain with performance introspection, we create a hybrid tool capable

of diagnosing fault information based on performance instrumentation. Extensive work has been done in the area of callstack unwinding and sampling based measurements in the DyninstAPI, TAU, and HPCToolkit projects [18].

To better understand the performance characteristics of an application, both profiling and tracing are relevant. While profiling shows summary statistics, tracing can reveal the temporal variation in application performance. Among tools that use the direct measurement approach, the VampirTrace [12] package provides a wrapper interposition library that can capture the traces of I/O operations using the Linux preloading scheme used in *tau_exec*. Scalasca [7] is a portable and scalable profiling and tracing system that can automate the detection of performance bottlenecks in message passing and shared memory programs. Like many other tools, including VampirTrace, it uses library wrapping for MPI. TAU may be configured to use Scalasca or VampirTrace internally. TAU, VampirTrace, and Scalasca use the PAPI [4] library to access hardware performance counters present on most modern processors. However, only the *tau_exec* scheme provides the level of integration of all sources of performance information – MPI, I/O, and memory – of interest to us, with the rich context provided by TAU. With this support, we can utilize the VampirServer [5] robust parallel trace visualization system to show the performance data through scalable timeline displays the state transitions of each process along a global timeline. Profile performance data can also be easily stored in the PerfDMF database [7]. TAU's profile browser, ParaProf, and its cross-experiment analysis and data-mining tool PerfExplorer [6] can interface with the performance database to help evaluate the scalability of an application.

VI. Significance to DoD

When application software experiences a runtime failure or performance problem, it is important for concise information about the error to be communicated to the development team. Particularly for multi-language applications that use a mix of Python, Fortran, C, and C++, current solutions are inadequate. This leaves a gap in communication between users experiencing bugs and/or performance issues and the code development team. This project delivers a tool that consolidates the execution data required for diagnostic purposes by utilizing powerful techniques for comprehensive measurement in the presence of execution errors. The goals of the project are twofold; first, to develop a runtime fault reporting tool to assist with debugging multi-language applications, and second, to close the loop with developers for more rapid turnaround of bug fixes. Absent any errors, the tool will report diagnostic information to users about the computational performance, memory usage, and IO. Such information is useful for users to understand the computational characteristics of their application and for planning their computing requirements.

The new tool addresses security concerns by avoiding the need for the user to provide the problem geometry and inputs to the development team diagnosing problems. The diagnostic file contains only runtime information so it is more easily exchangeable to members of the development team that may not have the requisite permissions to see the problem data. This is particularly important for the large amount of classified or proprietary work that takes place within the DoD.

The extensions to TAU described in this paper - simplified assessment of error diagnostics coupled with I/O and memory inspection for un-instrumented and instrumented applications - expand the available capabilities, allowing users to ask questions such as:

- Where and when did the program experience an anomalous operation?
- What was the nature of the fault?
- What is the heap memory utilization in the application at the time of failure?
- Were there any memory leaks in the application?
- What was the level of nesting of the callstack?
- What was the routine name, source file name, line number and module name at the fault location?
- What were the performance characteristics of the application at that time?
- How much time did the application spend in I/O, and communication operations at the time of fault?

without re-compiling or re-linking the application and evaluate the fault diagnostics and performance of codes that use multiple languages such as Python, Fortran, C, and C++.

VII. Conclusions

Modern scientific software requires software components written in different languages to interact with one another. For instance, software being developed by the CREATE air vehicles program involves high-level Python scripts executing lower-level C, C++, and Fortran90 software components. While this multi-language paradigm enhances the re-usability and extensibility of software, programming it is challenging due to a lack of debugging tools available for inter-language execution and memory leak analysis. Also, the software is intended to run on high-end parallel computer systems that demand a high level of sophistication from performance evaluation tools. In this paper, we describe a new diagnostic tool under development for multi-language applications. The tool, which builds on our previous efforts using TAU for performance and memory analysis, reports memory and IO information as well as useful diagnostic call-stack information when the code experiences some form of anomalous operation, such as a segmentation fault. As CREATE software gets deployed for classified and proprietary projects within the DoD, users that experience bugs or anomalous behavior can provide details on the code execution through the diagnostic report generated by the tool, without revealing details about the application that the code is being applied to.

Acknowledgment

This work was supported by the DoD High Performance Computing Modernization Program (HPCMP) User Productivity Enhancement, Technology Transfer and Training (PETTT) program and through support provided by the DoD HPCMO to the HIARMS Institute and the CREATE program.

REFERENCES

[1] Wissink, A.M., V. Sankaran, B. Jayaraman, A. Datta, J. Sitaraman, M. Potsdam, S. Kamkar, D. Mavriplis, Z. Yang, R. Jain, J. Lim, R. Strawn, "Capability Enhancements in Version 3 of the Helios High-Fidelity Rotorcraft Simulation Code," AIAA-2012-0713, AIAA 50th Aerospace Sciences Meeting, January 2012, Nashville TN.

[2] S. A. Morton, B. Tillman, D. R. McDaniel, D. R. Sears, T.R. Tuckey, "Kestrel – A Fixed Wing Virtual Aircaft Product of the CREATE Program," in Proc. DoD HPCMP UGC 2009 Conference, IEEE Computer Society, 2009.

[3] S. Shende and A. D. Malony, "The TAU Parallel Performance System," Int'l Journal of High Performance Computing Applications, SAGE Publishers, 20(2): pp. 287- 311, Summer 2006. http://tau.uoregon.edu

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," Int'l Journal of High Performance Computing Applications, 14(3), pp. 189-204, 2000. http://icl.cs.utk.edu/papi/

[5] A. Knupfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel, "Introducing the Open Trace Format (OTF)," Proc. ICCS 2006, LNCS 3992, Springer, 2006.

[6] K. Huck and A. Malony, "PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing," Proc. ACM/IEEE Conference on Supercomputing (SC'05), 2005.

[7] K. Huck, A. Malony, R. Bell, L. Li, and A. Morris, "PerfDMF: Design and Implementation of a Parallel Performance Data Management Framework," Proc. ICPP 2005, IEEE Computer Society, 2005.

[8] A. Knupfer, H. Brunst, and W. Nagel, "High Performance Event Trace Visualization," Proc. PDP 2005, IEEE, 2005. http://www.vampir.eu

[9] M. Geimer, F. Wolf, B. Wylie, and B. Mohr, "Scalable Parallel Trace-Based Performance Analysis," Proc. EuroPVM/MPI 2006, LNCS 4192, Springer, pp. 303-312, 2006. http://www.scalasca.org

[10] SourceForge, "pyMPI: Putting the py in MPI", http://pympi.sourceforge.net, 2012.

[11] A. Wissink and S. Shende, "Performance Evaluation of the Multi-Language Helios Rotorcraft Simulation Software," Proc. DoD HPCMP UGC 2007 Conference, 2007.

[12] S. Shende, A. D. Malony, A. Morris, and A. Wissink, "Simplifying Memory, I/O, and Communication Performance Assessment using TAU," Proc. DoD HPCMP UGC 2010 Conference, 2010.

[13] GNU Binutils, http://www.gnu.org/software/binutils/, 2012.

[14] GDB – The GNU Project Debugger, http://www.gnu.org/software/gdb/, 2012.

[15] Backtraces – GLIBC, http://www.gnu.org/software/libc/manual/html_node/Backtraces.html#Backtraces, 2012.

[16] TotalView, http://www.roguewave.com/products/totalview-family.aspx, 2012.

[17] DDT, http://www.allinea.com/products/ddt/, 2012.

[18] DyninstAPI, http://www.dyninst.org, 2012.

[19] A. D. Malony, J. Mellor-Crummey, and S. Shende, "Measurement and Analysis of Parallel Program Performance using TAU and HPCToolkit," chapter, Performance Tuning of Scientific Applications, (eds. D. Bailey, R. Lucas, and S. Williams), CRC Press, pp. 49-86, 2010.

[20] S. Shende, A. D. Malony, and A. Wissink, "A Fault Detecting Diagnostic Tool for Python-driven Multi-language Scientific Code," in Proc. DoD HPCMP UGC Conference, IEEE Computer Society, 2012.

[21] ParaTools, Inc., "PToolsRTE - Parallel Tools Runtime Environment", http://www.paratools.com/ptoolsrte, 2012.