Efficient Parallel Runtime Bounds Checking with the TAU Performance System

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

Andrew Wissink
U.S. Army Aviation Development Directorate - AFDD
NASA Ames Research Center
Moffett Field, CA
andrew.m.wissink@us.army.mil

***Abstract*—Memory errors, such as an invalid memory access, misaligned allocation, or write to deallocated memory, are among the most difficult problems to debug because popular debugging tools do not fully support state inspection when examining failures. This is particularly true for applications written in a combination of Python, C++, C, and Fortran. We present a tool that can help identify and debug memory errors in a multi-language program at the point of failure. Integrated in the TAU Performance System®, this debugging tool allocates pages of protected memory immediately before and after dynamic memory allocations. Accessing these "guard pages" raises an error signal that causes TAU to capture performance data at the point of failure, store detailed information for each frame in the callstack, and generate a file that may be sent to the developers for analysis. The tool works on parallel programs, providing feedback about every process regardless of whether it experienced the fault, and is useful to both software developers and users experiencing memory error issues as the file output may be exchanged between the user and the development team without disclosing potentially sensitive application data. This paper describes the tool and demonstrates its application to the multi-language CREATE-AV applications Kestrel and Helios. Since those codes are export controlled, we present results from an analogous code written specifically for testing but with structure and content derived from Helios and Kestrel. The analogous performance and debugging data closely match the data obtained from the CREATE-AV codes.**

I. Introduction

Modern software often incorporates components written in different languages. These can be native languages (e.g. C, C++, Fortran), scripting or interpreted languages (e.g. Python, Java) or other high-level domain-specific languages. For instance, scientific software often uses scripting languages such as Python to drive high-level operations while applying lower-level libraries written in C, C++, and Fortran for the compute-intensive numerics. Use of this multi-language paradigm facilitates modular software and enables exchange of software between different development groups. However, it also introduces a number of debugging and memory management complexities.

Memory errors – such as an invalid array index, misaligned allocation, or read/write to deallocated memory – are among the most difficult problems to debug. These errors often manifest long after an erroneous line of code has executed, causing inexplicable corruptions and crashes that debuggers cannot unravel. Worse, the program may appear to execute successfully, leading the user to believe that a corrupt result is reliable. Memory errors are not only harder to isolate in multi-language applications, but are more likely to occur due to different array indexing schemes, type sizes, or other inter-language inconsistencies. Memory debuggers like DUMA [1], Guard Malloc [2], Electric Fence [3], and PageHeap [4] generate core files when such an error occurs, but these files contain limited debugging information about the program execution status, memory profile, process call stack, and system resources in use at the time and point of failure. Without the broader multi-language context, it can be impossible to correctly identify the cause of failure from core files alone. Valgrind's Memcheck [5] is highly effective for detecting memory errors but incurs enormous runtime overhead (30x or more), making it impractical for long-running programs. AddressSanitizer [6] is a fast memory error detector with an average slowdown of only 2x, but it is only applicable to pure C/C++ applications.

We present a *runtime bounds checking* (RBC) tool that can help identify and debug memory errors in a multi-language parallel program at the point of failure. Integrated in the TAU Performance System® [7], this debugging tool intercepts all dynamic memory allocation calls and allocates pages of protected memory immediately before and after the allocated bytes. Accessing these "guard pages" raises a signal that is handled by TAU. TAU then captures 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 performance data includes time spent in code regions, I/O statistics, and communication and synchronization statistics. The tool works on parallel programs, providing feedback about every process regardless of whether it experienced the fault. The memory errors detected include:

1) Accesses beyond the top or bottom of an array, e.g. the Nth index of an N-element array in C/C++ or the 0th index of a Fortran array,
2) Mismatched allocation/deallocation functions, e.g. allocate with `malloc` and deallocate with C++ operator `delete`,
3) Accesses to a deallocated memory block,
4) Incorrect alignment requests in routines like `posix_memalign`,
5) Memory leaks.

The guard page approach is not new [1]–[4], however this is the first example of a tool that uses this approach to

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

generate a complete performance and debugging analysis of applications written in multiple languages. Furthermore, the generated callstack profiles do not contain any of the faulty application's memory core, so it is safe to share these files with developer teams even when the faulty application is processing proprietary or sensitive inputs. The runtime overhead of this tool is only 3-4x since invalid memory accesses are checked in hardware. Compared to binary rewriting memory debuggers like Valgrind that slow an application by 30x or more, the guard page approach is highly efficient.

II. Design and Implementation

TAU is a powerful tool for program performance analysis and runtime debugging that works well with multi-language parallel programs. TAU can isolate faults by capturing the signal associated with a fault and recording the program callstack. When an error is detected, TAU manages a graceful exit of the program while generating files containing performance and callstack information for every process in the parallel application. These files are self-contained, portable, and can be analyzed without access to the executing machine or program inputs. This approach works well for numerical errors (e.g., division by zero) and was used in [8] to identify and resolve bugs in the CREATE-AV Helios [9] and Kestrel [10] project codes.

This work extends these debugging capabilities with an RBC module that records the source location associated with an invalid memory operation and memory allocation information in the application profile. In the event of an error, TAU will orchestrate a graceful application shutdown, or if the user has requested it, TAU will attempt to resume application execution at the point of failure. When the application terminates (with or without error), TAU generates detailed profiles of the program callstack and program performance for every process and thread in the application. TAU's library wrapping and preloading features make it possible to debug memory errors in dynamically linked applications without changing the application source code or recompiling.

Detecting memory errors efficiently requires runtime observation of memory accesses without modifying the source code and without incurring high overheads. The general approach is to "wrap" allocation and deallocation routines such as `malloc`, `calloc`, `free`, the Fortran keywords `ALLOCATE` and `DEALLOCATE`, and the C++ `new` and `delete` operators. TAU's RBC module achieves this via a wrapper interposition library containing customized versions of the allocation and deallocation routines. These customized versions take precedence over the system's implementation and perform the additional checks and bookkeeping required for RBC. For dynamic executables, the wrapper interposition library is dynamically preloaded at runtime. For static executables, the wrapper interposition library must be linked to the application.

The allocation functions in the wrapper library allocate a page of memory before and/or after an allocation request and use the CPU's memory management unit (MMU) to protect that page from access. If the program accesses the protected page, a memory fault traps to a signal handler that accesses the program counter and attempts to translate the PC value to a source location via debugging symbol information. TAU will also associate the violating page reference with the memory allocation request. It records the source location associated with the invalid operation and the memory allocation information in the profile file. Translating addresses of code residing in dynamic shared objects (DSOs) requires special care as address offsets are used in DSOs. TAU maintains its own address map to identify the full name of the shared object as well as the executable and the routine name. If debugging symbols have been stripped from the program, hexidecimal addresses will be listed in the profile instead of source line numbers and file names.

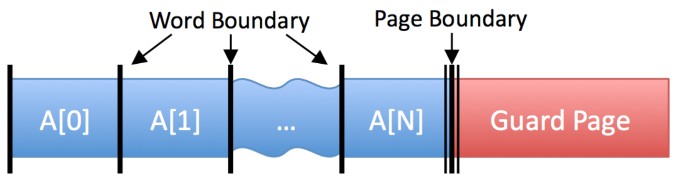


Figure 1. Allocation alignment when followed by guard page.

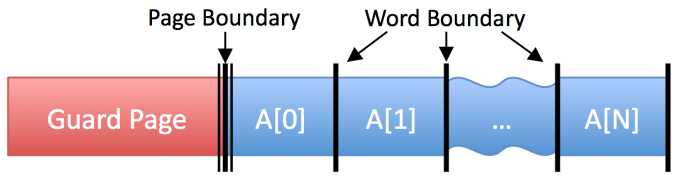


Figure 2. Allocation alignment when proceeded by guard page.

When the application experiences a memory error it is critical to diagnose the problem with respect to the executing context. Merely reporting the text output of the execution is rarely sufficient to fix the problem. The 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. This requires a full view of the execution state, particularly one that might come from a multi-language program, to gain a better insight into the program's layered workings. In addition, 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 documenting the error.

To gather this info, we use TAU's call stack capture module [8] to unwind and record the call stack of each thread of execution. Context specific information such as the calling routine name, file name, and where available, the source line number for each frame in the program's call stack are recorded in the profile. This occurs at runtime, automatically interposing with the application to generate diagnostics so that addresses of useful program information are extracted and mapped before the program terminates and that information is lost. The result is a complete picture of the application's execution at the time of failure.

A. *Memory Alignment*

The allocation functions in the wrapper library allocate a "guard page" of memory before and/or after an allocation and use the MMU to protect that page from access. If the program accesses the guard page, TAU is alerted via an error signal. This enables fast array bounds checking, but it brings hardware constraints. For example, in order to detect invalid accesses beyond the end of an array, that array must end on the

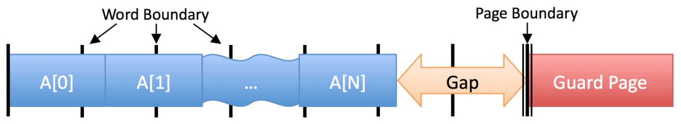


Figure 3. Gaps between the allocation and its guard pages occur when the allocation size is not a multiple of the word size.

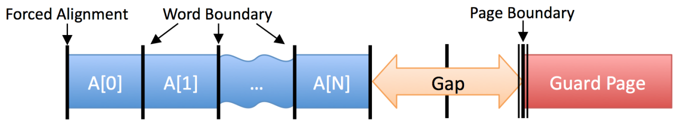


Figure 4. User-specified alignments may create gaps between the allocation and its guard pages.

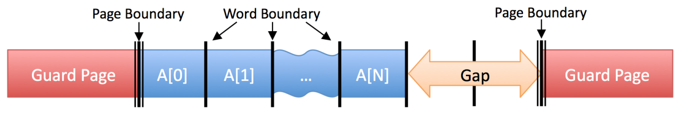


Figure 5. Gaps occur when the allocation size is not a multiple of the page size and guard pages are placed both before and after the allocation.

boundary of the guard page, which is a word-aligned address. That is, the starting address of the array must be the address of the guard page minus the array size as shown in Figure 1. Similarly, placing the guard page at the beginning of the array forces the array to begin on a page boundary as in Figure 2.

These forced alignments can create gaps between the array boundaries and the guard pages where detecting invalid accesses is expensive. Since most CPUs can only access memory on a word boundary, memory allocations must be at least word aligned and may be placed on coarser boundaries for performance reasons. Therefore, if the application makes a memory allocation using a size that is not a multiple of the word size then there is a gap between the end of the array and the guard page, as in Figure 3. Additionally, allocations made with explicit alignment specifications via functions such as `memalign` and `valloc` can cause a gap even if the array size is a multiple of the word size. This is shown in Figure 4. And finally, if both ends of the array are protected and the array size is not a multiple of the page size then there is a gap between the end of the array and the beginning of the guard page as in Figure 5.

B. Gap Protection

Protecting the gap from invalid accesses is challenging because accessing the memory in the gap will not trigger the signal handler and the gap size may be significant. Most x86 and PowerPC systems use 4K pages (SPARC systems support 8K pages) and the gap size may be as large as the page size minus one byte. Note that Linux hugepages may be up to 256M in size but enabling hugepages doesn't change the default pagesize. The same is true for Solaris large pages, so though the gap may be relatively large it will not exceed the minimum page size on the system even in the presence of large pages.

In order to maintain good runtime performance, we opted for a compromise between coverage and performance. Invalid writes are detected by dynamically checking the contents of all gaps. TAU's allocation functions fill the gap memory with a known bit pattern, which the user may specify by setting an environment variable. When a TAU event is triggered (e.g. a code region is entered or exited), TAU checks all gaps to see if the values in the gap have changed. If so, TAU records a range of line numbers in the application profile indicating where the invalid write likely originated. TAU cannot directly detect invalid reads from the gap under this scheme, but since dynamic allocations must be initialized before being read, it is likely that TAU will detect an invalid write to the gap before the invalid read occurs. Also, by choosing a sufficiently unlikely pattern, gap reads will lead to a program crash which will trigger the TAU signal handler and produce profile files containing the same data as would have been recorded if the invalid read had been detected immediately. Even if the program does not crash, a sufficiently unlikely fill value will produce program results that are clearly incorrect.

To improve TAU's coverage of the memory protection gaps, we considered a gap protection scheme similar to that used in Valgrind's Memcheck [5]. TAU has binary rewriting capabilities [11], [12] that could make this feature possible. However, we soon realized that the runtime overhead of this approach is too costly for long running applications. Memcheck dynamically checks every memory operation for correctness, which provides excellent coverage but at the cost of orders of magnitude runtime performance slowdown. Future work may pursue this approach for use with short-running applications.

C. Statically- and Dynamically-linked Applications

TAU's RBC module supports both dynamically- and statically-linked executables. Statically linked applications enable memory debugging by passing the `-optMemDbg` flag to the TAU compiler script used to compile the application. For example, a Fortran application will compile with `tau_f90.sh -optMemDbg foo.f90`. This will cause the compiler script to statically link the agent library to the application so that memory allocation and deallocatin function calls invoke the corresponding instrumented function.

Dynamically linked applications may enable memory debugging at runtime by launching the application with the `tau_exec` tool and passing the `-memory_debug` flag to `tau_exec`, or by passing `-optMemDbg` to the TAU compiler as if it were a statically linked application. Launching with `tau_exec` has the distinct advantage that the application need not be recompiled.

D. Overhead Control

Our guard page approach allocates at least two full pages of memory per allocation call, so there is the potential for severe memory overhead. To counter this, we have implemented user-adjustable thresholds for the size of the allocation and the number of allocations instrumented. These thresholds are set at runtime to allow iterative tuning of the execution parameters in the debugging environment. Users control the size of the memory allocations that should be instrumented by setting environment variables to specify a range of checked allocation sizes. If an allocation is made which falls outside that range then TAU will not allocate the additional protected pages above and below the allocation, but simply passes through to the system call. The default behavior is to assume that there are no limits on the size of the allocation. Since protecting both ends of the array is most effective when gap protection is enabled, the default behavior is to protect only the upper end of the array. Which ends of the array are protected is controlled by setting environment variables. The default behavior is to allocate guard pages after each allocation, but not before.

E. Mismatched or Invalid Allocation/Deallocation

A mismatched allocation/deallocation pair occurs when memory is allocated with one class of function but deallocated with another. For example, if memory is allocated with a call to `malloc` then it must not be deallocated with the C++ `delete[]` operator. To detect mismatch bugs, TAU records which allocation function was used to make each allocation. When memory is to be deallocated, TAU checks the compatibility of the allocation function with the called deallocation function. If the functions are incompatible, TAU records the source location of the mismatched call in the profile file.

TAU also checks the arguments of the allocate/deallocate functions for correctness. For some implementations of the POSIX library it is legal to call `malloc` with a size of zero, but this is often a bug. By default, TAU will record calls to `malloc` with zero-size in the profile file as bugs. The user may set an environment variable to disable this feature.

F. Detecting Accesses to Deallocated Memory

Once a block of memory has been deallocated then addresses in that block must not be referenced until they are reallocated. To detect invalid accesses to deallocated memory, the user may set an environment variable to a nonzero value to enable free memory protection. When this feature is enabled, deallocated memory is not returned to the free memory pool but instead is protected by the MMU so that reads or writes in the deallocated block will be detected by TAU. Guard pages for the allocation are deallocated to reclaim memory. This feature is disabled by default since it can incur high memory overhead.

III. Analysis

TAU's new runtime bounds checking features work seamlessly with TAU's traditional performance measurement and analysis workflow. To demonstrate this, we conducted a performance evaluation of the CREATE-AV codes Helios [9] and and Kestrel [10]. Helios performs high-fidelity modeling of rotorcraft aero and structural dynamics, while Kestrel is a high-fidelity, full vehicle, multi-physics analysis tool for fixed-wing aircraft. Both Helios and Kestrel use a multi-language software architecture consisting of 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++) integrated through a high-level Python-based infrastructure. Further details on the implementation and validation of Helios and details of prior work integrating TAU with Helios are covered in [9].

We executed the two CREATE-AV codes using `tau_exec` with the `-memory_debug` option on Riptide, an IBM iDataPlex at the MHPCC DSRC. We used the PToolsRTE 0.7

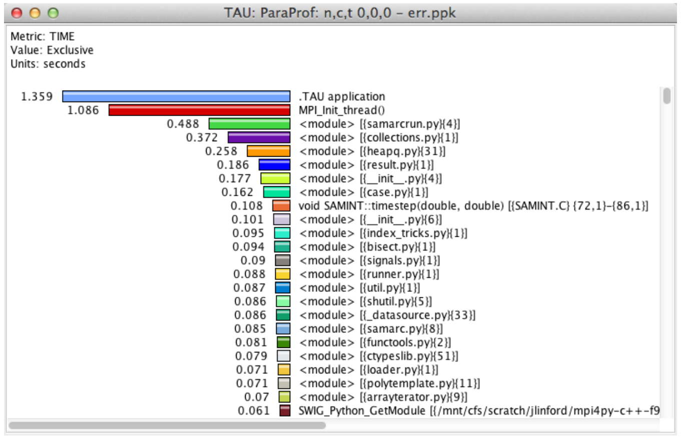
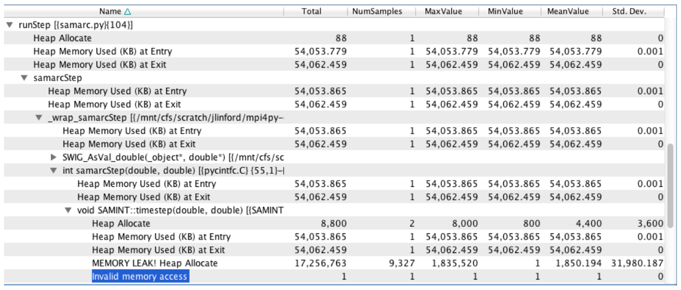


Figure 6. The application profile shows exclusive time in seconds across Python, C++, and Fortran.



Name △	Total	NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.
▼ runStep [{samarc.py}{104}]						
Heap Allocate	88	1	88	88	88	0
Heap Memory Used (KB) at Entry	54,053.779	1	54,053.779	54,053.779	54,053.779	0.001
Heap Memory Used (KB) at Exit	54,062.459	1	54,062.459	54,062.459	54,062.459	0
▼ samarcStep						
Heap Memory Used (KB) at Entry	54,053.865	1	54,053.865	54,053.865	54,053.865	0.001
Heap Memory Used (KB) at Exit	54,062.459	1	54,062.459	54,062.459	54,062.459	0
▼ _wrap_samarcStep [{/mnt/cfs/scratch/jlinford/mpi4py-						
Heap Memory Used (KB) at Entry	54,053.865	1	54,053.865	54,053.865	54,053.865	0.001
Heap Memory Used (KB) at Exit	54,062.459	1	54,062.459	54,062.459	54,062.459	0
▶ SWIG_AsVal_double(_object*, double*) [{/mnt/cfs/sc						
▼ int samarcStep(double, double) [{pycintfc.C} {55,1}-{						
Heap Memory Used (KB) at Entry	54,053.865	1	54,053.865	54,053.865	54,053.865	0.001
Heap Memory Used (KB) at Exit	54,062.459	1	54,062.459	54,062.459	54,062.459	0
▼ void SAMINT::timestep(double, double) [{SAMINT						
Heap Allocate	8,800	2	8,000	800	4,400	3,600
Heap Memory Used (KB) at Entry	54,053.865	1	54,053.865	54,053.865	54,053.865	0.001
Heap Memory Used (KB) at Exit	54,062.459	1	54,062.459	54,062.459	54,062.459	0
MEMORY LEAK! Heap Allocate	17,256,763	9,327	1,835,520	1	1,850.194	31,980.187
Invalid memory access	1	1	1	1	1	0

Figure 7. The Context Event Window in ParaProf shows the callpath leading to the invalid memory access.

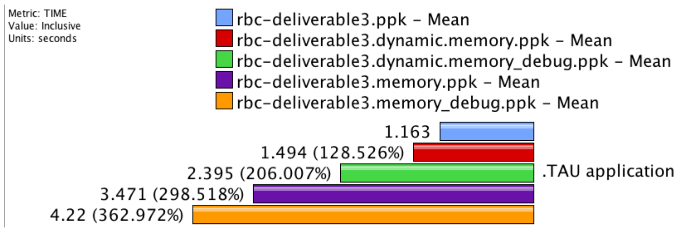
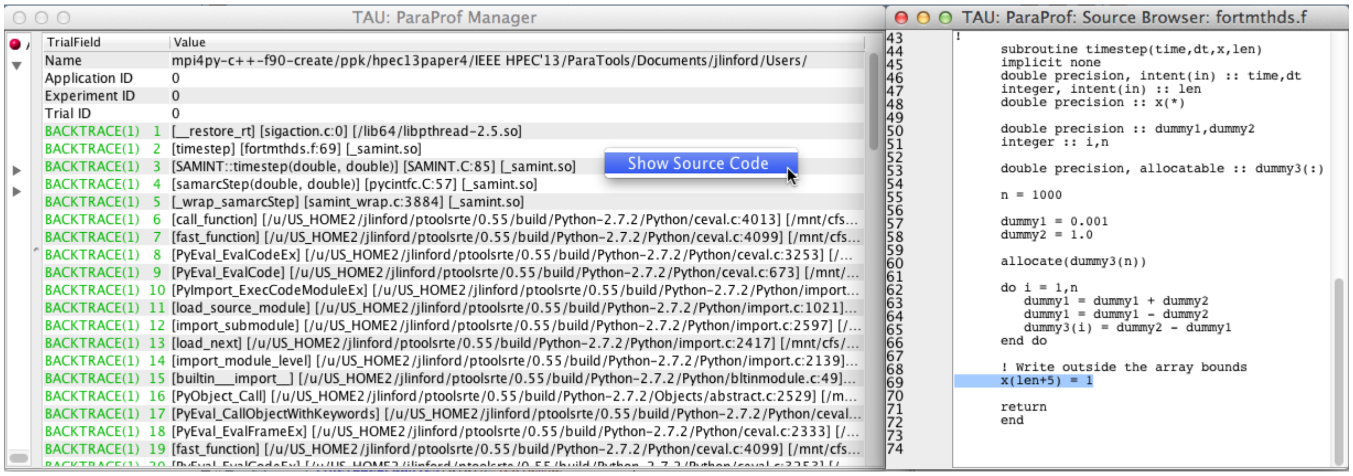


Figure 9. Application performance under various performance analysis and debugging configurations.

Python environment, which supports both pyMPI and mpi4py executions under TAU. Because Helios and Kestrel are export controlled codes who's results require special permission for public release, we instead present results from an analogous code written specifically for testing but with structure and content derived directly from Helios and Kestrel. The analogous performance and debugging data closely match the data obtained from the CREATE-AV codes.

TAU records the performance characteristics of the application in a performance profile on each application run. The profile can be analyzed with ParaProf, TAU's profile visualization tool, as shown in Figure 6. If a fault occurred during the run, ParaProf will display it in the context event window as shown in Figure 7 and record the application backtrace. The amount of dynamic memory allocated at the start and end of each frame is listed in the context event window. Figure 8 shows the application backtrace across Python, C++, C, and Fortran, including methods invoked by mpi4py and MPI.



TrialField		Value
Name		mpi4py-c++-f90-create/ppk/hpec13paper4/IEEE HPEC'13/ParaTools/Documents/jlinford/Users/
Application ID		0
Experiment ID		0
Trial ID		0
BACKTRACE(1)	1	[__restore_rt] [sigaction.c:0] [/lib64/libpthread-2.5.so]
BACKTRACE(1)	2	[timestep] [fortmthds.f:69] [_samint.so]
BACKTRACE(1)	3	[SAMINT::timestep(double, double)] [SAMINT.C:85] [_samint.so]
BACKTRACE(1)	4	[samarcStep(double, double)] [pycintfc.C:57] [_samint.so]
BACKTRACE(1)	5	[_wrap_samarcStep] [samint_wrap.c:3884] [_samint.so]
BACKTRACE(1)	6	[call_function] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/ceval.c:4013] [/mnt/cfs...
BACKTRACE(1)	7	[fast_function] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/ceval.c:4099] [/mnt/cfs...
BACKTRACE(1)	8	[PyEval_EvalCodeEx] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/ceval.c:3253] [/...
BACKTRACE(1)	9	[PyEval_EvalCode] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/ceval.c:673] [/mnt/...
BACKTRACE(1)	10	[PyImport_ExecCodeModuleEx] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/import...
BACKTRACE(1)	11	[load_source_module] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/import.c:1021]...
BACKTRACE(1)	12	[import_submodule] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/import.c:2597] [/...
BACKTRACE(1)	13	[load_next] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/import.c:2417] [/mnt/cfs/...
BACKTRACE(1)	14	[import_module_level] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/import.c:2139]...
BACKTRACE(1)	15	[builtin___import__] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/bltinmodule.c:49]...
BACKTRACE(1)	16	[PyObject_Call] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Objects/abstract.c:2529] [/m...
BACKTRACE(1)	17	[PyEval_CallObjectWithKeywords] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/ceval...
BACKTRACE(1)	18	[PyEval_EvalFrameEx] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/ceval.c:2333] [/...
BACKTRACE(1)	19	[fast_function] [/u/US_HOME2/jlinford/ptoolsrte/0.55/build/Python-2.7.2/Python/ceval.c:4099] [/mnt/cfs...

```
43  !
44        subroutine timestep(time,dt,x,len)
45        implicit none
46        double precision, intent(in) :: time,dt
47        integer, intent(in) :: len
48        double precision :: x(*)
49
50        double precision :: dummy1,dummy2
51        integer :: i,n
52
53        double precision, allocatable :: dummy3(:)
54
55        n = 1000
56
57        dummy1 = 0.001
58        dummy2 = 1.0
59
60        allocate(dummy3(n))
61
62        do i = 1,n
63           dummy1 = dummy1 + dummy2
64           dummy1 = dummy1 - dummy2
65           dummy3(i) = dummy2 - dummy1
66        end do
67
68        ! Write outside the array bounds
69        x(len+5) = 1
70
71        return
72        end
73
74
```

Figure 8. Shown in ParaProf, the application backtrace is gathered when a memory error is encountered. More than one backtrace may be recorded. The callpath from Python to C++ to Fortran90 is clearly shown. Right clicking on a backtrace entry shows the application sourcecode. The line of code that caused the memory error has been identified and highlighted.

Figure 9 shows the runtime overhead of using RBC on 128 nodes with each node executing two threads. The mean application run time is shown as exclusive time spent in the ".TAU Application" region. From top to bottom, the first (blue) bar shows run time without any instrumentation or debugging features enabled. The second bar (red) shows run time with memory leak detection enabled, but without guard pages. This incurs an average of 28.53% runtime overhead. The third bar (green) show run time with guard pages and memory leak detection, incurring approximately 106% overhead. The forth bar (purple) shows time with TAU source-based instrumentation and leak detection, and the fifth (orange) bar shows time with source-based instrumentation, guard pages, and leak detection for a maximum overhead of 263%. With all of TAU's memory debugging features enabled this application is 3.6x slower. Compared to popular binary rewriting memory debuggers that slow an application by 30x or more [5], TAU's RBC module is highly efficient.

IV. Related Work

The guard page design presented here is similar to that used in [1]–[4]. These tools use static or dynamic library interposition to intercept memory allocation and deallocation calls and allocate protected pages of memory before or after an allocation. When a guard page is accessed, the tool dumps the program core. Core files are unwieldy and contain limited information about the program execution status, memory profile, process call stack, and system resources in use at the time and point of failure. Without the broader multi-language context, it can be impossible to correctly identify the cause of failure from core files alone.

Valgrind's Memcheck [5] is a dynamic analysis tool that verifies every memory read or write at runtime. This approach is highly effective for detecting memory errors but incurs enormous runtime overhead, making it impractical for long-running programs. Programs running with Memcheck are often 30-50x slower, but the diagnostic output from Memcheck is more informative than a simple core file.

Tools like AddressSanitizer [6] and Dr. Memory [13] use shadow memory methods to isolate memory bugs. These methods have proven highly effective with AddressSanitizer achieving good coverage with only 2-3x runtime overhead. Dr. Memory incurs 10-15x runtime overhead on average. Shadow memory methods require binary instrumentation with the best performance coming from customized compilers. Therefore these methods are difficult to apply to multi-language applications involving both interpreted and compiled languages like Python and Fortran.

Debugging mixed language parallel programs that use Python is a daunting task. Commercial debuggers such as TotalView [14], DDT [15], and open source debuggers such as gdb [16] 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 TAU [7], DyninstAPI [11], and HPCToolkit [17] projects.

V. Significance to the 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. Current solutions are inadequate, particularly for multi-language applications that use a mix of Python, Fortran, C, and C++. 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 techniques for comprehensive measurement in the presence of memory 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 understanding the computational characteristics of an application and for planning computing requirements.

The new runtime bounds checking 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 the TAU Performance System 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:

- Does the application exhibit memory access violations?
- When and where do these memory violations occur?
- What is the heap memory usage in the application?
- What is the nature (read/write) of the violation and what is its source location?
- Which processes or threads experienced an error?
- 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 prior to the fault?

VI. Conclusion

Debugging memory errors in parallel, multi-language programs is a daunting task. Exceptional errors can lead to program failures with little information to deduce causes. More pernicious memory errors lead to wrong program results. We have presented a runtime bounds checking tool, developed inside the TAU Performance System®, that efficiently detects and records many common memory errors including out-of-bounds memory references made by an application. This is achieved via a scheme of runtime replacement of memory calls with versions that perform bounds testing with relatively low CPU overhead. The tool generates valuable diagnostic information to help debug why the program crashed and highlight the first location of the invalid memory access. It shows the source location associated with each read or write access violation together with other profile data such as the heap memory usage and the extent of memory allocation and deallocation. The tool works well in parallel and multi-threaded programs and was successfully applied to the CREATE-AV Kestrel and Helios projects to check potential memory access violations. It is freely available in the TAU Performance System and licensed under a BSD style license.

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 Air Vehicles Element of the Computational Research and Engineering for Acquisition Tools and Environments (CREATE-AV) Program sponsored by the HPCMP.

References

[1] H. Aygün, "D.U.M.A. - Detect Unintended Memory Access," http://duma.sourceforge.net/, May 2013.

[2] Apple Corp., "Guard malloc manual page," https://developer.apple.com, March 2009.

[3] B. Perens, "efence – electric fence malloc debugger," http://linux.die.net/man/3/efence, 1999.

[4] Microsoft Corp., "Gflags and pageheap," http://msdn.microsoft.com, July 2013.

[5] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, CA, USA, April 2005.

[6] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *USENIX ATC 2012*, 2012.

[7] S. Shende and A. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[8] S. Shende, A. Malony, J. Linford, A. Wissink, and S. Adamec, "Isolating runtime faults with callstack debugging using TAU," *Proc. HPEC 2012 Conference*, 2012.

[9] A. Wissink and S. Shende, "Performance evaluation of the multi-language Helios rotorcraft simulation software," in *Proceedings of the DoD High Performance Computing Modernization Program Users Group Conference*. IEEE Computer Society, 2007.

[10] S. A. Morton, B. Tillman, D. R. McDaniel, D. R. Sears, and T. Tuckey, "Kestrel – a fixed wing virtual aircaft product of the CREATE program," in *Proceedings of the DoD High Performance Computing Modernization Program Users Group Conference*. IEEE Computer Society, 2009.

[11] A. R. Bernat, K. Roundy, and B. P. Miller, "Efficient, sensitivity resistant binary instrumentation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Toronto Canada, July 2011.

[12] D. Barthou, A. Charif Rubial, W. Jalby, S. Koliai, and C. Valensi, "Performance tuning of x86 OpenMP codes with MAQAO," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Springer Berlin Heidelberg, 2010, pp. 95–113.

[13] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, 2011, pp. 213–223.

[14] "Memory debugging challenges: How to identify and resolve memory bugs in parallel and distributed applications." Rouge Wave Software, 5500 Flatiron Parkway, Suite 200, Boulder, CO 80301, USA, White Paper, January 2013.

[15] M. O'Connor, "Parallel debugging is easy," Allinea Software, White Paper, 2008.

[16] "GDB: The GNU project debugger," http://www.gnu.org/software/gdb/, April 2013.

[17] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs." *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.