Profiling Production OpenSHMEM Applications

John C. Linford[1(✉)], Samuel Khuvis[1], Sameer Shende[1], Allen Malony[1], Neena Imam[2], and Manjunath Gorentla Venkata[2]

[1] ParaTools, Inc., 2836 Kincaid St., Eugene, OR 97405, USA
{jlinford,skhuvis,sameer,malony}@paratools.com
http://www.paratools.com/

[2] Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN 37831, USA
{imamn,manjugv}@ornl.gov
http://ut-battelle.org/

Abstract. Developing high performance OpenSHMEM applications routinely involves gaining a deeper understanding of software execution, yet there are numerous hurdles to gathering performance metrics in a production environment. Most OpenSHMEM performance profilers rely on the PSHMEM interface but PSHMEM is an optional and often unavailable feature. We present a tool that generates direct measurement performance profiles of OpenSHMEM applications even when PSHMEM is unavailable. The tool operates on dynamically linked and statically linked application binaries, does not require debugging symbols, and functions regardless of compiler optimization level. Integrated in the TAU Performance System, the tool uses automatically-generated wrapper libraries that intercept OpenSHMEM API calls to gather performance metrics with minimal overhead. Dynamically linked applications may use the tool without modifying the application binary in any way.

Keywords: Profiling · Tracing · Performance analysis · The TAU Performance System · Code generation · Library wrapping

1 Introduction

OpenSHMEM application performance can be characterized via profiling and tracing tools built on the PSHMEM interface. For every routine in the OpenSHMEM standard, PSHMEM provides an analogous routine with a slightly different name. This allows profiling tools to intercept and measure OpenSHMEM calls made by a user's application by defining routines with the same function signatures as OpenSHMEM routines – *wrapper functions* – which call the appropriate PSHMEM routines. For example, the TAU Performance System® [7] provides an OpenSHMEM wrapper library which can be linked to any OpenSHMEM application to acquire runtime measurements of OpenSHMEM routines. The library can be used with statically or dynamically linked applications with runtime overhead between 1.5% and 4% [4]. Regardless of which events are recorded, TAU's overhead is approximately O(1) in the number of application processes, i.e. as the

M. Gorentla Venkata et al. (Eds.): OpenSHMEM 2016, LNCS 10007, pp. 219–224, 2016.
DOI: 10.1007/978-3-319-50995-2_15

number of SHMEM processing elements (PEs) increases the overhead incurred by TAU remains relatively constant. This makes TAU an appropriate choice for profiling large-scale OpenSHMEM applications when PSHMEM is available.

For reasons of practicality, applications are typically developed on small-scale representative systems before being deployed on large-scale production systems. Yet it is often the case that performance bugs – software faults that affect the application's performance but not correctness – present themselves only at scale. Metrics such as time spent in code regions, compute intensity, message size, and communication volume are especially difficult to discern in production environments or at large scale. A production system may use highly optimized runtime libraries where performance tool interfaces (i.e. PSHMEM) have been disabled, rendering profiling and tracing tools like TAU ineffective. Tools that do not rely on PSHMEM but instead periodically sample the application (e.g. HPCToolkit [1,5]) cannot resolve this problem due to their inability to capture atomic events (e.g. the size, sender, and receiver of a message or the size of a memory allocation) and their reliance on debugging symbols, which are often stripped from production binaries. In short, OpenSHMEM developers would like to characterize the performance of production applications operating at large scales without modifying the application or relying on debugging symbols or special tools interfaces like PSHMEM.

This work-in-progress paper presents a tool that generates direct measurement (i.e. not sampled) performance profiles and traces of OpenSHMEM applications when PSHMEM is unavailable. The tool extends the existing OpenSHMEM profiling capabilities in TAU and therefore has similar runtime overhead (less than 4%). By building on TAU, we receive the full benefit of TAU's measurement layer so there are no restrictions to the types of performance data that can be gathered, i.e. PAPI can be used to gather hardware performance counters without any caveats. As detailed in Sect. 2, the tool parses the OpenSHMEM header files and automatically generates source code for wrapper libraries that intercept OpenSHMEM API calls at link-time or at run-time so that both dynamically and statically linked applications can be profiled. Since the tool uses source code parsing and code generation, it does not require debugging symbols and functions regardless of compiler optimization level.

2 Approach

Our goal is to provide performance data without relying on any special features of a particular OpenSHMEM implementation, i.e. PSHMEM. At a high level this involves two steps: constructing functionality similar to what is provided by the PSHMEM interface and making it available to the application.

2.1 Symbol Wrapping

For every routine in the OpenSHMEM standard, PSHMEM provides an analogous routine with a slightly different name. We use *symbol wrapping* via the

program linker to do the same. Nearly all program linkers support a `-wrap foosym` command line option to enable wrapping of the symbol `foosym`. Any undefined reference to `foosym` will be resolved to `__wrap_foosym` and any undefined reference to `__real_foosym` will be resolved to `foosym`. In this case, we use symbol wrapping to provide a unique wrapper function for each API function defined in an OpenSHMEM implementation's header files. When the application's object files are linked to form the executable file, a `-wrap` flag for every OpenSHMEM API call is passed to the linker via the special `@argfile` syntax supported by most linkers.

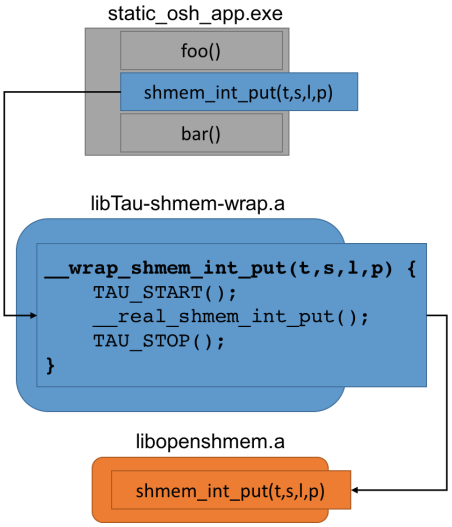

Fig. 1. Symbol wrapping via the program linker replacing a call to `shmem_int_put` with a wrapper function at link time. The wrapper function uses TAU to record performance data and invokes the original `shmem_int_put`.

Figure 1 demonstrates symbol wrapping with an OpenSHMEM application that is statically linked against the OpenSHMEM implementation library `libopenshmem.a`. At link time, the call to `shmem_int_put` in the application is replaced with a call to `__wrap_shmem_int_put`, which is implemented in the `libTau-shmem-wrap.a` wrapper library. The wrapper function uses TAU to record performance data and invokes `__real_shmem_int_put`, which the linker replaces with a call to the original `shmem_int_put` as defined in `libopenshmem.a`.

Symbol wrapping works equally well for statically linked applications and dynamically linked applications that statically link against the OpenSHMEM implementation. However, applications that link dynamically against `libopenshmem.so` should use library preloading instead of symbol wrapping because symbol wrapping will only intercept SHMEM calls made from the application itself.

2.2 Library Preloading

Symbol wrapping is a powerful, low overhead way to wrap the OpenSHMEM API, but it requires the user to re-link their application against a special library of wrapper functions. This is not always possible in a production environment, so we use *library preloading* to achieve dynamically what the linker does statically. The `LD_PRELOAD` environment variable specifies a list of additional shared libraries to be loaded before all others, selectively overriding functions in other shared libraries. We use the `LD_PRELOAD` environment variable to insert a *dynamic symbol wrapper* at the front of the search list. The dynamic symbol wrapper will resolve any undefined reference to `foosym` to `__wrap_foosym` and any undefined reference to `__real_foosym` to `foosym`, just as the linker does statically when passed the `-wrap` command line option. This requires the application to be dynamically linked against the OpenSHMEM implementation library.

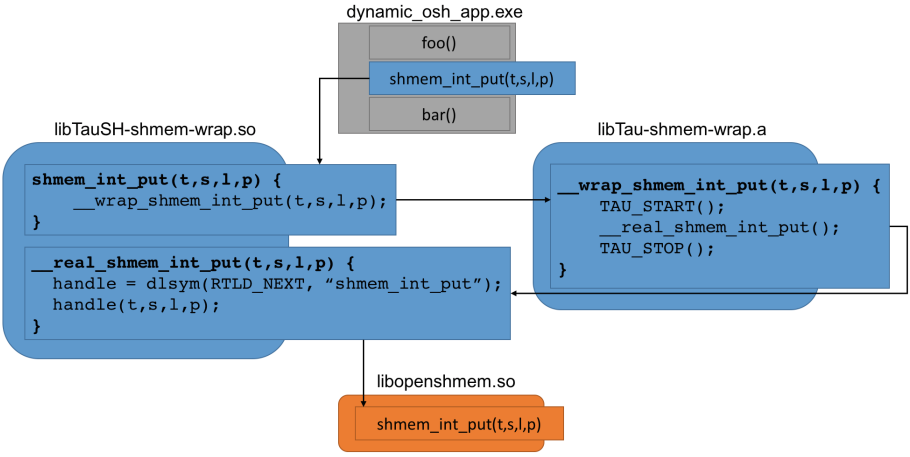

Fig. 2. Using a dynamic symbol wrapper to dynamically resolve undefined references to `shmem_int_put` to `__wrap_shmem_int_put` and undefined references to `__real_shmem_int_put` to `shmem_int_put`.

Figure 2 shows how the dynamic symbol wrapper library achieves symbol wrapping at runtime when `libTauSH-shmem-wrap.so` is prepended to the `LD_PRELOAD` environment variable. Because the dynamic symbol wrapper is the first library on the search list, the call to `shmem_int_put` in the application resolves to the definition of `shmem_int_put` provided by the dynamic symbol wrapper. This implementation simply passes control to the `__wrap_shmem_int_put` function defined in our wrapper library. When the wrapper library invokes `__real_shmem_int_put`, that symbol resolves dynamically to the implementation provided in `libTauSH-shmem-wrap.so`. The dynamic linker's programing interface is then used to discover the address of the original implementation of `shmem_int_put` as defined in `libopenshmem.so`.

2.3 Automatic Wrapper Library Generation

In order to construct a tools interface for an arbitrary SHMEM implementation, we use the Program Database Toolkit (PDT) [3,6] to parse the implementation's header files (e.g. `shmem.h` and `shmemx.h`) and discover the available API. For each API function parsed, a wrapper function is automatically generated that tracks the performance characteristics of that routine, e.g. wall clock time. If the routine also sends or receives data (e.g. `shmem_int_put`) then the wrapper also tracks the message size, target PE, and source PE. The wrapper functions can also measure hardware performance counters via PAPI [2] to track cache misses, operation counts, etc. For example, the application profile will show if a call to `shmem_barrier` used busy-wait.

3 Conclusions and Future Work

We present a tool that generates direct measurement performance profiles of OpenSHMEM applications even when PSHMEM is unavailable. The tool operates on dynamically linked and statically linked application binaries, does not require debugging symbols, and functions regardless of compiler optimization level. This work completely removes the need for a PSHMEM interface with no significant disadvantage to the user, however PSHMEM is still valuable to tools other than TAU which cannot automatically generate wrapper libraries.

Many OpenSHMEM implementations – most notably OpenSHMEM reference implementation 1.2 – do not provide the implementation library in both static and dynamic forms by default. Only the static library, `libopenshmem.a`, is built by default. Performance tools that use this approach would benefit from having both the static and dynamic libraries available by default as it would fully enable the library wrapping features we have described. Without a dynamic library, only link-time wrapping is possible.

TAU could also benefit from an interface which exposes synchronization of the symmetric heap. At present, TAU intercepts the underlying system allocation and deallocation calls and OpenSHMEM library calls to mark operations on the symmetric heap. However, it is difficult to observe in a trace when an update to the symmetric heap becomes visible to other PEs. TAU could make use of a mechanism for notifying a performance measurement system of symmetric heap updates when they occur to improve the quality of the performance data.

Acknowledgments. This work was supported by the United States Department of Defense (DoD) and used resources of the Computational Research and Development Programs and the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurrency Comput. Pract. Exp. **22**(6), 685–701 (2010)

2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **3**(14), 189–204 (2000)
3. Lindlan, K., Cuny, J., Malony, A., Shende, S., Mohr, B., Rivenburgh, R.: A tool framework for static and dynamic analysis of object oriented software with templates. In: SC 2000: High Performance Networking and Computing Conference (2000). http://www.cs.uoregon.edu/research/pdt
4. Linford, J., Simon, T.A., Shende, S., Malony, A.D.: Profiling non-numeric OpenSHMEM applications with the TAU performance system. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 105–119. Springer, Heidelberg (2014). doi:10.1007/978-3-319-05215-1_8
5. Malony, A., Mellor-Crummey, J., Shende, S.: Methods and strategies for parallel performance measurement and analysis: experiences with TAU and HPCToolkit. In: Bailey, D., Lucas, R., Williams, S. (eds.) Performance Tuning of Scientific Applications. CRC Press, New York (2010)
6. Quinlan, D.: ROSE: compiler support for object-oriented frameworks. In: Proceedings of Conference on Parallel Compilers (CPC 2000), Aussois, France, January 2000
7. Shende, S., Malony, A.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)