Dynamic Performance Callstack Sampling: Merging TAU and DAQV

Sameer Shende, Allen D. Malony, and Steven T. Hackstadt

Computational Science Institute, Department of Computer and Information Science, University of Oregon, Eugene OR 97403
{sameer, malony, hacks}@cs.uoregon.edu

Abstract. Observing the performance of an application at runtime requires economy in what performance data is measured and accessed, and flexibility in changing the focus of performance interest. This paper describes the performance callstack as an efficient performance view of a running program which can be retrieved and controlled by external analysis tools. The performance measurement support is provided by the TAU profiling library whereas tool-program interaction support is available through the DAQV framework. How these systems are merged to provide dynamic performance callstack sampling is discussed.

1 Introduction

The are several motivations for wanting to observe the performance of a parallel application during its execution [8] (e.g., to terminate a long running job or to steer performance variables). The downside in doing so is the deleterious effect on performance that may result. This trade-off forces consideration of a means to capture just enough performance information to make possible essential performance "views." Additionally, there is the issue of how external performance analysis tools access the performance data resident in the program's execution state. The overhead of data transfer is a pragmatic problem (more data takes longer to send), but flexible control, via a well-defined external interface, of what data to send and even what level of instrumentation to enable may allow certain performance measurement decisions to be runtime selectable.

In this paper, we describe the dynamic performance callstack tool that we are developing as part of the TAU profiling package [1] for parallel, multi-threaded C++ programs. We also discuss how the performance callstack information is made available to analysis and visualization tools running in the program's computational environment. This is done using the DAQV-II tool interaction framework [5].

2 Performance Callstacks

A *callstack* at a point in execution shows the current execution location(s) and the sequence of procedure calls that led to it [4]. A *performance callstack* is a view on a program's performance execution profile at runtime. The execution profile shows where time is being spent in a program's code (mainly with respect to routines) for each thread of execution. A *performance callstack profile* at a point in time is defined as the profile of the application with respect to the active profiled blocks on the callstack if the application had terminated at that time.

Selective profiling allows a user to profile a group of profiled blocks. A profiled block is said to be active if its instrumentation is enabled at runtime. If a profiled block is not active, it does not appear on the callstack and no statistics are maintained for it. A profiled block can be a routine (function) or a user-defined statement-level timer that has both inclusive and exclusive profiled quantities associated with it. A profiled quantity could be time or the value of a hardware performance counter such as the number of secondary data cache misses.

The performance callstack view is defined only for those routines in the calling stack of a thread at the point where the callstack is sampled. Looking at a single sample, the performance callstack shows, for each profiled block in the calling stack, its current execution profile statistics. These statistics include the number of invocations or calls, the number of profiled subroutines called by it, the aggregate exclusive and inclusive time spent in the routine, and the instance exclusive and inclusive time spent in the routine since the start of each instance of the profiled block on the callstack.

To illustrate how these are calculated, we look at an example shown below. In this pseudocode, routine entry and exit times (with respect to the wallclock time) are marked by open and close braces, respectively. A slice of the program's execution profile is taken when the code indicated by the TAU_MONITOR() routine is encountered. At this instance, the performance callstack for that thread of execution is calculated and the performance metrics calculated represent the execution profile, had the application terminated at that instant. Table 1 shows how the performance callstack statistics described in this section are calculated.

Example program showing routine extry and exits w.r.t time

```
ROUTINE                          TIME (usec)
main() {                              0
       foo() {                        5
             }                       10

       bar() {                       15
             foo() {                 20
                 TAU_MONITOR()       25
```

In this way, callstack profiling helps the user understand the performance profile metrics at a single point in time for only those routines that are active and on the callstack. Multiple samples taken over time can be displayed in the form of a callstack trace history showing how performance behavior changes. Depending on where the callstack is sampled, performance views for different subsets of a program's routines can result.

Table 1. Callstack statistics for the example

Routine on Callstack	Calls	Subrs	Excl (usec)	Incl (usec)	Instance Excl (usec)	Instance Incl (usec)
main()	1	2	5	25	5	25
bar()	1	1	5	10	5	10
foo()	2	0	10	10	5	5

3 TAU Portable Profiling Library

The TAU portable profiling library [12] is used to build the performance callstack view. The library features the ability to capture performance data for C++ function, method, basic block, and statement execution, as well as template instantiation. It also supports the definition of profiling groups for organizing and controlling instrumentation. The performance callstack contains the TAU profiling data for those functions in the calling stack. From the profiling data collected, TAU's profile analysis procedures can then generate a wealth of performance information for the user. It can show the exclusive and inclusive time spent in each function with nanosecond resolution. For templated entities, it shows the breakup of time spent for each instantiation. Other data includes the number of times each function was called, the number of profiled functions each function invoked, and the mean inclusive time per call. Time information can also be displayed relative to nodes, contexts, and threads [3]. All of this analysis is also available to the user of the performance callstack view.

4 Runtime Access to Performance Callstack View

The performance callstack information provides a snapshot on a program's performance during execution. One common approach to making performance data available to tools is to save it in a trace file for post-execution analysis [9, 11]. Because multiple nodes of an application produce callstack data, multiple trace files are needed and must be merged to give a complete, time-consistent view of the application's performance. This approach can be easily extended to provide access to the callstack data while the application is executing if the trace files can be shared with the analysis tools. However, this complicates tool design, and if the application processes are running on distributed machines, a shared file approach may not be possible.

Another alternative is to build into the application a runtime interface that allows external tools to access callstack data over a network. Unfortunately, instead of multiple files, tools must deal with multiple network connections to application nodes, and the application must manage these connections and coordinate the callstack data access. However, the interaction between the application and the tools is more tightly coupled in this approach. In addition to the communications functionality, there are issues concerning application-tool synchronization, callstack data consistency, and performance perturbation. Idealy, a solution would not impact the performance of the application significantly beyond the cost of file I/O, yet would allow callstack data to be accessed in a distributed environment with a simple interface for tools.

Because analysis tools work with the callstack data as a single, unified sample, it is desirable to provide a data access interface that obviates concern for where the individual parts of the callstack (node, context, and thread parts) are located in the application's execution environment. Similarly, the application should not be bothered by the number and location of tools, merely informing the external interface of where the callstack data is located and when it is available. The "glue" between the tool and application interfaces then must implement the mapping of a high-level, callstack data view to its individual parts and runtime location while servicing callstack access requests from multiple analysis tools. We decided to use the DAQV framework to implement the interfaces and glue for runtime access to performance callstack views. The DAQV system is described below followed by its integration with TAU and use for callstack access.

5 The DAQV Program Interaction Framework

The DAQV-II [5] framework for program interoperability provides external tools with a view of distributed data as a logical global array that can be accessed selectively via a high-level array reference; see Figure 1.

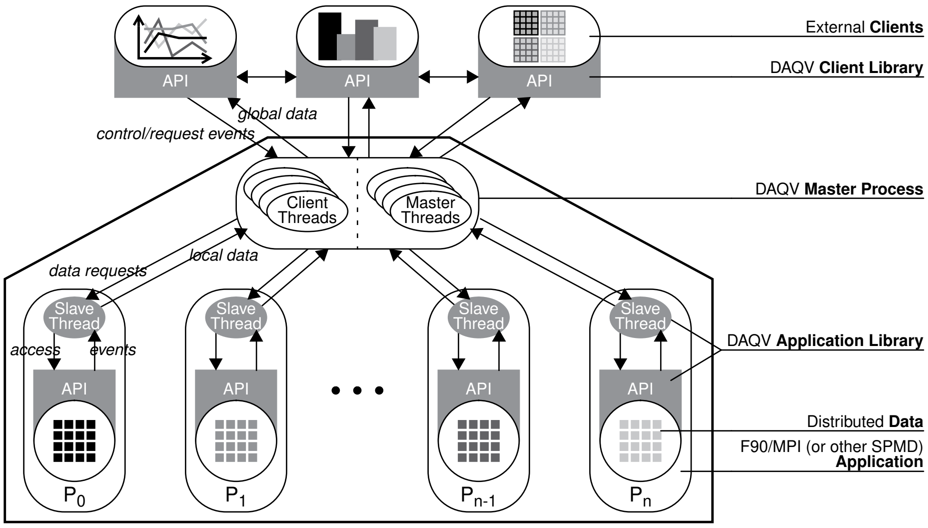

Fig. 1. DAQV-II Framework

Each application process is linked with a DAQV library that provides a simple procedural interface allowing the programmer to describe how data is distributed and to indicate places in the code where it may be accessed. The library creates a *slave* thread that shadows execution of the application process. The purpose of these threads is to maintain information about available distributed data and to perform accesses to that data when requested. The reason for embodying this functionality in the form of a thread is so that querying the information and accessing the data can be done concurrently with the execution of the application processes, if desired. DAQV-II uses the threading system provided by the Nexus multithreaded communication library [2] to create and manage these slave threads.

The individual slave threads are coordinated by a separate process called the DAQV *master*. The master process is also responsible for interacting with the other tools and applications in the DAQV environment by acting as the "single point of contact" for the entire set of application processes. The master process spawns additional threads to handle the various requests it receives; because it also uses Nexus, the master process is fully multithreaded and can take advantage of multiple processors, if available.

External tools, also known as DAQV *clients*, use a client interface that implements a high-level array access model. From the point of view of a tool, the interface gives access to arrays registered by the application. The client interface runs as a separate thread in the tool process and is responsible for communicating requests to and receiving events from the master, and participating in data communication. A detailed description of the operation of array access can be found in [7].

DAQV-II was originally designed to allow external tools to access distributed application data. For example, a visualization tool could use DAQV-II to access arbitrary subsections of the global array created by each of the declared program array instances in the multiple processes of a parallel program. Each process registers its portion of the global array with DAQV; the DAQV master process coordinates these registrations and interprets them in a global context. At points indicated in the program source code, interactions with external tools (via the DAQV master) are allowed. It is the reponsibility of the application programmer to ensure that access is provided at scientifically and semantically meaningful points in the code.

DAQV-II was designed to support both synchronous and asynchronous (with respect to application execution) data access. Synchronous data access is fairly straightforward and uses a debugger-like model whereby application execution is suspended while data access occurs. Asynchronous access, however, requires a separate thread of control to carry out data access while the application continues execution. Our initial implementation provides synchronous access to program data because accessing this type of data asynchronously is prone to consistency problems (i.e., an executing application is free to modify its program data without concern for what other threads may be doing).

However, accessing program performance data that is being collected and stored by a separate profiling library demands a low-impact, performance-efficient monitoring technique if that data is to be made available to external tools. The profiling library goes to great lengths to minimize perturbation; requesting and transporting that data to an external tool should, too.

6 Integration of TAU and DAQV

When considering techniques for online monitoring, there are several options available. In a traditional client/server model, interactions are typically synchronous between the requester of the data (the client) and the source of the data (the server). Excessive synchronization is likely to deteriotate application performance. In addition, monitoring performance data in an application with multiple threads/processes forces clients to manage interaction with, and data from, multiple servers. This makes client implementation unnecessarily complex.

Another option exists in our first implementation of the DAQV system [6]. Our primary objective was to simplify tool interaction with parallel applications by removing the need to interact with each process individually. That is, client tools could view a parallel application as a single entity and make logical requests for data from the global array structure defined by the collection of individual program arrays. This greatly simplified client development. We supported two modes of operation, *push* and *pull*. Under the push model, data was automatically delivered to the appropriate display tool as the program executed, according to routines inserted into the source code. This approach was very simple to use but not very flexible. To support more interactive array access, we implemented a pull model which allowed rudimentary control over program execution and runtime selection of the arrays to be visualized. However, both modes of operation required that the application program execution be suspended (by calling a DAQV routine). Thus, the perturbation caused by synchronization was still substantial.

In our most recent version, DAQV-II, we follow a similar abstract model that allows clients to view parallel applications as a single entity. But we also address the limitations of synchronous data access by using the metaphors of *probe* and *mutate*, which allows DAQV-II to support both synchronous and asynchronous data access. As mentioned above, for scenarios involving declared program data, we have, thus far, adhered to a synchronous approach. But realizing the potential for DAQV-like functionality beyond just accessing distributed program arrays (e.g., for accessing performance data or program monitoring), supporting asynchronous access was important. In this model, program execution may continue while a separate thread of control reads or writes the data of interest. The probe/mutate model removes the notion of synchronization from data access; it simply indicates the type (read/write) of access being performed.

The model supported by DAQV-II is particularly suitable for callstack monitoring. First, it allows asynchronous access and minimizes the synchronization overhead experienced by the application. Second, it supports a simple abstraction for interacting with parallel applications and eases the tool development process. Third, DAQV allows multiple client tools to access the "global" performance callstack simultaneously.

Adapting DAQV-II for use with TAU required only minor extensions. A routine for registering TAU performance data with DAQV was added, as was a new data distribution type to support TAU's performance callstack data. Nexus remote service request handlers were added to support the asynchronous data collection and transport. Client requests for data were able to use the existing application programming interface supported by the DAQV client library.

We use the DAQV-II framework to access the performance callstack and deliver the profile data to external analysis tools. Figure 2 shows how we have merged the DAQV-II framework with the TAU performance callstack measurements. Here, the performance callstacks for the parallel threads are distributed across the processing nodes in the parallel execution.

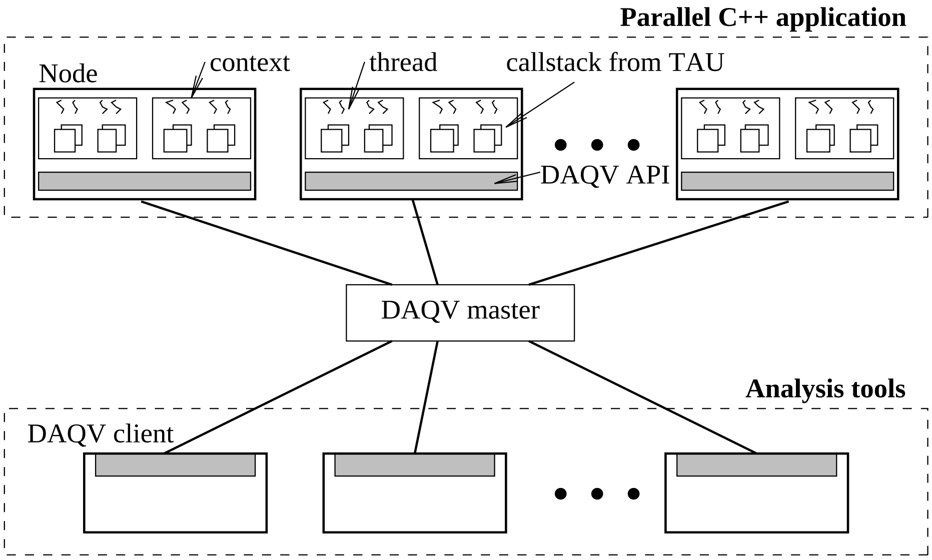

Fig. 2. TAU-DAQV integration

DAQV-II allows this distributed callstack data to be described as a single global callstack array that can be requested by clients. The callstack data is collected in each thread when the TAU_MONITOR() routine is executed. This callstack data snapshot is then registered with the DAQV-II system. The registration process informs DAQV of the location and size of the data so that it may fill subsequent requests for it.

Figure 3 depicts a high-level view of the DAQV protocol as used with TAU. Clients attach to the DAQV master process and receive information about registered data. Later, clients send data requests to the master process, which forwards the requests to each of the DAQV slave threads. These threads send the registered callstack data to the master, which collects responses from all nodes and then forwards the global callstack to the client that requested it. The data is accessed asynchronously. DAQV need only ensure that the location and size of the registered callstack data is not changed while it is being accessed. This requires a small amount of locking to occur between the TAU_MONITOR() routine and the data access handler.

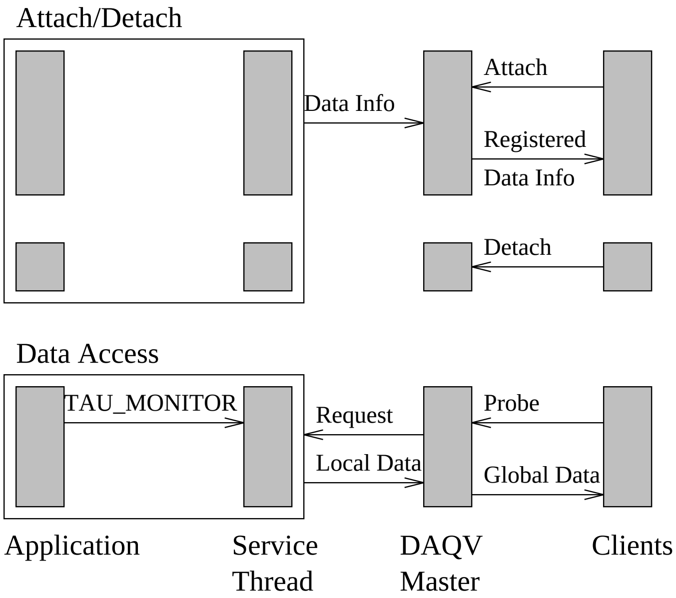

Fig. 3. DAQV protocol

Thus, the parallel program does not need to perform a barrier operation, and it can continue to execute with minimal intrusion. The synchronization operations are off-loaded to the DAQV slave threads, which are responsible for communication with the master process; this further reduces the intrusion in the parallel program.

DAQV-II allows multiple clients to get callstack data simultaneously from the running program. Each client could perform a different analysis on the callstack data, present different views of the data, or implement different sampling intervals. For scientists who are geographically separated, DAQV-II facilitates collaborative monitoring by allowing them to attach, monitor the execution, and detach. The intrusion that accessing callstack data introduces in the parallel program is independent of the number of client monitors attached to the master.

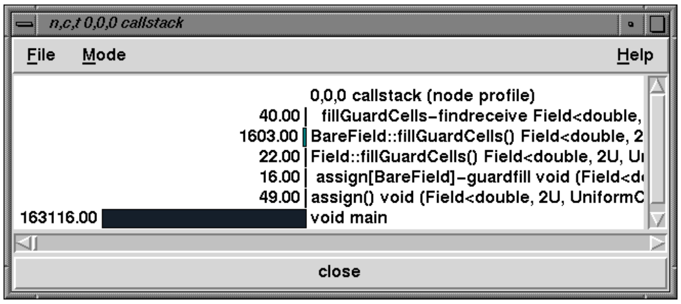

Fig. 4. Callstack on node 0 of a POOMA 2D Diffusion equation simulation

Figure 4 shows the callstack view of a two dimensional diffusion equation which tracks the progression of the diffusion of a heat source on a mesh with respect to time. It was implemented using the POOMA [10] object oriented scientific computing framework.

7 Conclusions

The merging of the TAU and DAQV-II systems described above has been implemented for performance callstack sampling. Callstack analysis and visualization tools have also been constructed. The TAU portable profiling library captures performance data in parallel C++ programs for functions, methods, basic blocks, statements, and template instatiations. The TAU performance callstack encapsulates the collected profiling data for all the functions currently in the calling stack. Access to this information at runtime can yield a wide range of useful performance information to the user.

However, accessing this data at runtime requires consideration of data consistency, synchronization, application perturbation, ease of use, and client development. DAQV-II provides an interoperability and data exchange infrastructure appropriate for program and performance monitoring. Asynchronous data access minimizes synchronization overhead and application perturbation. DAQV's simple abstraction for application interaction simplifies the development of client tools and facilitates their access to performance callstack data.

We feel that the performance callstack contains performance information that is critical to performance decisions made while an application is executing. The system we have built by merging the TAU portable profiling library with the DAQV-II interaction framework facilitates efficient and convenient access to this information by external tools.

Acknowledgments

This work was supported in part by the Department of Energy ASCI program (Contract No. C70660017-3) and DOE 2000 program (Agreement No. DEFC

0398 ER 259 986). We would like to thank the Los Alamos National Laboratory for their sponsorship of TAU and DAQV projects. The authors sincerely acknowledge the important technical feedback provided by Steve Karmesin and Pete Beckman of LANL as well as the invaluable contributions of Ariya Lohavanichbutr, Chad Busche, and Michael Kaufman of the Department of Computer and Information Science, University of Oregon, on the implementation of the system.

References

1. Advanced Computing Laboratory (LANL): TAU Portable Profiling URL:http://www.acl.lanl.gov/tau. (1998)
2. Foster, I., Kesselman, C., Tuecke, S.: The Nexus Approach to Integrating Multithreading and Communication. Jour. of Parallel and Distributed Computing. Vol. 37 (1). Aug (1996) 70–82
3. Gannon, D., Beckman, P., Johnson, E., Green, T., Levine, M.: HPC++ and the HPC++LIB Toolkit. Technical Report Department of Computer Science, Indiana University (1998)
4. High Performance Debugging Forum: HPD Version 1 Standard: Command Interface for Parallel Debuggers. High Performance Debugging Forum and Oregon State University (1997)
5. Hackstadt, S., Harrop, C., Malony, A.: A Framework for Interacting with Distributed Programs and Data. In: Proc. of the Seventh Int'l Symp. on High Performance Distributed Computing 1998 (HPDC-7). IEEE, July (1998)
6. Hackstadt, S., Malony, A.: DAQV: Distributed Array Query and Visualization Framework. Journal of Theoretical Computer Science, special issue on Parallel Computing Vol. 196, No. 1-2, April (1998) 289–317
7. Malony, A. D., Hackstadt, S.: Performance of a System for Interacting with Parallel Applications. International Jornal of Parallel and Distributed Systems and Networks. (1998)
8. Miller, B., Callaghan, M., Cargille, J., Hollingsworth, J., Irvin, R., Karavanic, K., Kunchithapadam K., Newhall T.: The Paradyne Parallel Performance Measurement Tools. IEEE Computer Vol. 28, No. 11, November (1995)
9. Mohr, B., Malony, A., Cuny, J.: TAU. In: Wilson, G., Lu, P. (Eds.): Parallel Programming using C++. M.I.T. Press (1996)
10. Reynders, J. et. al.: Pooma: A Framework for Scientific Simulation on Parallel Architectures. In: Wilson, G., Lu, P. (Eds.): Parallel Programming using C++. M.I.T. Press (1996) 553–594
11. Shende, S., Cuny, J., Hansen, L., Kundu, J., McLaughry, S., Wolf, O.: Event and State-based Debugging in TAU: A Prototype. Proc. SIGMETRICS Symp. on Parallel and Distributed Tools. ACM May (1996)
12. Shende, S., Malony, A. D., Cuny, J., Lindlan, K., Beckman, P., Karmesin, S.: Portable Profiling and Tracing for Parallel, Scientific Applications using C++. Proc. 2nd SIGMETRICS Symp. on Parallel and Distributed Tools. ACM Aug (1998)