Design and Implementation of a Hybrid Parallel Performance Measurement System

Alan Morris, Allen D. Malony, and Sameer Shende
Performance Research Laboratory
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403
{amorris,malony,sameer}@cs.uoregon.edu

Kevin Huck
Barcelona Supercomputing Center
Centro Nacional de Supercomputación (BSC/CNS)
C/ Jordi Girona 29, Barcelona 08034, Spain
kevin.huck@bsc.es

***Abstract*—Modern parallel performance measurement systems collect performance information either through probes inserted in the application code or via statistical sampling. Probe-based techniques measure performance metrics directly using calls to a measurement library that execute as part of the application. In contrast, sampling-based systems interrupt program execution to sample metrics for statistical analysis of performance. Although both measurement approaches are represented by robust tool frameworks in the performance community, each has its strengths and weaknesses. In this paper, we investigate the creation of a hybrid measurement system, the goal being to exploit the strengths of both systems and mitigate their weaknesses. We show how such a system can be used to provide the application programmer with a more complete analysis of their application. Simple example and application codes are used to demonstrate its capabilities. We also show how the hybrid techniques can be combined to provide real cross-language performance evaluation of an uninstrumented run for mixed compiled/interpreted execution environments (e.g., Python and C/C++/Fortran).**

Keywords: parallel, performance, measurement, analysis, sampling, tracing, profiling.

I. Introduction

In the field of scalable parallel performance tools there are differences of opinion about how the performance of a running program should be observed. The nature of the differences is technical, but the opinions are more philosophical, metaphysical, and even religious. While healthy debate is alive and well in the performance tools community, it has not led to a consensus on performance observation, nor is it likely to. Rather, performance tool research and development has somewhat polarized itself in camps based on two general performance measurement strategies: *sampling-based* or *probe-based*. Sampling-based measurement (SBM, a.k.a. *statistical sampling*) determines the performance of an application's execution by statistical observation via interrupts. How interrupts are generated and what performance information can be obtained distinguishes two SBM types: *event-based sampling* (EBS) and *instruction-based sampling* (IBS)[1]. In contrast, probe-based measurement (PBM, a.k.a., *direct measurement*) instruments the application code at specific points to collect performance data at the time the instrumented measurement code (the "probe") is executed. Once performance data is collected, application execution continues.

The main technical difference between SBM and PBM is clear. The controversy comes from the set of assumptions each strategy makes concerning accuracy and intrusion. SBM can measure and attribute performance information accurately as long as 1) code segments are executed repeatedly, 2) the execution is sufficiently long to collect a large number of samples, and 3) the sampling frequency is uncorrelated with the execution behavior. In short, SBM is grounded in statistical sampling theory. On the other hand, PBM can measure performance information accurately as long as 1) the overhead of executing the measurement code is sufficiently small relative to the size of the entity being measured, and 2) the intrusion due to measurement overhead in one thread does not affect the performance of any other threads. In short, PBM is grounded in measurement theory.

These two theoretical perspectives draw the dividing lines for parallel performance tools. The SBM camp argues that instrumentation is undesirable, that sampling results in much lower overhead and distortion, and that only sampling can observe fine-grained performance. The PBM camp argues that inaccuracies in statistical measures are to be avoided, that instrumentation is necessary for obtaining some performance information, and that certain parallel interactions can only be observed and understood with direct measurements. Of course, the reality is that there are no truths in parallel performance observation and each technique has its advantages and disadvantages, strengths and weaknesses. The proper performance tool engineering response then would be to develop a combined measurement approach. Indeed, the

[1]Instruction-based sampling is built on an event-based sampling substrate.

opportunity has been there for quite some time. However, given the complexity of performance tool development for scalable parallel platforms, it is understandable that this has not been previously pursued.

In this paper we introduce a parallel performance observation strategy based on a *hybrid measurement* framework combining probe-based measurement, based on the *TAU performance system* [1] technologies, and sampling-based measurement, using technologies from *HPCToolkit* [2] and others. Our goal is to demonstrate how an integrated measurement approach can be used to gain advantages of dual techniques and deliver effective utility in practice. Section II describes our approach to the design of the hybrid measurement system and includes a description of our prototype implementation. In Section III, we describe approaches to the analysis of the output from the hybrid measurement system, including both profile and trace analysis. Some simple examples are used to highlight the capabilities of the hybrid measurement system in Section IV. Section V covers the use of the prototype on real world codes at larger scale. Section VI compares our approach with related work. Finally, future directions of this work along with final remarks are detailed in Section VII.

II. Measurement System Design

The overall design of a hybrid measurement system will include both probe-based and sampling-based measurement components with integration points between them. A reasonable starting point is to just use two tools, one of each type, at the same time. For instance, we could instrument a program with TAU and execute it with a statistical profiler turned on. Thus, one of the key questions of a hybrid measurement design is how to gain measurement synergy through more tightly coupled integration.

The design strategy we adopted in our hybrid measurement system is to have the direct measurement system runs as before, measuring performance characteristics of regions of user code, or on abstract regions/phases/data. Meanwhile, the sampling subsystem is fully active and receives interrupts from the system, either based on timer signals or hardware counter overflows. Now, it is what happens at points where the hybrid system gains control – probe and interrupt – that represent integration opportunities. For instance, at measurement start and stop points, information could be passed to the sampling subsystem to perform necessary bookkeeping. At interrupt sampling points, information could be retrieved from the direct measurement subsystem to better qualify the execution states.

Our hybrid measurement system, called *TAUebs*, integrates the TAU performance system with event-based sampling measurement. The internal sampling technology in TAUebs is built upon previous work by several different research groups. Hardware counter sampling and measurement is made possible by the PAPI project [3]. The timer-based sampling capability is borrowed from the PerfSuite project [4]. Callstack unwinding on fully optimized codes is provided by an adaptation of the HPCToolkit [2] toolset developed at Rice University. The instrumentation and direct measurement infrastructure is based on TAU, which is also the basis for the integration. It should be noted that a hybrid measurement system could also be based off of a sampling tool with instrumentation/measurement technology folded in from other tools. We chose the former for our familiarity with the code and methodology of the TAU performance system.

In a sampling-based measurement tool, the interrupt handler gathers information for a sample and either updates an aggregated data structure, such as a program counter (PC) histogram for online profiling, or writes the information to a trace. The sample information contains PC context data and performance data. The PC context can include the PC calling path obtained by unwinding the calling stack frames. In our hybrid scheme, there is additional information available. TAUebs can capture the active context as seen by direct measurement system by interrogating TAU's event stack (we call this the *TAU context*). The PC context and the TAU context may share entries when routines have been instrumented. However, when non-routine level instrumentation is inserted in user codes (e.g., loops, phases, abstract user-level events), these events may only be visable in the TAU context and can be stored with the sample.

When recording both the PC calling context and the TAU context at a sample point, there is a possibility that the two will overlap. One objective of a hybrid approach is to use direct measurement for events where probe-effects are minimal and sampling-based measurement for small-grained observation. The appearance of an overlap affords an opportunity to optimize retention of context information. One approach we attempted was to simply store both contexts and attempt to resolve them post-mortem. This has a higher overhead due to the full unwinding of the callstack beyond the overlap point. To improve speed, we perform some bookkeeping at region start/enter hooks by walking back a few steps of the program counter callstack. We store this callstack snippet temporarily for the sampling subsystem to compare against. The sampling subsystem can then stop walking the stack when it matches against any address in the stored snippet. We found that we needed to store a small section of addresses in the snippet rather than a single address due to the fact that our instrumentation hooks are encountered at different levels based on the instrumentation language and layer (e.g., MPI wrapper library).

The interrupt context is also an opportunity to store additional information from the direct measurement tool such as the delta time to the start of the currently active TAU event, which can be used for code folding analysis[5]. We

can also store delta-end values by doing some bookkeeping at direct-measured event stop hooks, when a sample has been captured during the event's lifetime.

The output of our hybrid system is the standard profile/trace output from the code instrumented and measured with TAU, as before, plus the new data from the sampling subsystem. Our current approach is to output the samples as traces where each record contains a timestamp, a key to the currently active TAU context, and a key to the sampled callstack. Additionally, we measure the actual values of hardware counters and store them with each record, if desired. TAUebs also have record holders for delta start and stop values for each metric, if desired.

One of the more powerful capabilities of the TAUebs hybrid measurement system is the capability to fuse a program counter callstack view with an interpreted language's internal callstack. Typical sampling-based tools such as HPCToolkit are "language independent" because they operate on application binaries rather than source code. For programming languages that are compiled to machine code with debugging information, this works just fine because that information can be used to map instructions back to application source code. However, attempts to use this approach with a mixed Python/C/Fortran code or Java/JNI code will fail because the callpaths shown by the performance tool will contain references to the internal implementation routines of the interpreter or virtual machine. This data is nearly useless and most likely meaningless to the application programmer whose interpreted code regions are not shown in the reported performance data.

Probe-based measurement systems can easily use the hooks provided by interpreters and virtual machines such as Python and Java to generate events for direct observation. Indeed, they generally do so without any modifications to user's source code. Using our hybrid approach, we can measure both interpreted code and uninstrumented, fully optimized C/C++/Fortran libraries in the same execution.

III. Data Analysis

Our initial prototype system processes the EBS traces in two distinct ways: merged profile creation and trace conversion.

A. Merged profile creation

In the profile merge mode, the ParaProf parallel profile viewer reads the TAU measured profile and then augments it with data from the EBS trace. The traces contain both the TAU event stack and the callstack obtained through program counter stack walking. These callstacks are merged and the data is stored into the ParaProf profile. The basic idea behind the combination of the two types of profiles is a simple count of the number of samples encountered for a given TAU event path (i.e., callpath). We know the total overall time spent in a given TAU callpath from the standard profile. If ten samples occurred while this callpath was active, we can distribute the time associated with the callpath among the 10 samples. For example, if eight of the samples were from one particular location and two of them from another, we would assign 80% of the time to the first location, and 20% to the second.

When dealing with the program counter callstacks, we will have intermediate parent nodes between the location where the sample took place and the currently active TAU callpath. These intermediate nodes will be inserted into the profile and be assigned the appropriate amount of *inclusive* time. They will not have any *exclusive* time associated because no samples occurred for that location. We have multiple methods for handling the intermediate parents. In one mode, we can treat the translated addresses as singular locations that represent the *callsite* of the calling context. The *callsite* is distinguished by the fact that it describes where in the parent routine, the call took place. Instead of only showing that routine A called routine B called routine C and the sample occurred at line 45 of routine C, we could show routine A calling routine B at line 23 of routine A, calling routine C at line 36 of B, calling C, and the sample occurring at line 45 of C. Another mode of interpreting the data is to not use the callsites, but just use the routine name (and possibly the filename as well, which will not change for any location in the routine). The advantage of this approach is that it aggregates all the callsites for a particular function so that the user can see the total amounts across them all. It also makes tree-like callpath display easier to navigate since it avoids the proliferation of intermediate nodes. We have the capability to perform both kinds of interpretation in our processing.

It should be noted that the amount of performance data from direct measurement or sampling can be adjusted from one side of the measurement spectrum to the other. At one extreme, we could use no direct measurement, or instrument only "main" with a singular top level timer. Here the processed data would equate to a sampling-only based tool where the overall time is divided amongst the samples encountered over the entire execution. At the opposite end of the spectrum, all user routines could be instrumented, and the sampling data would provide only the fine-grained line number information within routines. No single approach is necessarily better or worse, but should instead be adapted to the situation.

B. Trace conversion

The other method of trace processing that we have developed in our prototype implementation is that of an EBS trace to OTF converter. The objective here is to visualize the EBS trace data in a powerful trace visualizer such as Vampir [6]. The challenge here is to convert the samples in the trace to an enter/exit style trace that still retains the essence of the collected data.

We perform the trace conversion using a tool that maintains a call stack at any given point. It processes the trace, one sample at a time and after merging the TAU event callpath with the program counter callstack given for each sample, it inspects the differences in the resolved callpath. If the resolved callpaths are the same, then current callpath for the trace is unchanged for that timestamp. Because we collect PAPI metrics at each sample point (whether we are overflowing on a PAPI metric or not), we also write the values of the PAPI metrics into the trace for hardware counter analysis when looking at the trace data in Vampir.

Figure 1 shows the display of a simple EBS trace in Vampir. The dynamic callstack is at the top with rates graphs of floating point instructions and L1 data cache misses below. The code performs two matrix multiplications: one using improper loop indexing (*slow_matmult*) and one using proper index ordering (*fast_matmult*). The floating point rate and the L1 data cache miss rate show the difference. We can also see that the proper loop indexing runs for less time (i.e. shown horizontally) for the same amount of work.

IV. Demonstration

In order to demonstrate some of the capabilities of the hybrid measurement system, we show some profile data from a simple example. The example is a mixed C and Fortran toy example where python code calls a matrix multiply operation implemented in C. With the hooks provided by the direct measurement system, we will show that the hybrid approach presents the user with the most relevant picture of the performance data. This example uses no instrumentation, but instead, the python performance interface to gain hooks into the system. So no code modifications or recompilations were necessary, only a simple wrapping of the invocation to start the profiler.

Figure 2 shows the callstacks from sampling-only based measurement and probe-only based measurement. The user's Python routines and filename do not appear since they are invisible to the measurement tool. Instead, the user is presented with a very large callstack tracing through the Python runtime system. The measurement only tree on the right shows what a direct measurement system sees without the use of any instrumentation on the C-based library used in the example. We see only the user's Python routines.

Figure 3 shows the combined hybrid approach where the two callstacks are merged and resolved at the correct point and only the user's code is seen. Routine nesting is exactly as the user expects.

V. Examples

We have tested our prototype hybrid measurement system against a number of real world codes on a variety of platforms, including leadership class machines. We picked three to demonstrate different aspects of the hybrid measurement system.

Tool	Instrument	Sample Period	Runtime	Overhead
None	–	–	654s	–
HPCToolkit	–	1/2000s	674.21s	20.21s (3.1%)
HPCToolkit	–	1/1000s	682.97s	28.97s (4.4%)
HPCToolkit	–	1/500s	704.06s	50.06s (7.7%)
TAU	Main	–	654.40s	0.40s (0.1%)
TAU	Main	1/2000s	672.55s	18.55s (2.8%)
TAU	Main	1/1000s	682.23s	28.23s (4.3%)
TAU	Main	1/500s	701.23s	47.23s (7.2%)
TAU	Limited	–	663.05s	9.05s (1.4%)
TAU	Limited	1/2000s	686.61s	32.61s (5.0%)
TAU	Limited	1/1000s	697.15s	43.15s (6.6%)
TAU	Limited	1/500s	714.91s	60.91s (9.3%)

TABLE I
Runtimes and overheads for MADNESS

A. MADNESS

To evaluate our ability to handle codes that are traditionally difficult to instrument either by source instrumentation, compiler-inserted instrumentation, or binary instrumentation, we chose to look closely at MADNESS [7], a quantum chemistry application. MADNESS makes heavy use of C++ templates, new C++ features, assembler regions/files, and a significant amount of code in header files, all making source instrumentation challenging. The use of compiler instrumentation is difficult at this point since compiler support for selective instrumentation is very limited and a fully instrumented execution with GNU compiler instrumentation incurred an overhead of **2901%** due to the numerous getter/setter routines and other small routines. Using source based instrumentation with MADNESS was challenging, but once performed, using selective instrumentation, we saw more reasonable overheads of 4.6% to 6%. By using selective instrumentation, we have introduced potential blind spots, though they are known blind spots.

Using the hybrid measurement system, we can measure these sections of uninstrumented code with the sampling subsystem. This will also allow us to see inside any uninstrumented math libraries used.

We ran MADNESS in SMP mode using 8 threads on an 8-core 2.3Ghz machine. We experimented with different source instrumentation levels and different sampling intervals of 500 samples per second (1/2000s), 1000 samples per second (1/1000s) and 2000 samples per second (1/500s). We compared the overheads to HPCToolkit's hpcrun with the same sampling rates.

Table I shows the runtimes and overheads for MADNESS. The overheads are comparable to that of HPCToolkit. The instrumentation column represents the level of instrumentation present. Where a double dash is in the column, it is not applicable. For TAU, we have either *Main* which refers to main-only instrumentation, which is effectively the same as no instrumentation (only *main* is instrumented) and *limited* instrumentation which refers to the automatic

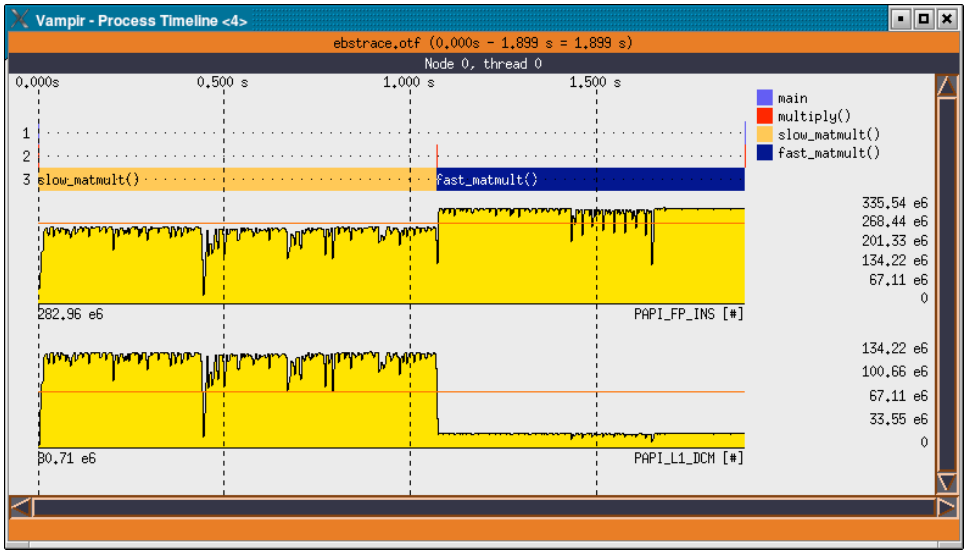

Fig. 1. Vampir display of EBS trace converted to OTF, two PAPI counter rates are shown for two different matrix multiply routines.

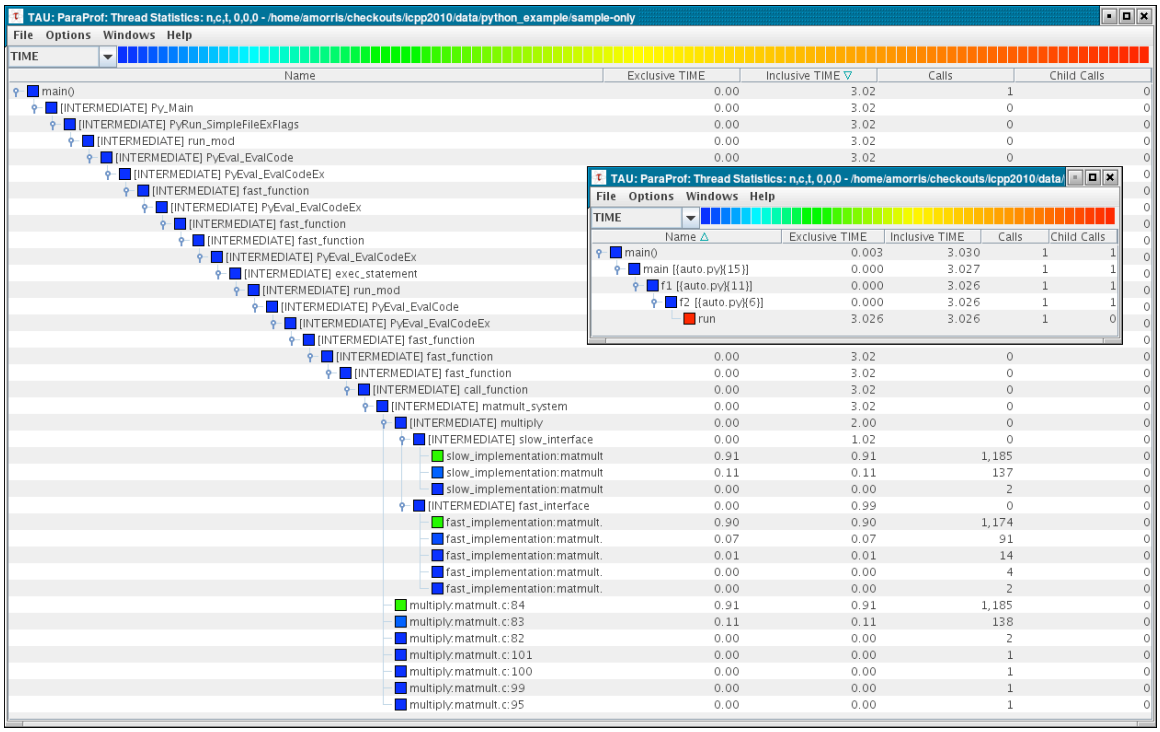

Fig. 2. Call trees from sampling-only (left) based and probe-only (right) based measurement.

selective instrumentation done by TAU. We can see that the TAU EBS approach is a bit more simplistic and at a low core count generating minimal trace data with low sample rates, the overhead is less than that of HPCToolkit, due primarily to the fact that we are doing very little processing at runtime, only writing a trace record. When TAU instrumentation is used, there is more work to do, and we incur the overhead of the measurement probes. These numbers are higher than HPCToolkit, but are still comparable.

An eleven minute run of MADNESS on 8 threads using 1/1000s sampling period generated approximately 67MB of trace data per thread, and a callpath profile of about 250KB. Figure 4 shows the combined profile data for thread 0. The first four routines in the tree are standard instrumented

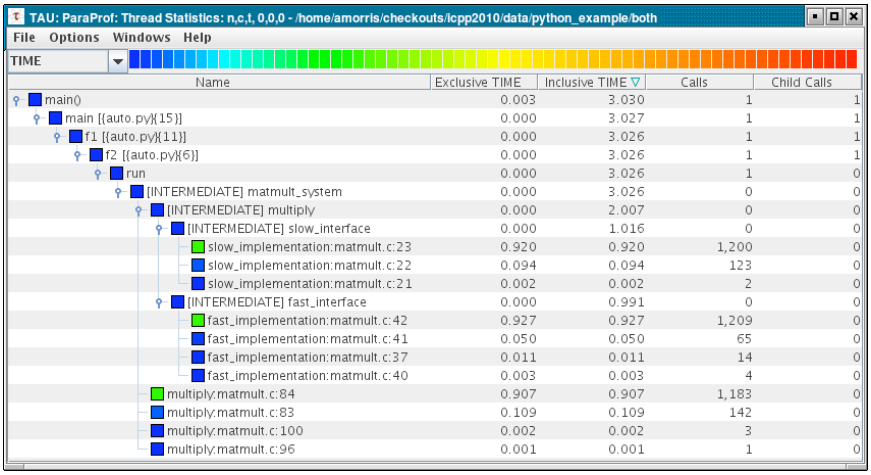

Fig. 3. Call tree using hybrid measurement approach, representing the user's view of both their Python and C code.

TAU events. The nodes labeled *INTERMEDIATE* are those uninstrumented routines between the sample and the nearest instrumented region. In this view, we see a significant amount of time in **.TKLOOP16** from *mtxm_gen.h*, which is an assembly file that the TAU source instrumentor cannot instrument. In Figure 5, we can see the trace data visualized in Vampir, showing a thread of execution for a small segment of the trace. The routines are grouped and colored based on the filename. Along the bottom, the rate of floating point instructions is seen.

B. GPAW

Our simple example showed the potential power of our method with mixed Python and C environments. For a real world example of this approach, we applied our prototype to GPAW (Grid-Based Projector-Augmented Wave Application) [8]. GPAW is a DFT-based code that is built on the projector-augmented wave (PAW) method and can use real-space uniform grids and multigrid methods.

We did not instrument GPAW using source or binary instrumentation. Instead, we used a simple wrapper to start the Python profiling interface and used LD_PRELOAD to capture MPI events. In Figure 6, we can see the resulting combined profile for node 0 from a 128 processor MPI parallel run. In the figure, we can see both user Python methods near the top of the calltree and lapack C routines at the bottom of some callpaths.

C. FLASH

FLASH [9] is a parallel adaptive-mesh multi-physics simulation code designed to solve nuclear astrophysical problems related to exploding stars. The FLASH code solves the Euler equations for compressible flow and the Poisson equation for self-gravity.

Description	Procs	Runtime	Overhead
uninstrumented	240	636s	–
TAU measurement	240	648s	12s (1.9%)
TAU measurement and sampling	240	672s	36s (5.6%)
uninstrumented	484	674s	–
TAU measurement	484	689s	15s (2.2%)
TAU measurement and sampling	484	713s	39s (5.8%)
uninstrumented	1004	649s	–
TAU measurement	1004	670s	21s (3.2%)
TAU measurement and sampling	1004	697s	48s (7.4%)
uninstrumented	2176	656s	–
TAU measurement	2176	695s	39s (5.9%)
TAU measurement and sampling	2176	699s	42s (6.6%)
uninstrumented	4416	669s	–
TAU measurement	4416	729s	60s (9%)
TAU measurement and sampling	4416	771s	102s (15.2%)
uninstrumented	8192	729s	–
TAU measurement	8192	847s	118 (16.2%)
TAU measurement and sampling	8192	863s	134s (18.4%)
uninstrumented	15812	781s	–
TAU measurement	15812	997s	216s (27.7%)
TAU measurement and sampling	15812	1144s	363s (46.5%)

TABLE II
RUNTIMES AND OVERHEADS FOR FLASH

We ran FLASH on Intrepid, an IBM BlueGene/P at the Argonne Leadership Computing Facility, on up to 15,000 cores to investigate scaling issues of our method.

Table II shows the resulting perturbation of our current approach. It is important to note that this does not include the majority of I/O metadata operations such as file open/close that can take significant amounts of time if one file is written per processor on high core counts. The numbers presented here represent only the increase in time for the actual application runtime. It is clear that in the high core count cases, the tracing we are performing is having a significant impact. Further work needs to be done to determine the best course of action to remedy this.

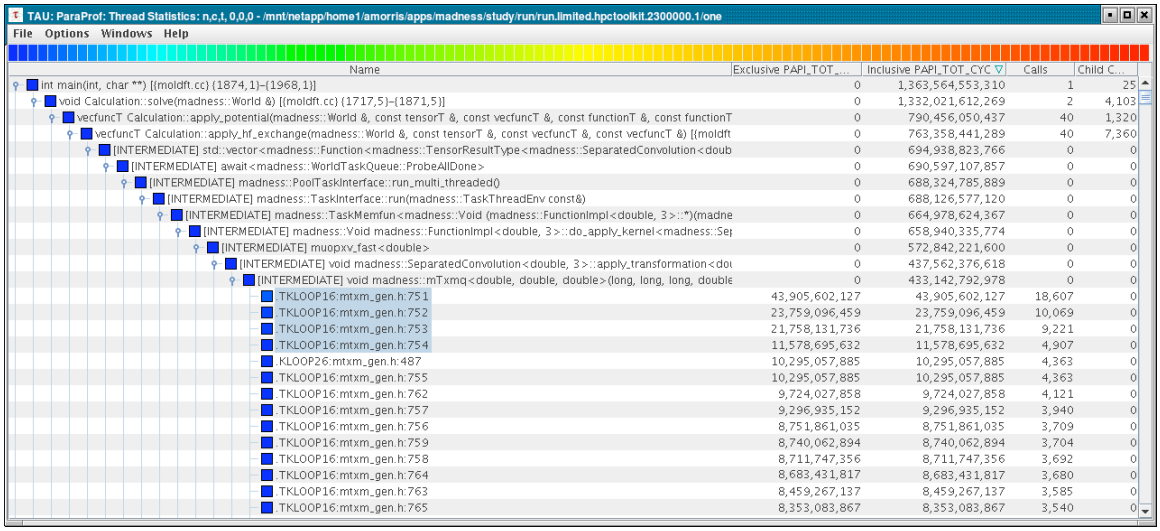

Fig. 4. MADNESS combined probe and sampling based profile. We see a hot path going down to some sampled routines implemented in assembly from top level source events with probes.

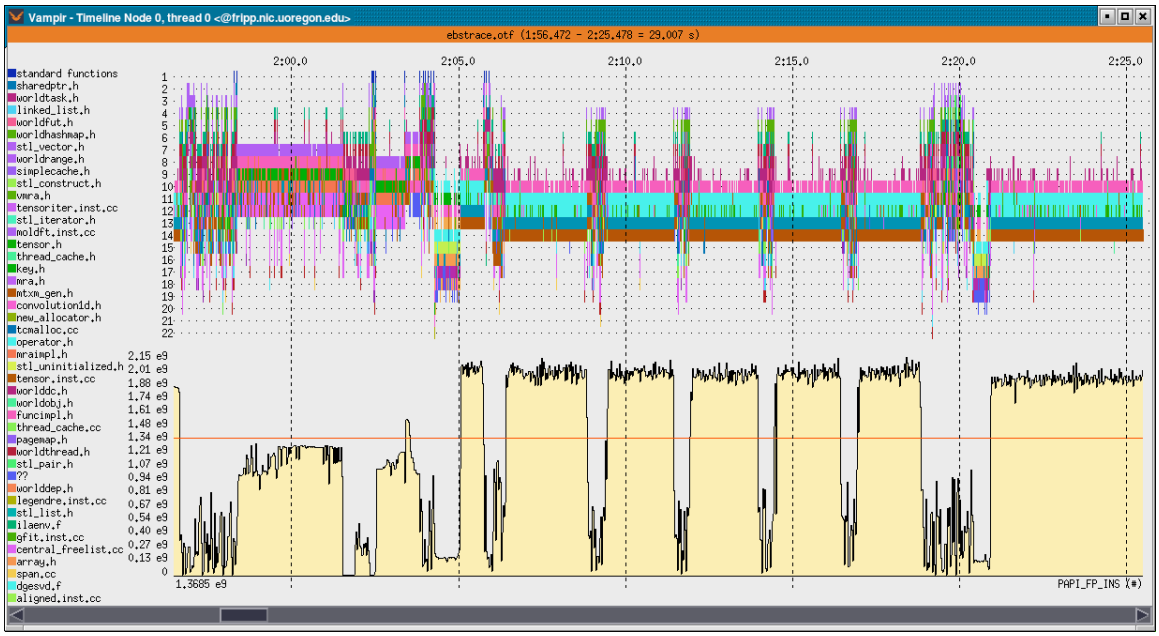

Fig. 5. MADNESS Trace for one thread shown in Vampir. Routines are colored by filename and floating point counter rates are shown at the bottom.

ParaProf's calltree viewer can also show reverse call trees. For FLASH, we can see the reverse calltree in figure 7, showing that some of the top routines were those sampled from internal routines in the MPI communication library. Using the reverse calltree, we can partition that time out, in this case about 53 seconds, which is almost entirely from MPI_Barrier calls. Of those MPI_Barrier calls, about half the time was under the region *** *custom:guardcell Barrier*. FLASH is internally instrumented with region timers that can be configured to map to TAU timers, and those are what we see. These regions do not correspond to routine names, and so a sampling-only tool would not able to see them. Figure 8 shows trace data for a 240 processor run, zoomed to show about 40 seconds of execution time on 60 of the MPI ranks. The events are colored according to the filename that each routine is from. Patterns of execution are clearly seen. The alternating behavior seen between consecutive nodes is due to the different internal routines of the MPI implementation. We ran FLASH in dual mode, meaning that two MPI ranks were assigned to each node,

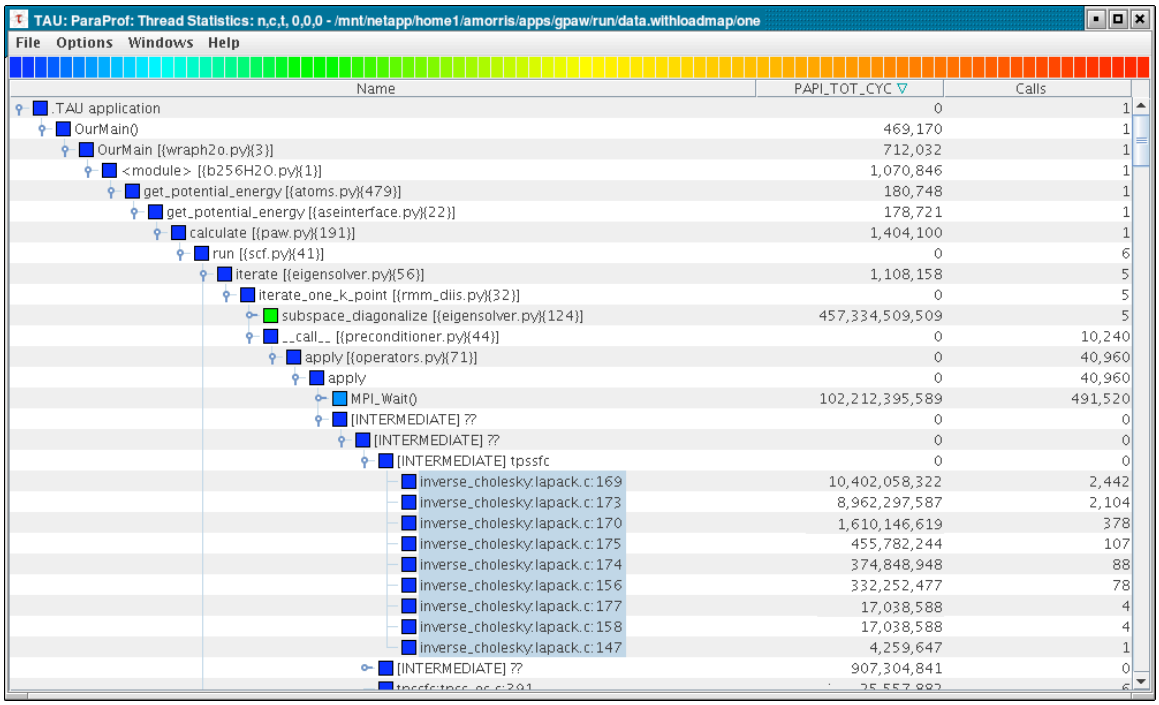

Fig. 6. GPAW combined profile. We can see the application python routines call down into an uninstrumented lapack library.

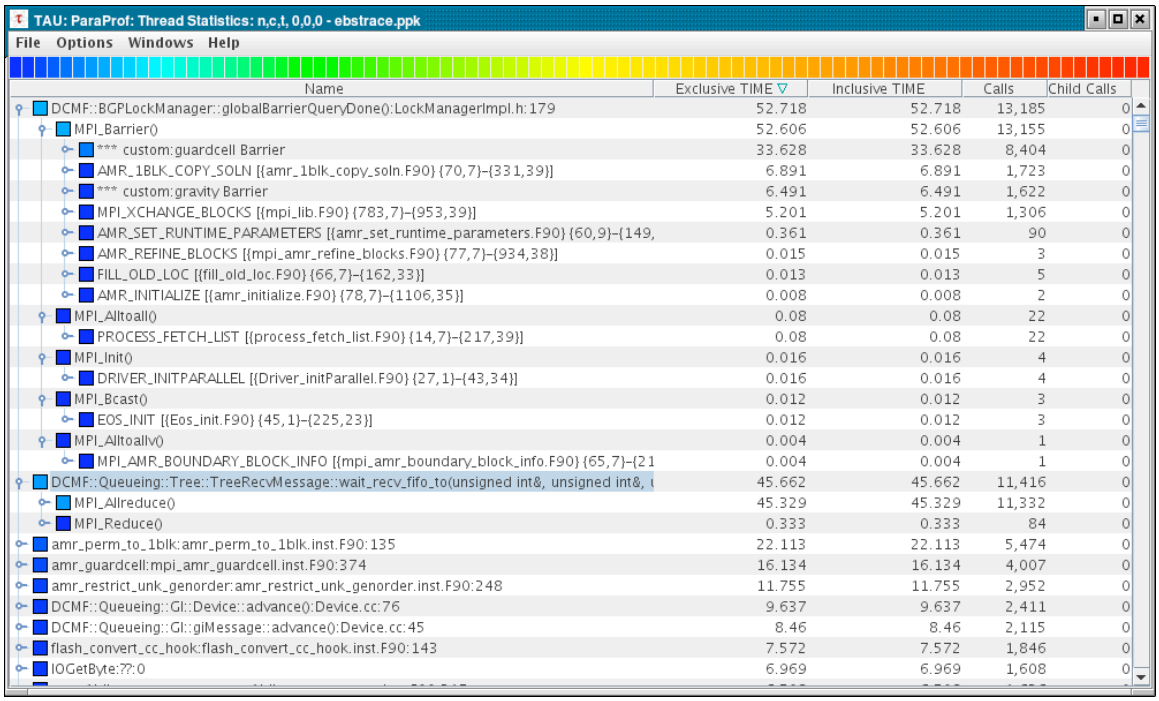

Fig. 7. Reverse calltree of FLASH combined profile showing the probe-based parent routines in the MPI Wrapper library for internal MPI implementation routines.

the MPI implementation has specialized routines for each mode and global synchronizations make use of them. The trace visualization and profiles shows this clearly.

VI. Related Work

Our hybrid measurement system inherits ideas and building blocks from a rich parallel performance tools com-

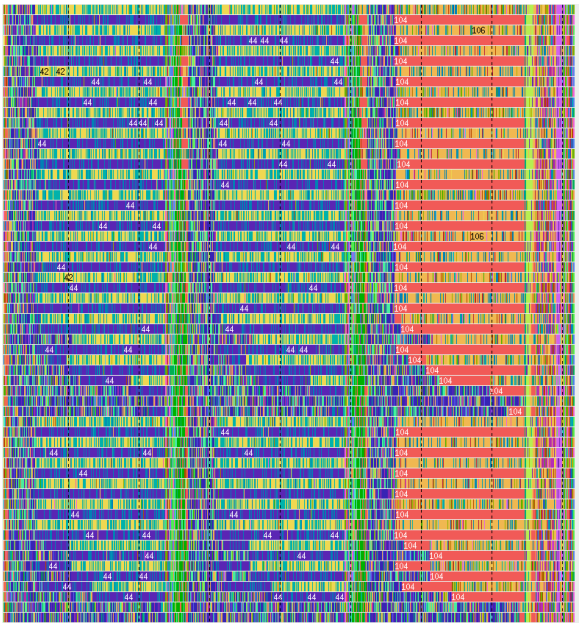

Fig. 8. EBS trace data converted to OTF, shown in Vampir for FLASH running on 240 processors (60 shown). Each horizontal line represents an MPI rank. Routines are colored by filename.

munity. Related research is distinguished by the choice of measurement technique – *sample-based* or *probe-based* measurement – as well as by how performance data is captured – profiling or tracing.

As described in the introduction, sample-based measurement techniques are of two general types: *event-based sampling* (EBS) and *instruction-based sampling* (IBS). When an "event" (e.g., timer, performance counter overflow) in EBS occurs, it triggers an interrupt, giving the measurement system an opportunity to sample the program counter (PC). The PC is then correlated with the event and the performance data collected is defined by the event properties. For instance, the classic Unix *gprof* [10] tool is based on timer-generated interrupts, allowing a performance histogram of time spent in code regions to be calculated. EBS tools such as *PerfSuite* [11] can use other events (e.g., floating point counter overflow) to observe different performance aspects. It is also possible to interrogate the callstack at the time of interrupt to determine the routine calling path. For instance, *Open|Speedshop* [12] (O|SS) is a measurement tool that utilizes the `libunwind`[1] technology for callstack walking. The *StackWalkerAPI* [13] is another library developed for this purpose. *HPCToolkit* [2] and the Sun *Performance Analyzer* are EBS measurement tools with stack walking mechanisms built in. Our approach utilizes HPCToolkit's support, but we have tested with libunwind and StackWalkerAPI.

Instruction-based sampling is also interrupt-driven. In addition to sampling the PC or calling stack, IBS uses hardware support to follow the current (next) instruction through the pipeline, collect information about its execution, and pass this data back to the interrupt handler. The IBS technique was introduced with the *ProfileMe* [14] tool for use with in the Digital Alpha 21264 microprocessor. HPCToolkit, the AMD *CodeAnalyst* [15], and the *Memphis* [16] measurement tool all use IBS support in modern processors. IBS can be beneficial in understanding memory-related performance problems. Although our hybrid measurements does not use IBS presently, we do not see any technical limitations to do so in the future.

EBS and IBS measurements are most often collected in profiles calculated during execution for each thread. Of the tools listed above, only HPCToolkit, the Sun Performance Analyzer, and Memphis will collect time-sequenced samples for post-processing. Our approach shares this features for event-based sampling. However, none of these tools gathers probe-based measurements, except in the case of MPI communication events in the Sun Performance Analyzer.

In contrast to sample-based measurement, code instrumentation is required for probe-based measurement. This allows for direct measurement of performance between code points using timers or hardware counters. Typically, application and communication routines are instrumented. Probe-based measurements are captured in both profile and trace form. Vampir [6] provides MPI communication measurement using the *PMPI* interposition library, as does O|SS, Sun Performance Analyzer, and several other tools. *Scalasca* [17] implements both scalable profiling and tracing, but has no support for sampling.

While some of the tools above support both sample-based and probe-based alternatives (e.g., O|SS), none of them implement a hybrid measurement capability like what we report in this paper. The *mpiP* tool implements an different hybrid technique for dynamic statistical profiling of communication activity by sampling probe-based measurements of MPI operations. *Paraver* [18] is developing a novel technique for EBS tracing that allows high-resolution performance analysis of routine execution using sampling, folding those samples along iteration boundaries [5]. This technique allows for detailed analysis of behavior in between instrumentation boundaries, with low measurement overhead. The inspiration for our hybrid measurement research came from Paraver's approach. In fact, we are working with the Paraver team to capture EBS traces that are compatible with their analysis tools.

VII. Conclusions and Future Work

We have developed a hybrid parallel performance measurement system integrating probe-based and sampling-based strategies, and ideas and technologies from the TAU performance system, PerfSuite, HPCToolkit, and Paraver. Our initial TAUebs prototype has been tested on multiple

platforms with simple examples and advanced application codes at small and relatively large scale. However, we limited our current solution in several aspects. We made a conscious decision to only implement EBS tracing because it was more challenging and because it targeted requirements for our interaction with the Paraver team. Though we have demonstrated that this approach has a low overhead, there are many reasons why the creation of a runtime hybrid profiler for the sampling is desirable. The creation of trace files can be problematic for systems without the necessary I/O infrastructure and the post-processing of large numbers of these files can be time consuming. A runtime aggregator would save disk space and post-mortem workflow time for the end user for those cases where traces are not needed or desired. Although a runtime profiler based on hybrid measurement is not necessary the same as a pure statistical profiler, it would make sense to build on the work previously done by other groups. One of the main problems to overcome is the scalable data structure required for runtime fine-grained profile generation. For instance, we could borrow or adapt the calling context tree (CCT) approach in HPCToolkit or the work from SGI/France on distributed hash table (DHT) profile memory allocation being integrated in PerfSuite. Any technique will be integrated with the TAU profiling framework.

Also, we are not presently performing any binary analysis of the executable code where the samples are taking place. TAUebs is merely translating the addresses from the trace, including the calling context, to function, source file, and line number data. There is a great deal to be gained from binary analysis to determine loop bounds and procedure inlining. Considerable success has been achieved from years of effort (e.g., see *hpcstruct* from HPCToolkit), and we can certainly leverage the work done there to enhance our sample-based analysis.

Our trace conversion tool does not currently allow the merging of a direct measurement trace along with the sampling trace. Such an addition would be a great benefit for a number of reasons. Besides all of the benefits from being able to see uninstrumented routines, TAUebs can collect hardware performance periodically and show a graph of hardware counter performance over time (e.g., see Figure 5). By indexing this information with an event trace, the hardware performance can be correlated with the application semantics at different levels.

We believe that both probe-based and sampling-based measurement tools have their place in the modern HPC performance tool eco-system. The advantages of a hybrid measurement system become significant when the limitations of only a single approach become a challenge to performance problem solving. By building hybrid capabilities in the TAU performance system, it is possible to "dial in" the degree of integration to target measurements needs. Based on our TAUebs findings, we plan to continue to pursue this approach to provide a more complete, more capable performance measurement system.

REFERENCES

[1] "TAU: Tuning and Analysis Utilities," http://www.cs.uoregon.edu/research/paracomp/tau/.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, 2010.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, Fall 2000.

[4] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, "Perfsuite." [Online]. Available: http://perfsuite.ncsa.uiuc.edu/

[5] H. Servat, G. Llort, J. Giménez, and J. Labarta, "Detailed performance analysis using coarse grain sampling," in *PROPER 2009*, 2009.

[6] H. Brunst, D. Kranzlmüller, and W. E. Nagel, "Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz," *Distributed and Parallel Systems, Cluster and Grid Computing*, vol. 777, 2004.

[7] R. J. Harrison, G. I. Fann, T. Yanai, and G. Beylkin, "Multiresolution quantum chemistry in multiwavelet bases." in *International Conference on Computational Science*, ser. Lecture Notes in Computer Science, P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, Eds., vol. 2660. Springer, 2003, pp. 103–110.

[8] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen, "Real-space grid implementation of the projector augmented wave method," *Phys. Rev. B*, vol. 71, no. 3, p. 035109, Jan 2005.

[9] R. Rosner et. al., "Flash Code: Studying Astrophysical Thermonuclear Flashes," *Computing in Science and Engineering*, vol. 2, pp. 33–41, 2000.

[10] S. Graham, P. Kessler, and M. McKusick, "gprof: A Call Graph Execution Profiler," *SIGPLAN '82 Symposium on Compiler Construction*, pp. 120–126, June 1982.

[11] R. Kufrin, "PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux," in *Sixth International Conference on Linux Clusters (LCI)*, 2005.

[12] M. Schulz, J. Galarowicz, and W. Hachfeld, "Open—speedshop: open source performance analysis for linux clusters," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, p. 14.

[13] StackWalkerAPI. [Online]. Available: http://www.paradyn.org/html/stackwalker1.1-features.html

[14] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: hardware support for instruction-level profiling on out-of-order processors," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, 1997, pp. 292–302.

[15] Advanced Micro Devices, "AMD CodeAnalyst performance analyzer," http://developer.amd.com/cpu/codeanalyst.

[16] C. McCurdy and J. Vetter, "Memphis: Finding and Fixing NUMA-related Performance Problems on Multi-core Platforms," *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[17] F. Wolf, B. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi, "Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications," in *Proc. of the 2nd HLRS Parallel Tools Workshop*. Springer, July 2008, pp. 157–167.

[18] I.-H. Chung, S. Seelam, B. Mohr, and J. Labarta, "Tools for scalable performance analysis on petascale systems," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, 2009, pp. 1–3.