2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

# General Hybrid Parallel Profiling

Allen D. Malony[1] and Kevin Huck[2]

[1]Department of Computer and Information Science, University Oregon,Eugene, Oregon
[2]Performance Research Laboratory, Neuroinformatics Center, University of Oregon, Eugene, Oregon

***Abstract*—**
**A hybrid parallel measurement system offers the potential to fuse the principal advantages of probe-based tools, with their exact measures of performance and ability to capture event semantics, and sampling-based tools, with their ability to observe performance detail with less overhead. Creating a hybrid profiling solution is challenging because it requires new mechanisms for integrating probe and sample measurements and calculating profile statistics during execution. In this paper, we describe a general hybrid parallel profiling tool that has been implemented in the TAU Performance System. Its generality comes from the fact that all of the features of the individual methods are retained and can be flexibly controlled when combined to address the measurement requirements for a particular parallel application. The design of the hybrid profiling approach is described and the implementation of the prototype in TAU presented. We demonstrate hybrid profiling functionality first on a simple sequential program and then show its use for several OpenMP parallel codes from the NAS Parallel Benchmark. These experiments also highlight the improvements in overhead efficiency made possible by hybrid profiling. A large-scale ocean modeling code based on OpenMP and MPI, MPAS-Ocean, is used to show how the TAU hybrid profiling tool can be effective at exposing performance-limiting behavior that would be difficult to identify otherwise.**

***Index Terms*—Parallel, performance, analysis, tools**

## I. Introduction

In the world of parallel performance analysis, tools are generally distinguished by their choice of measurement approach. *Sampling-based* (a.k.a. statistical sampling) methods measure performance by statistical observation during interrupts. *Probe-based* (a.k.a. direct) methods insert code into a program to make visible specific events during execution that can be measured directly. The technical differences are clear and advocates of each can point to their advantages versus the other method (see [8]) and rightly so. The fundamental issues inherent in statistical sampling theory and measurement theory – accuracy, intrusion, uncertainty – can be found in many areas of scientific practice, often leading to competing techniques being present in many fields. However, there are important benefits to be gained in performance observability for parallel computing by looking for opportunities to integrate sampling-based and probe-based techniques in a *hybrid* measurement system, especially to improve performance interpretation and fidelity.

The research presented here is a continuation of work to create a general hybrid measurement system for parallel performance analysis. By "general" we mean the ability to apply the joint capabilities flexibly to performance analysis requirements. Our initial effort [4] extended the TAU Performance system® [1] with sampling-based measurement for hybrid parallel *tracing*. The basics for enabling sampling in a probe-based infrastructure were developed, but tracing is not a general solution for all performance analysis scenarios and is prone to scalability problems. Profiling, while giving up tracing's observation of time dynamics, is more versatile in practice. However, hybrid profiling is more challenging due to the complex analysis being done online.

The paper reports on our progress to realize a general hybrid parallel profiling solution. Specifically, we see the contributions of our work including:

- Design of techniques for the merging of probe-based and sampling-based techniques for parallel profile measurements.
- Solving issues of sample attribution relative to probed events not on the routine calling stack.
- Development and implementation of the hybrid profiling techniques in the TAU measurement system and analysis tools.
- Demonstration of general hybrid parallel profiling on a multiscale ocean modeling application written in OpenMP and MPI, in particular to show its ability to report richer performance information.

Section §II describes the approach to hybrid measurement integration and the challenges for profiling. Our solution design and its implementation in TAU are presented in Section §III. Here we review the development problems encountered and their resolution in our working prototype. Section §IV evaluates the hybrid methods with two case studies. Several projects relate to our efforts and are discussed in Section §V. The paper concludes with a synopsis of results, planned improvements, and an outlook to future directions.

## II. Approach

Event-based sampling is so-called because certain *events* trigger interrupts to occur, enabling sampling measurements to observe program state. Timer or hardware interrupts are commonly used to stop a thread of execution and interrogate the program counter, callstack, and other *context* about the computation. Sampling measures performance statistically, in the sense that time and counts are attributed to the program context at the time of the interrupt. Under assumptions of regularity, stationary, and large sample size, performance can be approximated with relatively good accuracy. In particular, sampling can capture fine-grained behavior that would be hard to see accurately with probes. However, all information about the program's computational structure (e.g., routines, dynamic callgraph/callpaths, loops) must be determined at runtime through PC resolution and callstack interrogation. The degree of callstack unwinding can be controlled to determine the granularity of mapping of samples to the dynamically-created callgraph structure during execution. A profile for each thread of execution is created, which at minimum contains performance data for each routine seen in execution at the time a sample was taken.

In contrast, *events* in probe-based measurement represent where the probes are placed in a program and what aspect of the program they reflect. Here, events are made manifest when the inserted probes are executed. Typically, *begin/end* events (a.k.a. interval events) are used to make measurements between a begin and end point (e.g, routines, loops). Profile measurements are made directly for each thread of execution based on which events are executed in the threads. Because events can be nested, the *event stack* maintained by the profiling system can be used to attribute performance data to *event paths*. A key distinction with sampling is that *only* probed events will be observed (measured) and the events do not have to correspond *only* to routines. Which events are instrumented (i.e., which probes are active) can be controlled in probe-based measurement. The program's computational structure as represented in the performance profile is described directly by the events that occurs during execution.

The objective of a general hybrid parallel profiling tool is to merge probe-based and sampling-based techniques to deliver a more powerful framework for performance measurement that can be controlled for specific needs. Our approach uses a probe-based measurement infrastructure as scaffolding for overlaying sampling-based performance data. The basic idea is to associate samples with the currently active probe "context" (i.e., observation focus), as identified by the event stack. In this way, samples taken can be immediately partitioned based on the current context. The timers and samples can capture metrics other than just wall clock time, such as hardware counters, memory usage, and power consumption, providing a rich measurement capability.

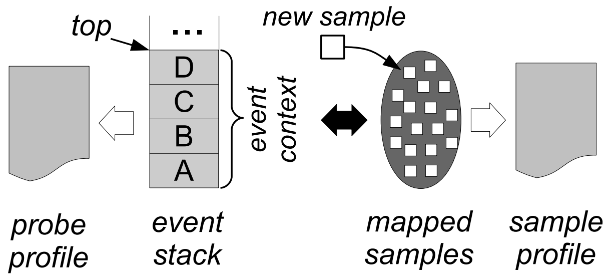


Fig. 1. Hybrid profiling concept.

Figure 1 depicts the proposed hybrid profiling approach. On the left side is the dynamic event stack created by a probe-based tool. At this point in the execution, the program is "in" *event D*, which is nested within *event C*, *event B*, and *event A*, respectively. The event sequence $A \rightarrow B \rightarrow C \rightarrow D$ in the event stack form a unique *context*. When a new sample occurs, the current context is used to map the sample to a set of samples for that context and the profile associated for the mapped samples is updated. The probe-based profile is updated when event transitions take place. The hybrid profiling approach is parallel because every thread of execution captures its own profile as above At the end of the program execution, all of the hybrid thread profiles are output and a parallel profile analysis tool is used to interpret and present the data.

## III. Development

The general hybrid profiling approach discussed above has been developed and is available in the TAU Performance System. The starting point for our work was the earlier implementation of sampling-based measurements in TAU coupled with trace recording. However, a parallel profiling solution introduces further complexities due to the need for online profile analysis. In contrast to a profile generated from trace-based analysis, the goal here is to capture all of the performance information possible at runtime that will produce equivalent profile results.

The following describes how this is accomplished in TAU. We start with improvements made in the underlying sampling infrastructure. The hybrid profiling measurement and analysis implemented in TAU is then described, along with what the user would expect to see in the parallel profile output. TAU has unique support for probe-based measurement that make it attractive for supplementing with hybrid capabilities. These are discussed and a simple example is given to help understand the hybrid profiling features.

### A. Sampling Methods

Sampling relies on a trigger to interrupt the program's execution, typically from periodic timers or hardware counter overflows. Our default sampling mechanism is based on timer interrupts occurring at 100ms intervals (configurable through an environment variable). It is also possible to enable interrupts on PAPI [10] counter overflow. Hybrid profiling in TAU is done at the thread level. Thus, all active threads of the application will see interrupts. When an interrupt occurs, a TAU handler routine is activated and performs the following steps:

- Determine program counter where the program was executing
- Check if executing TAU code and process sample accordingly
- Unwind the calling stack a certain *unwind degree* (configurable)

The current program counter context is provided by the signaling mechanism. Because hybrid profiling is active, there is a chance that TAU was in the middle of handling an event probe for the application at the time of the interrupt. A flag is set upon entry to TAU and reset on exit. If the interrupt handler finds the flag set, it does not process the sample. A process-global count of samples dropped is incremented and will be output with the final profile.

The interrupt handler can capture call path data by unwinding the call stack. If call stack information is not required, a *flat profile* of the time spent for each line of code sampled in the context of TAU probe events is recorded. Enabling call stack unwinding to a certain degree $D$ allows call paths of up to length $D$ ending at the sampled PC to be used to distinguish the sample in the hybrid profile. Alternatively, the unwinding can terminate automatically by comparing the callstack of the sample with the callstack at the beginning of the current timer. When a common ancestry in the stack is found, unwinding can terminate. There are different packages available for call stack unwinding, each with it advantages and disadvantages. Previously, we used the HPCToolkit [2] call stack unwinder because it was most robust. However, that code is tightly coupled with HPCToolkit and not available as a library. Instead, we created a modular call stack unwinding interface in TAU whereby different options can be used more flexibly, depending on platform availability. The Linux *backtrace* facility will be used by default. However, the *libunwind* [12] library is often available and is arguably the most portable stack unwinding system. It is our preferred unwinding option new integrated framework and has clear documentation describing which of its interface calls are thread and signal safe. Unfortunately, libunwind is not 100% robust on all major HPC platforms. *StackwalkerAPI* [13] is a strong alternative to libunwind, although less robust with respect to its use within the context of a signal handler. We have included support for StackwalkerAPI under our modular framework with caveats.

### B. Hybrid Profile Measurement

Once the PC and call path are determined, TAU performs the following steps to measure the hybrid profile for a sample:

- Determine the TAU event context
- Update the hybrid profile for the sample and event context

The TAU event context is updated during probe measurement operations. When a probe begin event occurs, a new event context is determined by a hash of the old event context and the new event. The new event is placed on the TAU event stack together with the new event context. Thus, when a sample is being processed, the current event context can be determined quickly.

For every sample uniquely defined by the tuple {*event context, PC, call path*}, the frequency count and accumulated measurements are updated and stored for every thread. Concurrently, TAU profiles for every probed event that occurs on every thread are being measured as the application executes. The extent of the hybrid profile measurement is determined by the number of probe events, the depth of event paths desired, the depth call path desired, and the uniqueness of event contexts observed. All but the last of these determinants are under the user's control.

### C. Hybrid Profile Generation

At the end of the application's execution, the sample addresses are resolved to their symbolic names, profile metric statistics are calculated, and the hybrid profiles are written to files. For each probed event context, an entry is created in the profile to represent the samples captured while the application was in that context. These "context events" share the names of their probed parents and are labeled as "CONTEXT" in the profile. For example, the Hybrid profile in Figure 2 has a CONTEXT event for a leaf timer event (matrixMultiply [{matmult.cpp}{32}]¡size¿=¡512¿]) that contains samples. For every sample in the container, the PC is resolved using GNU binutils [14] into the symbolic name of the function being executed at the time of the interrupt, the file location of the code, and the line number information. This is a best-effort operation. In the worst case, if we fail to acquire any symbolic information, "UNRESOLVED" is displayed with the option to also output the address value. If stack unwinding is enabled, the unwound callpath to unresolved samples can potentially provide sufficient context to resolve a sample address to a meaningful library API call up the call stack. We have chosen to treat the sampling information for a single line of code, where known, as the smallest unit for presentation. This required us to accumulate the sampled metric for different addresses to their common unique symbolic information and line numbers.

For all of the samples recorded within the bounds of a unique event context, their accumulated metric total is represented as the inclusive metric for the associated context event. This inclusive metric should be an approximation of the exclusive metric of the actual probed event. Each uniquely-resolved sample string is represented as a single new "SAMPLE" event, in the form of a function name by file path name by line number triple. "SAMPLE" events are always leaf nodes in the hybrid profile. With call stack unwinding enabled, "UNWIND" event nodes represent call sites that eventually end with a sample event as part of a functional call stack. Depending on the unwinding depth, the collection of these call stack chains form a forest with each tree's root node as a direct child of the appropriate sample parent.

Interestingly, TAU's existing parallel profile output format was able to be used directly for representing the new hybrid profile information because event names are represented as strings and the annotations above (CONTEXT, SAMPLE, UNWIND, UNRESOLVED) could be easily encoded. TAU's parallel profile output procedures could also be used directly. TAU can produce a profile file for each thread of execution or a single merge file for all thread profiles. The *ParaProf* parallel profile analysis tool has been improved to display hybrid profiles. The example below and in Section §IV give some demonstration of the types of views ParaProf provides. One feature ParaProf provides is a summary aggregation of samples within a routine, displaying this information with the "SUMMARY" label in the profile.

### D. Benefits of Hybrid Profiling in TAU

Hybrid parallel profiling brings significant benefits to a purely probe-based performance tool like TAU. It is challenging for probes to observe the performance of routines that take little time per call. The overhead of probe execution affects the accuracy of small measurements. Furthermore, the overhead for high-frequency probes accumulates throughout the program, intruding on performance behavior. The only recourse for TAU previously was to identify those offending probe events and disable them. In this case, with hybrid profiling, it is possible to see lightweight program execution with samples, while contextualizing the samples relative to the events still being probed. In addition, the integration of sampling brings an ability to see finer-grained projection of performance to regions of code.

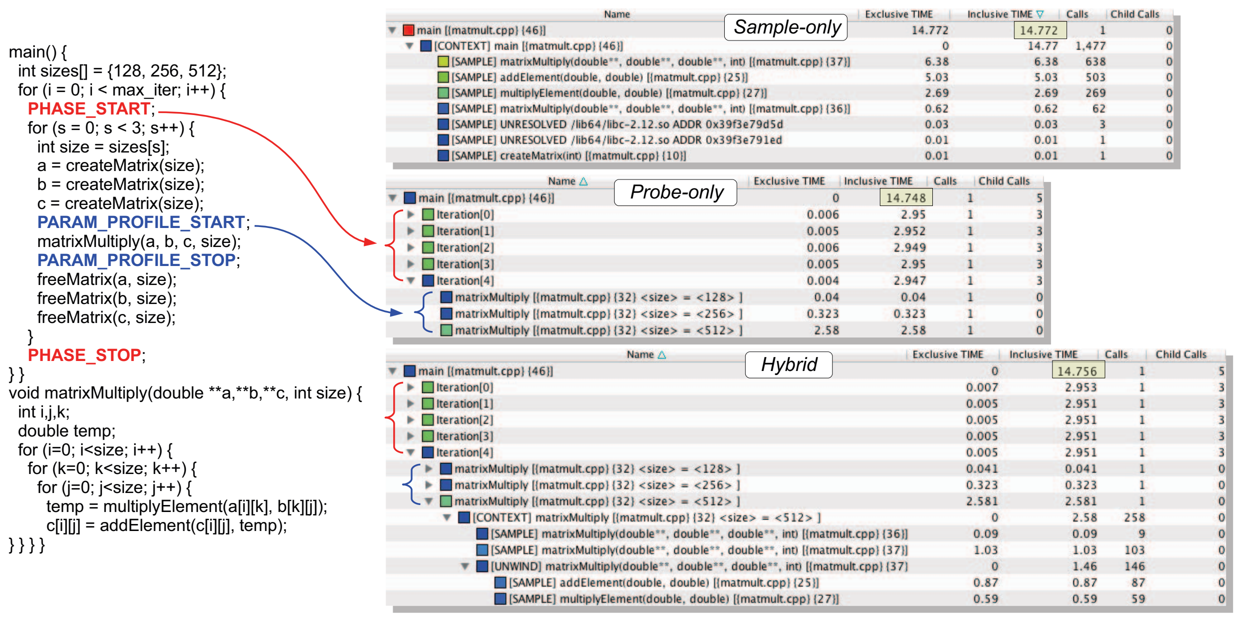
There are unique features of TAU's performance measurement methodology that are enhanced in hybrid profiling. Basically, an event in TAU is represented by a name and a probe is a point in a program's execution where an event is seen and measured. If TAU sees a new event name at a probe, it creates a new event. Thus, a variety of events can be created to represent the execution by just introducing new names. Standard events based on program structure, such as routines and loops, can be combined with more abstract events based on program phases, parameters, time, and program state. In this way, program execution semantics can be captured in the profiling measurements. Because event context is used to partition the samples obtained in hybrid profiling, the event semantics extend to the hybrid data. This is demonstrated in the example below.

Another important advantage of hybrid profiling is found in TAU's support for multi-language applications, particularly those involving high-level scripting, as with Python. The problem with pure sampling in these scenarios is the need to fully unwind the callstack to see high-level Python statements. The middle portions of the callstack reflects the inner workings of the Python runtime system, which might be of little interest to the user. In general, hybrid profiling can be used to do away with unnecessary callstack detail and still retain performance attribution to high-level events.

### E. Simple Example

Figure 2 shows a simple sequential example that demonstrates hybrid profiling. On the left is a program that multiplies two matrices, $a$ and $b$, for 3 different matrix sizes, $128x128$, $256x256$, and $512x512$. Little routines to multiply and add elements are used in the matrix multiplication code. The matrix creation and free routines use memory allocation and free functions internally. The outer loop repeat the matrix multiple 5 times, just to make the program more interesting. The macros `PHASE_START`, `PHASE_STOP`, `PARAM_PROFILE_START`, and `PARAM_PROFILE_STOP` are used to enable TAU probe instrumentation.

Consider a sampling-only profile of the program. TAU hybrid profiling can be controlled to enable this option, with only `main` instrumented. The top screenshot displays the result from an execution of the program on a Intel Xeon X5650 compute node at 2.67GHz, with 72 GB memory. ParaProf's unfolded call path display is used to show where the samples occur relative to the calling routines and call sites. We have exposed the samples for the `matrixMultiply` call. Notice that we can see samples occurring in the small functions along



```
main() {
  int sizes[] = {128, 256, 512};
  for (i = 0; i < max_iter; i++) {
    PHASE_START;
    for (s = 0; s < 3; s++) {
      int size = sizes[s];
      a = createMatrix(size);
      b = createMatrix(size);
      c = createMatrix(size);
      PARAM_PROFILE_START;
      matrixMultiply(a, b, c, size);
      PARAM_PROFILE_STOP;
      freeMatrix(a, size);
      freeMatrix(b, size);
      freeMatrix(c, size);
    }
    PHASE_STOP;
} }
void matrixMultiply(double **a,**b,**c, int size) {
  int i,j,k;
  double temp;
  for (i=0; i<size; i++) {
    for (k=0; k<size; k++) {
      for (j=0; j<size; j++) {
        temp = multiplyElement(a[i][k], b[k][j]);
        c[i][j] = addElement(c[i][j], temp);
} } } }
```

| Name | Exclusive TIME | Inclusive TIME ▽ | Calls | Child Calls |
|---|---|---|---|---|
| main [{matmult.cpp} {46}] | 14.772 | 14.772 | 1 | 0 |
| [CONTEXT] main [{matmult.cpp} {46}] | 0 | 14.77 | 1,477 | 0 |
| [SAMPLE] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {37}] | 6.38 | 6.38 | 638 | 0 |
| [SAMPLE] addElement(double, double) [{matmult.cpp} {25}] | 5.03 | 5.03 | 503 | 0 |
| [SAMPLE] multiplyElement(double, double) [{matmult.cpp} {27}] | 2.69 | 2.69 | 269 | 0 |
| [SAMPLE] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {36}] | 0.62 | 0.62 | 62 | 0 |
| [SAMPLE] UNRESOLVED /lib64/libc-2.12.so ADDR 0x39f3e79d5d | 0.03 | 0.03 | 3 | 0 |
| [SAMPLE] UNRESOLVED /lib64/libc-2.12.so ADDR 0x39f3e791ed | 0.01 | 0.01 | 1 | 0 |
| [SAMPLE] createMatrix(int) [{matmult.cpp} {10}] | 0.01 | 0.01 | 1 | 0 |

| Name △ | Exclusive TIME | Inclusive TIME | Calls | Child Calls |
|---|---|---|---|---|
| main [{matmult.cpp} {46}] | 0 | 14.748 | 1 | 5 |
| Iteration[0] | 0.006 | 2.95 | 1 | 3 |
| Iteration[1] | 0.005 | 2.952 | 1 | 3 |
| Iteration[2] | 0.006 | 2.949 | 1 | 3 |
| Iteration[3] | 0.005 | 2.95 | 1 | 3 |
| Iteration[4] | 0.004 | 2.947 | 1 | 3 |
| matrixMultiply [{matmult.cpp} {32} <size> = <128> ] | 0.04 | 0.04 | 1 | 0 |
| matrixMultiply [{matmult.cpp} {32} <size> = <256> ] | 0.323 | 0.323 | 1 | 0 |
| matrixMultiply [{matmult.cpp} {32} <size> = <512> ] | 2.58 | 2.58 | 1 | 0 |

| Name △ | Exclusive TIME | Inclusive TIME | Calls | Child Calls |
|---|---|---|---|---|
| main [{matmult.cpp} {46}] | 0 | 14.756 | 1 | 5 |
| Iteration[0] | 0.007 | 2.953 | 1 | 3 |
| Iteration[1] | 0.005 | 2.951 | 1 | 3 |
| Iteration[2] | 0.005 | 2.951 | 1 | 3 |
| Iteration[3] | 0.005 | 2.951 | 1 | 3 |
| Iteration[4] | 0.005 | 2.951 | 1 | 3 |
| matrixMultiply [{matmult.cpp} {32} <size> = <128> ] | 0.041 | 0.041 | 1 | 0 |
| matrixMultiply [{matmult.cpp} {32} <size> = <256> ] | 0.323 | 0.323 | 1 | 0 |
| matrixMultiply [{matmult.cpp} {32} <size> = <512> ] | 2.581 | 2.581 | 1 | 0 |
| [CONTEXT] matrixMultiply [{matmult.cpp} {32} <size> = <512> ] | 0 | 2.58 | 258 | 0 |
| [SAMPLE] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {36}] | 0.09 | 0.09 | 9 | 0 |
| [SAMPLE] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {37}] | 1.03 | 1.03 | 103 | 0 |
| [UNWIND] matrixMultiply(double**, double**, double**, int) [{matmult.cpp} {37} | 0 | 1.46 | 146 | 0 |
| [SAMPLE] addElement(double, double) [{matmult.cpp} {25}] | 0.87 | 0.87 | 87 | 0 |
| [SAMPLE] multiplyElement(double, double) [{matmult.cpp} {27}] | 0.59 | 0.59 | 59 | 0 |

Fig. 2. TAU hybrid profiling is shown for a simple sequential example (left). Profiles from executions with only samples (top) and only probes (middle) are compared to hybrid profiling. The runtime of the program without measurement was 14.751 seconds. The column "Calls" shows the number of calls for timers, and the number of samples for samples. Inclusive and exclusive times for samples are expected to be the same value.

with their routine ancestry. This verifies that sampling is working properly. However, what is not captured is the relationship of the samples with the matrix sizes and with the outer iterations.

Suppose we want to see these relationships. A probe-based profile alone would lose the samples entirely, as seen in the middle screenshot. We do see how TAU can generate events corresponding to loop iterations and specialize according to the calling parameters. However, only when hybrid profiling is enabled do we see the complete picture in the bottom screenshot. TAU is able to fully associated samples with where they occur relative to the active event context. Because iteration phases and parametrized routines translate to unique event names, new event contexts result. Thus, similar information would be seen if we unfolded the other iteration phases and matrix sizes.

## IV. Results

To evaluate hybrid profiling in parallel programs, we first tested it with the NAS Parallel Benchmarks (NPB) [15], focusing on overhead versus information content for different profiling alternatives. We then applied hybrid profiling to a scalable ocean modeling simulation to investigate the causes for performance inefficiencies in computation and communication routines.

### A. NAS Parallel Benchmarks Overview

The OpenMP version of the NPB (NPB-3.2.1 OpenMP) provided us with a good cross-section of codes to compare profiling methods (sampling only, probe only, hybrid), particularly with respect to their measurement overhead. Our goal was to characterize the overhead against uninstrumented (clean) runs. Probed runs used full and selective instrumentation of routines. Full instrumentation represents a profile where every function in the benchmark is instrumented with a timer. Selective instrumentation is performed by analyzing a full instrumentation profile to remove lightweight routines that are called frequently, and introduce excessive measurement overhead. Hybrid uses the same instrumentation as selective with sampling turned on and unwinding limited to a known degree. That is, the unwinding is stopped when an entry in the program stack for the current sample matches an entry in the program stack captured at the current timer entry. In general, we expect overhead characteristics to be code dependent. Specifically, we are looking to see what the relative overhead of adding hybrid support would be compare to sampling and probing alone. The benchmarks were compiled and executed on a single node of the ACISS cluster at the University of Oregon [**?**] with two Intel(R) Xeon(R) X5650 2.67GHz 6-core CPUs and 72GB of memory.

Table I gives the execution time results for 4 of the NPB programs. All show speedup as the number of

TABLE I
NAS PARALLEL BENCHMARKS OVERHEADS. THE *Clean* COLUMN REPRESENTS THE TIME TO EXECUTE THE BENCHMARK WITH NO MEASUREMENT. THE *Sampling* COLUMN REPRESENTS THE TIME TO EXECUTE THE BENCHMARK USING TAU SAMPLING ONLY. THE *Select* COLUMN REPRESENTS THE TIME TO EXECUTE USING SELECTIVE INSTRUMENTATION. THE *Hybrid* COLUMN REPRESENTS THE EXECUTION TIME WHEN USING THE HYBRID PROFILING TECHNIQUE. THE *Full* COLUMN REPRESENTS THE EXECUTION TIME WHEN PERFORMING FULL INSTRUMENTATION.

| **BT** | *Clean* | *Sampling* | *Select* | *Hybrid* | *Full* |
|---|---|---|---|---|---|
| 1 | 246.20 | 248.71 | 250.61 | 244.97 | 2355.43 |
| 2 | 125.24 | 125.38 | 125.82 | 124.95 | 1644.89 |
| 4 | 64.90 | 64.67 | 65.26 | 64.49 | 875.45 |
| 8 | 37.65 | 37.77 | 37.99 | 38.19 | 1033.80 |
| 12 | 30.15 | 31.37 | 30.43 | 30.48 | 1002.12 |

| **EP** | *Clean* | *Sampling* | *Select* | *Hybrid* | *Full* |
|---|---|---|---|---|---|
| 1 | 61.46 | 64.40 | 61.43 | 63.17 | 61.89 |
| 2 | 30.96 | 31.81 | 30.88 | 31.76 | 31.03 |
| 4 | 15.53 | 16.01 | 15.51 | 15.90 | 15.58 |
| 8 | 8.10 | 8.34 | 8.11 | 8.33 | 8.15 |
| 12 | 5.44 | 5.54 | 5.43 | 5.55 | 5.42 |

| **FT** | *Clean* | *Sampling* | *Select* | *Hybrid* | *Full* |
|---|---|---|---|---|---|
| 1 | 51.68 | 51.24 | 52.62 | 51.43 | 56.46 |
| 2 | 26.62 | 26.68 | 26.83 | 26.63 | 29.44 |
| 4 | 14.00 | 14.06 | 14.01 | 13.95 | 15.27 |
| 8 | 9.42 | 9.57 | 9.42 | 9.50 | 9.96 |
| 12 | 8.76 | 9.05 | 9.09 | 9.12 | 9.76 |

| **LU-HP** | *Clean* | *Sampling* | *Select* | *Hybrid* | *Full* |
|---|---|---|---|---|---|
| 1 | 248.02 | 278.16 | 253.27 | 280.81 | 251.80 |
| 2 | 121.24 | 125.90 | 125.75 | 125.57 | 127.26 |
| 4 | 64.22 | 68.47 | 69.24 | 68.70 | 69.64 |
| 8 | 37.81 | 42.43 | 43.52 | 44.05 | 45.11 |
| 12 | 29.92 | 34.30 | 36.26 | 36.16 | 37.61 |

threads increases in different modes, except for BT(Full). We will get back to this momentarily. The next general result is that hybrid and selective are relatively close in time for parallel execution. This is encouraging since it implies that we can acquire additional information from sampling at low cost. There are interesting behaviors in the 1 thread timings. Although 5 tests were run for each case and the minimum time taken, 1-thread runs can be sensitive to measurement, leading to unexpected results, such as BT(Clean) and FT(Clean) being slower than their hybrid versions, and LU-HP(Full) being faster than LU-HP(Select). It is interesting to see that there are cases where sampling can result in higher overheads versus probe measurements. This would usually be the case when there were few routines instrumented, such as in EP. However, the sampling runs here are also unwinding all the way in order to provide attribution. Here hybrid can be beneficial for filling in the gaps and providing context to limit unwinding.

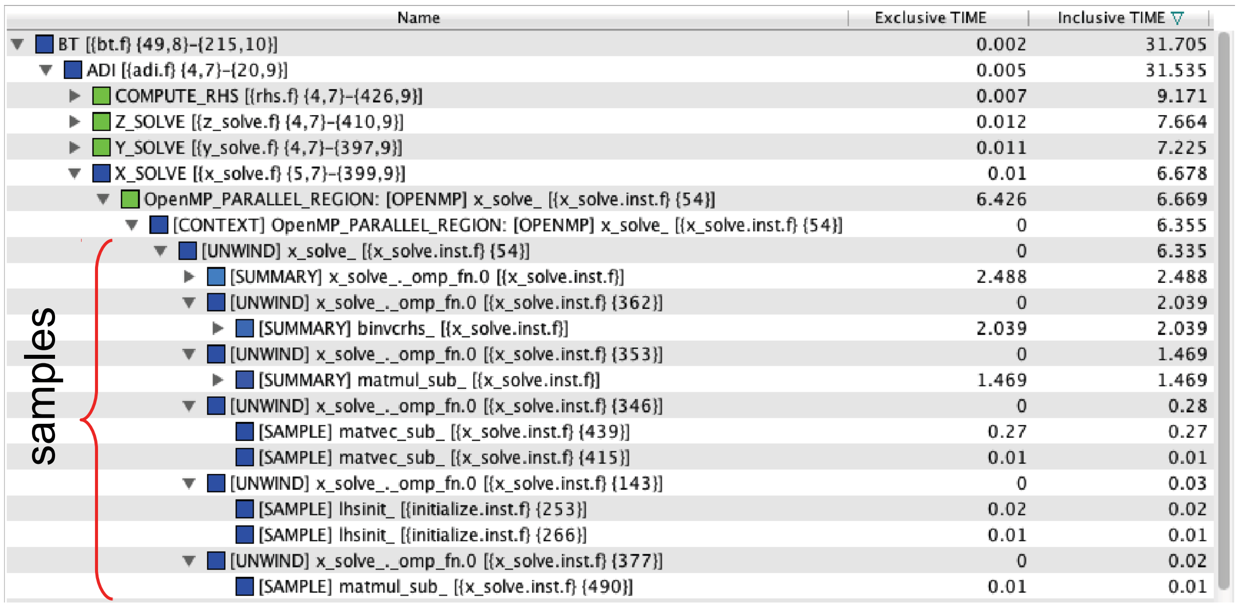
The Block Tri-diagonal (BT) experiments are demonstrative of the problems naïve, full instrumentation can cause. In cases such as these, selective instrumentation is necessary to reduce overhead, but it comes at the potential loss of performance detail. The main computation phases of BT solve along the X, Y, and Z dimensions and a selective instrumentation approach can observe these events with reasonable overhead. However, each phase calls several smaller routines many times. In order to associate the performance of these routines with the phases, we used hybrid profiling as shown in Figure 3. The `Z_SOLVE`, `Y_SOLVE`, and `X_SOLVE` events are TAU phases, providing context to the `OpenMP_PARALLEL_REGION` event coming from the OpenMP GOMP runtime system instrumentation. What is important to observe is the exposure of the lower-level routines now possible, for very little additional cost.

### B. MPAS-Ocean

The Model for Prediction Across Scales (MPAS) [9] is a framework project jointly developed by the National Center for Atmospheric Research (NCAR) and Los Alamos National Lab (LANL) in the United States. The framework is designed to perform rapid prototyping of single-component climate system models. Several models have been developed using the MPAS framework. MPAS-Ocean [11] is designed to simulate the ocean system for a wide range of time scales and spatial scales from less than 1 km to global circulations.

Like other MPAS simulations, MPAS-Ocean is developed in Fortran using MPI for large scale parallelism. MPAS-Ocean has been ported to several architectures and compilers, and in an effort to increase concurrency and efficiency on a wider range of large-scale distributed systems with multicore nodes OpenMP worksharing regions have been introduced. Many systems also provide vector instruction capability, further increasing the potential for concurrency. The developers have recently restructured key aspects of the code and annotated them with compiler pragmas to encourage the use of vector optimizations with the Intel compiler.

The MPAS-Ocean application had previously been instrumented with internal timing functions. TAU was integrated by replacing the timing infrastructure with TAU timers. Linking in TAU also provided measurement of MPI communication through the PMPI interface, OpenMP through library tool support, and hardware counter data using PAPI. The application instrumentation was mostly at a high level, encapsulating key phases in the simulation model. The application timers indicated that the ocean tendency tracer computation ran significantly faster ($\sim 19\%$ faster) with the code restructuring and vector instructions, along with a mysterious greater reduction in MPI communication time. Hybrid measurement was used on Hopper, a Cray XE6, to gain deeper insight into what was causing the performance improvement. The addition of samples should provide



| Name | Exclusive TIME | Inclusive TIME ▽ |
|---|---|---|
| ▼ 🟦 BT [{bt.f} {49,8}-{215,10}] | 0.002 | 31.705 |
| ▼ 🟦 ADI [{adi.f} {4,7}-{20,9}] | 0.005 | 31.535 |
| ▶ 🟩 COMPUTE_RHS [{rhs.f} {4,7}-{426,9}] | 0.007 | 9.171 |
| ▶ 🟩 Z_SOLVE [{z_solve.f} {4,7}-{410,9}] | 0.012 | 7.664 |
| ▶ 🟩 Y_SOLVE [{y_solve.f} {4,7}-{397,9}] | 0.011 | 7.225 |
| ▼ 🟦 X_SOLVE [{x_solve.f} {5,7}-{399,9}] | 0.01 | 6.678 |
| ▼ 🟩 OpenMP_PARALLEL_REGION: [OPENMP] x_solve_ [{x_solve.inst.f} {54}] | 6.426 | 6.669 |
| ▼ 🟦 [CONTEXT] OpenMP_PARALLEL_REGION: [OPENMP] x_solve_ [{x_solve.inst.f} {54}] | 0 | 6.355 |
| ▼ 🟦 [UNWIND] x_solve_ [{x_solve.inst.f} {54}] | 0 | 6.335 |
| ▶ 🟦 [SUMMARY] x_solve_._omp_fn.0 [{x_solve.inst.f}] | 2.488 | 2.488 |
| ▼ 🟦 [UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {362}] | 0 | 2.039 |
| ▶ 🟦 [SUMMARY] binvcrhs_ [{x_solve.inst.f}] | 2.039 | 2.039 |
| ▼ 🟦 [UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {353}] | 0 | 1.469 |
| ▶ 🟦 [SUMMARY] matmul_sub_ [{x_solve.inst.f}] | 1.469 | 1.469 |
| ▼ 🟦 [UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {346}] | 0 | 0.28 |
| 🟦 [SAMPLE] matvec_sub_ [{x_solve.inst.f} {439}] | 0.27 | 0.27 |
| 🟦 [SAMPLE] matvec_sub_ [{x_solve.inst.f} {415}] | 0.01 | 0.01 |
| ▼ 🟦 [UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {143}] | 0 | 0.03 |
| 🟦 [SAMPLE] lhsinit_ [{initialize.inst.f} {253}] | 0.02 | 0.02 |
| 🟦 [SAMPLE] lhsinit_ [{initialize.inst.f} {266}] | 0.01 | 0.01 |
| ▼ 🟦 [UNWIND] x_solve_._omp_fn.0 [{x_solve.inst.f} {377}] | 0 | 0.02 |
| 🟦 [SAMPLE] matmul_sub_ [{x_solve.inst.f} {490}] | 0.01 | 0.01 |

Fig. 3. Hybrid profile of BT showing merged events and samples.

deeper insight into the uninstrumented regions of code which would explain the performance improvement.

In the tracer computation, two key phases of the advection phase were reduced in execution time, and two others to a lesser degree. These four phases compute high/low order horizontal/vertical flux and they make calls to one of the vectorized routines, `mpas_ocn_tracer_advection_mono_tend`. The sampled routine showed a significant decrease in average time spent in that function, as expected with four-way vectorization. Table II shows the reduction in various hardware measurements.

TABLE II
BEFORE AND AFTER VECTORIZATION OF `MPAS_OCN_TRACER_ADVECTION_MONO_TEND`.

| | Before | | After | |
|---|---|---|---|---|
| Metric | Mean | Max | Mean | Max |
| Time | 39.165s | 152.850s | 19.807 | 35.990s |
| TOT_INS | 2.65E10 | 8.74E10 | 1.77E10 | 3.69E10 |
| TOT_L1_DCM | 4.58E8 | 2.67E9 | 6.12E7 | 1.98E8 |
| FP_INS | 5.89E9 | 1.95E10 | 2.43E9 | 5.13E9 |

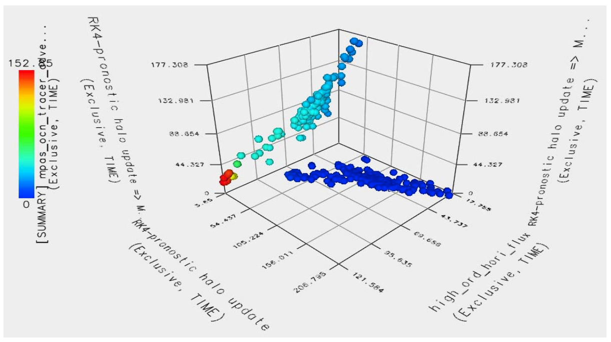
Table II also shows a very significant reduction in the worst performing thread, about 117 seconds. This explains the reduction in `MPI_Wait` synchronization times during the subsequent RK4-pronostic halo update. Because all of the computation is executed within one OpenMP region with guided scheduling and without explicit thread barriers, the reduced variability between thread computation times results in reduced synchronization times. Figure 4 shows a very strong correlation between the sampled computation and directly measured communication events.

## V. Related Work

The probe-based and sampling-based measurement methodologies have been implemented standalone in several parallel performance tools, but only a few integrate aspects of both in one form or another. Here we focus on representative related research work where some degree of hybridization is found.

The Unix *gprof* [16] tool is perhaps the earliest example where routines are instrumented to record call count and ancestor/descendant information, with sampling used to capture a flat time profile. The attribution of execution time along the callgraph is done by statistical distribution based on parent-child call counts. Originally, gprof only worked with sequential programs, but more modern versions support multi-threaded profiling. Whereas gprof provides basic functionality, *HPC-Toolkit* [2] is probably the most sophisticated parallel performance tool that relies almost exclusively on sampling. It collects full callpath information with optimized callpath resolution based on calling context trees. HPC-Toolkit can produce both profiles and traces of scalable parallel applications. Its ability to attribute causes of performance problems by “blame shifting” benefits from HPCToolkit’s targeted instrumentation of certain execution states (via probe wrappers of resource-related routines) as a way to enhance context information.

The work by Servat et al. [5], [6] is an interesting hybrid approach that uses computation region instrumen-


(a) Without vector instructions.

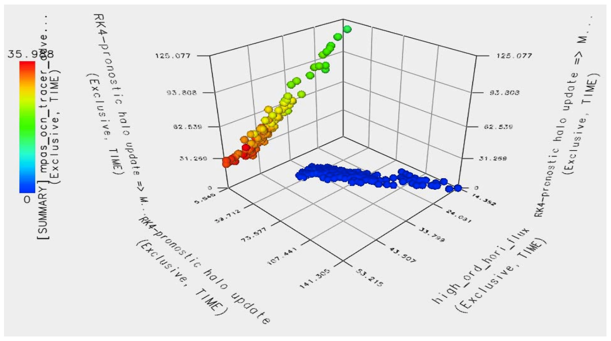

(b) With vector instructions.

Fig. 4. Strong positive correlation between computation and synchronization in MPAS-Ocean. The depth and color represent the time spent in `high_ord_hori_flux` and the sampled `mpas_ocn_tracer_advection_mono_tend` routine, while the height and width represent the time spent in `MPI_Wait` and the halo update, respectively. With vector instructions the correlations remain, even strengthen, but the observed ranges are smaller, as shown in Table IV-B.

tation (i.e., probes between consecutive MPI exit/entry points) together with sampling for computing fine-grained performance for representative region execution as determined by clustering. The approach gathers the probes and samples in a trace and "folds" the collected samples into their respective synthetic region by preserving their relative metrics (time, hardware counters) to show high-fidelity region computing evolution.

Our earlier research on hybrid performance measurement [4] was inspired by Servat et al.'s techniques. Here we integrated a sampling mechanism with the TAU measurement infrastructure with the purpose of overcoming the inherent probe-based limitations of observing lightweight events and fine-grained performance behavior. It was the first real example of an integrated hybrid measurement system in that it allowed both methods to be fully utilized and the analysis to be merged. Because traces were produced, we could effectively capture equivalent performance data to Servat et al., but with more event detail. However, tracing was the only means for producing data. Providing general hybrid profiling is harder and is the focus of our present work.

The research work of Szebenyi et al. [7] is closest to our efforts. They implemented a hybrid profiling technique in the Scalasca tool where only MPI events were measured with probes and the rest of the computation profiled with sampling. (Interestingly, this is similar to the approach of Servat et al. above, except with profiling and minus the clustering and folding.) The work advanced the efficiency of obtaining callstack information at MPI entry events and reconciling performance data gathered by the two methods. Our contribution in this paper essentially extends the MPI-based hybrid profiling to include any event, thereby allowing for more refined event-sample association without the need to generate execution traces. The callstack unwinding optimizations they developed are easily incorporated in our TAU implementation.

## VI. Conclusion and Future Work

A general hybrid parallel profiling technique has been designed to merge sampling and probing measurement techniques in a single integrated tool. The approach was implemented in the TAU performance system and applied to sequential and parallel codes to demonstrate the functionality available and how the tool might be applied in practice. Our work builds on a research history where steps towards hybrid measurement have demonstrated important benefits, but a full parallel profiling solution has not been realized. The main research contribution we bring is in the annotation of runtime samples with event context, allowing the probed events to form the skeleton for sample storage and profile measurements. Furthermore, the hybrid parallel profiling data we are able to obtain can be processed by a powerful parallel profile analysis tool. ParaProf can show fully integrated views of profile and sample information for parallel multi-threaded and message passing programs. The MPAS-O application was given as a example of performance problems that routinely appear for which a hybrid approach can help to uncover.

There are several directions for future research and development we are pursuing. The current implantation of hybrid parallel profiling in TAU provides the user with various options to control the degree of measurement:

- Probing: select events, enable event path profiling for a specified depth
- Sampling: set timer interrupt period, set depth call stack unwinding

Also, the user can capture hardware counters for both events and samples. These options all interact to determine the level of performance observation. However, there are scenarios where we might want to use some dynamic hybrid control in order to improve the measurement efficiency. For instance, suppose we wanted to unwind the call stack for a sample until we encounter the parent of the last probed event. The idea is to limit the degree of unwinding by using what is known about the event's routine ancestry. While this can reduce the time spent unwinding, it requires a means to determine an event's parent quickly when the event is entered. Unfortunately, our current approach is too inefficient. However, we believe that the optimizations discussed in the work by Szebenyi et al. [7] can be applied to this problem. (Note, an event's routine ancestry is also useful for contextualizing the event. We plan to use the solution approach to develop event callsite profiling.)

The hybrid techniques we have developed can be directed towards developing a "folding" technique online. Folding is useful in iterative applications where an iteration timer is inserted into the application that indicates the beginning and end of an iteration. The basic idea is to capture samples that occur between two paired (start/stop) probe events. However, instead of recording the sample with respect to its PC in the sample profile, the time since the start event is used. The samples from all iterations can be sorted by their deltas values and "folded" into a synthetic iteration, in order to provide a higher resolution sampling of the synthetic iteration with a lower sampling frequency. Servat et al. [6] post-process a sample trace, but we believe that is tracing is not absolutely necessary. Much of the hybrid profiling infrastructure is there to test this theory with minor modifications.

## VII. Acknowledgments

This work is supported by the United States National Science Foundation SI2-SSI grant 1148346 and United States Department of Energy SciDAC grant DE-SC0006723. The authors would like to thank Doug Jacobsen of Los Alamos National Laboratory for his contribution of the MPAS-Ocean profile data. Doug Jacobsen is supported by the US DOE Office of Science, Biological and Environmental Research program. ACISS was supported by an NSF OCI Major Research Instrumentation grant 0960354.

## References

[1] S. Shende and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 287–311, Summer 2006.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, N. Tallent, "HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs," *Concurrency and Computation: Practice and Experience*, Vol. 22, No. 6, pp. 685–701, 2010.

[3] R. Kufrin, "Perfsuite: An Accessible, Open Source Performance Analysis Environment for Linux," *Linux Cluster Conference) (LCC)*, 2005.

[4] A. Morris, A. Malony, S. Shende, K. Huck, "Design and Implementation of a Hybrid Parallel Performance Measurement System," *International Conference on Parallel Processing* pp. 492–501, 2010.

[5] H. Servat, L. Germán, K. Huck, J. Giménez, J. Labartaa, "Framework for a Productive Performance Optimization," *Parallel Computing*, Vol. 39, Issue 8, pp. 336-351, August 2013.

[6] H. Servat, G. Llort, J. Giménez, J. Labart, "Detailed Performance Analysis using Coarse Grain Sampling," *Workshop on Productivity and Performance (PROPER)*, pp. 185–198, 2009.

[7] Z. Szebenyi, T. Gamblin, M. Schulz, B. de Supinski, F. Wolf, B. Wylie, "Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profling of MPI Programs," *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 640-651, May 2011.

[8] A. Malony, J. Mellor-Crummey, S. Shende, "Methods and Strategies for Parallel Performance Measurement and Analysis: Experiences with TAU and HPCToolkit," D. Bailey, R. Lucas, S. Williams (Eds.), in *Performance Tuning of Scientific Applications*, pp. 49–86, CRC Press, New York, 2010.

[9] MPAS: Model for Prediction Across Scales, http://mpas-dev.github.io, Los Alamos National Lab and the University Corporation for Atmospheric Research, 2013.

[10] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, "Using PAPI for Hardware Performance Monitoring on Linux Systems", in *Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*,2001.

[11] T. Ringler, M. Petersen, R.L. Higdon, D.W. Jacobsen, P.W. Jones, M. Maltrud, "A Multiresolution Approach to Global Ocean Modeling," in *Ocean Modeling*, v. 69, pp. 211–232, Sept. 2013.

[12] The libunwind project, http://www.nongnu.org/libunwind/, 2013.

[13] StackwalkerAPI, http://www.dyninst.org/stackwalker Dyninst Project, 2013.

[14] GNU Binutils, http://www.gnu.org/software/binutils/ GNU, 2013.

[15] NAS Parallel Benchmarks, http://www.nas.nasa.gov/publications/npb.html NASA Advanced Supercomputing Division, 2013.

[16] S. Graham, P. Kessler, M. Mckusick, "Gprof: A call graph execution profiler", in *SIGPLAN Not.*, v. 17, pp. 120–126, 1982.

[17] ACISS, http://aciss.uoregon.edu/ University of Oregon, 2013.