LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-468224

Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs

Z. Szebenyi, T. Gamblin, M. Schulz, B. de Supinski, F. Wolf, B. Wylie

January 31, 2011

IPDPD 2011
Anchorage, AK, United States
May 16, 2011 through May 20, 2011

Disclaimer

Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs

Zoltán Szebenyi*‡, Todd Gamblin§, Martin Schulz§, Bronis R. de Supinski§, Felix Wolf*†‡, Brian J. N. Wylie*

* Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany
† German Research School for Simulation Sciences, 52062 Aachen, Germany
‡ RWTH Aachen University, 52056 Aachen, Germany
§ Lawrence Livermore National Laboratory, CA 94550 Livermore, USA
{z.szebenyi, f.wolf, b.wylie}@fz-juelich.de
{tgamblin, schulzm, bronis}@llnl.gov

***Abstract*—We can profile the performance behavior of parallel programs at the level of individual call paths through sampling or direct instrumentation. While we can easily control measurement dilation by adjusting the sampling frequency, the statistical nature of sampling and the difficulty of accessing the parameters of sampled events make it unsuitable for obtaining certain communication metrics, such as the size of message payloads. Alternatively, direct instrumentation, which is preferable for capturing message-passing events, can excessively dilate measurements, particularly for C++ programs, which often have many short but frequently called class member functions. Thus, we combine these techniques in a unified framework that exploits the strengths of each approach while avoiding their weaknesses: We use direct instrumentation to intercept MPI routines while we record the execution of the remaining code through low-overhead sampling. One of the main technical hurdles mastered was the inexpensive and portable determination of call-path information during the invocation of MPI routines. We show that the overhead of our implementation is sufficiently low to support substantial performance improvement of a C++ fluid-dynamics code.**

I. Introduction

Effectively harnessing the many-fold parallelism available on modern supercomputers becomes increasingly challenging. In particular, meeting the performance expectations for programs running on today's complex hardware can require significant effort. We can reduce this effort through appropriate performance-analysis technology, such as performance profiles that measure the execution time spent in different parts of the program. State-of-the-art parallel profilers, such as HPC-Toolkit [1] and TAU [2], further reduce the effort through context-sensitive analysis that differentiates not only between different functions, but also between the call paths leading to those functions, which delineate the performance phenomena more precisely. In addition to time and hardware counters, parallel profilers may also acquire parallelization metrics such as message counts and the communication volume.

One can broadly distinguish between two profiling techniques: *sampling* and *direct instrumentation*. The first approach periodically samples the program counter as the program progresses to measure performance aspects statistically. From the program counter value, the profiler can easily derive the function the program was executing when the sample was initiated by an interrupt. We then estimate the fraction of the overall runtime spent in a given function by the fraction of samples that occur during it. In contrast, direct instrumentation (or *event-based* instrumentation), inserts hooks at function entry and exit points. This insertion can occur at levels ranging from the source code to the binary file or even the memory image [3]. These hooks allow the profiler to maintain a shadow stack at runtime. The stack stores the time at which a function is entered, which is subtracted from the time when the function returns. The accumulated differences precisely capture how much time was spent in each function. To attribute times to individual call paths, a call-path profiler can determine the current call path in two different ways: It can maintain a shadow function stack, again controlled by direct instrumentation; or it can use the technically more complex *stack unwinding*, sometimes also referred to as *stack walking* [4].

Sampling and direct instrumentation both have advantages. We can easily control measurement dilation under sampling by adjusting the sampling frequency. However, it delivers an incomplete picture, potentially missing critical events or providing inaccurate estimates. Moreover, because the timer interrupt can occur at arbitrary program locations, sampling complicates accessing not only the current call path, but any details of the program state, such as the arguments of the current function. As a result, many tools instead use direct instrumentation to capture communication metrics such as message payload sizes. The MPI profiling interface [5], which leverages direct instrumentation through interposition wrappers, reflects this insight. However, direct instrumentation can result in excessive measurement dilation if applied indiscriminately, which quickly becomes apparent in C++ programs, which often have many short but frequently called class member functions. While heuristics can balance the amount of direct instrumentation with the need to cover all relevant program regions, they often require additional program runs.

Our approach combines the two methods in a unified framework that exploits the strengths of each while avoiding their weaknesses. Specifically, we make the following contributions:

- The design of a call-path profiling technique that combines direct instrumentation of MPI routines with sam-

pling of all other program regions;
- Modifications of the call-path profiler of the Scalasca toolset [6] to implement this technique;
- An inexpensive and portable enhancement of the classic stack-unwinding mechanism that can identify the call paths of frequently called communication routines without incurring excessive overheads;
- A study of a fluid-dynamics code written in C++ that demonstrates how our profiling technique sufficiently reduces measurement intrusion to guide optimizations that improve overall performance by a factor of 11.6 (and reduce time for file writing 75-fold).

Overall, our approach captures accurate call-path profiles with a variety of communication metrics for a broad range of applications without requiring cumbersome function selection or expensive runtime filtering mechanisms.

The article is structured as follows. The next section presents our call-path profiling approach and its integration into Scalasca. In Section III, we conduct a detailed experimental comparison of our new method to Scalasca's traditional profiling options, giving evidence of cases for which it was the only way to achieve acceptable overhead. Section IV presents our study of the fluid-dynamics application.

II. Hybrid call-path profiling

Call-path profiling requires two ingredients: (i) a mechanism to determine the call path at a given point in time and (ii) a method to decide when such a point has arrived. In our approach, we use stack unwinding for the former and combine sampling with direct instrumentation for the latter.

We conceptually divide the execution of each MPI process into two disjoint, alternating phases – execution outside and inside the MPI library. Outside the MPI library, we use sampling, which is independent of the frequency of routine entries and exits and, thus, provides better control of runtime overhead. Inside the MPI library, we use direct instrumentation, which greatly facilitates calculation of metrics based on parameters of MPI routines, such as the total number of bytes sent. We can easily intercept MPI calls with PMPI wrappers by relinking or dynamic loading, thus avoiding complex code transformations required for some direct instrumentation mechanisms.

Stack unwinding is central to our approach. Because we use direct instrumentation only for MPI routines, maintaining a shadow stack with direct instrumentation of user code will not work correctly. Stack unwinding is essentially a stateless operation that can be applied at any time during program execution, and it allows us to determine the call path that leads to invocation of a particular MPI routine whenever it occurs. Thus, stack unwinding is triggered by two different classes of events, either when a timer interrupt occurs during computation or when the program enters an MPI routine.

Another distinction concerns the attribution of the time spent in these two states. We account for time spent outside MPI to individual call paths based on the frequency of samples that exhibit them. We account for time spent inside MPI based on entry and exit timestamps. Our solution combines the advantages of both techniques: rich and reliable MPI performance information including messaging statistics and a low overhead approximation of application performance elsewhere with no impact from small, frequently called functions.

We integrate our solution into Scalasca [6], a postmortem performance-analysis tool suitable for large-scale MPI codes. We primarily target reduction of measurement dilation during call-path profiling, particularly due to class-member functions in C++ applications. The previous version of the call-path profiler included in Scalasca relied exclusively on direct instrumentation to track the current call path and to determine the time spent in specific call paths. In addition to elapsed times and, optionally, hardware counter measurements throughout the program, Scalasca collects communications metrics, typically the number of messages and bytes sent and received.

Although source-code translators and binary instrumentation are also available, Scalasca most commonly employs automatic compiler instrumentation to insert hooks for direct instrumentation into user functions. Scalasca can dynamically filter out small but frequently called functions to lower the overhead of indiscriminate user-code instrumentation. These functions, which cause significant dilation but contribute little to the overall execution time, are placed on black lists that prevent them from invoking the code that was added during instrumentation. However, this approach usually requires an extra run of the application under full instrumentation to identify the functions to place on the black list. Our new hybrid approach eliminates this cost.

Our approach generically builds on third-party libraries to perform the stack-unwinding procedure, which provides increased portability and flexibility. We focus primarily on an implementation that is based on libunwind [7], an easy-to-use C library that supports a range of platforms and offers advanced features such as changing the instruction pointer, the stack pointer or different processor registers. The latter capability directly supports our stack unwinding optimizations. We demonstrate that our optimizations are independent of the platform or the unwinding library through x86_64 and PowerPC implementations based on StackwalkerAPI [8], a C++ library with similar functionality.

Our approach faces two challenges. First, we must reduce stack unwinding overhead in the presence of frequent MPI calls. Second, we must integrate sampling-based and event-based time measurements within the same experiment. We discuss these challenges in the following subsections.

A. Fast Call-Path Unwinding

Although our approach avoids excessive measurement dilation of user-code profiling through sampling, unwinding the call stack during every MPI call can still dilate measurements significantly if MPI calls are frequent. While we can easily adjust the sampling rate, the rate at which an application calls MPI functions is beyond our control. We therefore introduce several optimizations to lower the overhead of stack unwinding including non-trivial measures such as caching function start addresses and thunk stacks.

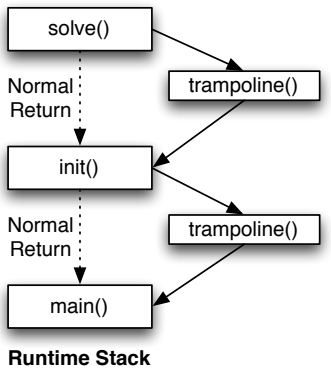
1) Unwinding only relevant MPI functions: In extreme cases, MPI call frequency exceeds the the sampling frequency of 100Hz that we use in our implementation by orders of magnitude. Many applications repeatedly invoke auxiliary MPI functions with insignificant execution times. For example, in the SPEC MPI2007 application *142.dmilc*, more than 99.5% of all MPI calls are to `MPI_Comm_rank`, while 68.4% in *147.l2wrf2* are to `MPI_Cart_shift`. We exploit Scalasca's ability to configure certain groups of MPI wrappers individually to turn off stack unwinding for a broad range of MPI functions that are not performance critical. However, we still count the occurrences of those calls with negligible overhead, which facilitates eliminating unnecessarily frequent ones.

2) Caching region identifiers: Using libunwind to determine the name of a function with a given start address incurs significant overhead. We also incur a high cost to map function names to region identifiers by which Scalasca uniquely identifies the functions internally. Thus, we implement a hash table that maps start addresses to their corresponding region identifiers, which eliminates the name lookup and string matching except for each function's first occurrence.

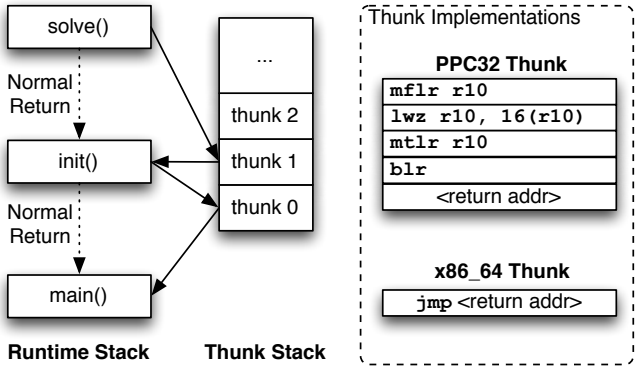
3) Caching start addresses: We must use another, even more expensive libunwind call to look up the start address of the function being executed based on the current value of the instruction pointer. Caching this information using a hash table is non-trivial because the timer interrupt can occur during execution of essentially any instruction, which would present a large set of keys for non-trivial applications. Fortunately, this problem only applies to the topmost stack frame. For all other frames (and the topmost stack frame when unwinding from MPI wrappers), the instruction pointer must refer to an instruction just after a function call site, corresponding to the return-pointer value of the function being called. Since the largest applications only have several thousand call sites, we can use a hash table to look up the function start address for most stack frames. We handle instruction addresses in the topmost stack frame (i.e., those the timer interrupts) with a separate, less efficient look-up structure, in which we check if the address falls in between the start and end address of a known function. Caching the start address is our most effective optimization, removing 90% of our unwinding overhead.

4) Light-weight thunk stacks: The optimizations that we have discussed so far either reduce the number of unwind operations or the time to perform actions related to a single stack frame. Our next optimization reduces the number of stack frames that we must examine to determine the full call path. Specifically, we detect when the current frame is the last element of a prefix of a previously unwound call path. Since we already know the prefix, we can avoid re-examining its stack frames. To mark prefix frames, we change each frame's return address as we unwind the stack. Based on these special return addresses, we can later identify marked frames and associate each with its corresponding prefix.

Changing bits in return addresses to mark frames could alter the program's control flow. Instead, we allocate a special region of contiguous memory to hold a *thunk stack*, shown



(a) Trampoline function.



(b) Thunk stack.

Fig. 1. Prefix optimization mechanisms.

in Figure 1(b). The thunk stack is composed of entries, or *thunks*, that mirror the state of the runtime stack when it was last unwound. Each time we walk over a stack frame during unwinding, we create a thunk in the stack that branches to that frame's return address. We then modify the original return address to point to the thunk. Since we know that instrumented return addresses will fall somewhere inside the thunk stack, we can identify them by comparing each return address to the thunk stack's location. Entries in the thunk stack are constant-size, so we can compute the depth of a particular prefix by subtracting the address of the bottom of the thunk stack from an instrumented return address.

Other tools use different mechanisms to perform the same optimization. HPCToolkit uses a lightweight trampoline, as shown in Figure 1(a) [9]. Instead of pointing return addresses to a stack, the topmost function of a prefix is instrumented to return into a trampoline function. The trampoline performs bookkeeping for profiling and tracks the frame in the dynamic call tree to which it currently points. Upon return, the trampoline installs itself in the next lower stack frame so that that frame's function will also call the trampoline when it returns.

Each individual thunk is simpler than the HPCToolkit trampoline function. Figure 1(b) shows our thunk implementations. On x86_64, the 64-bit jump performed by the thunk is a single instruction, and on PowerPC it only requires four instructions. Further, our thunks do not modify the stack as the trampoline does. We thus avoid signal safety issues within

our instrumentation code, as our sampling signal handler does not need to check whether it is within the trampoline to avoid write conflicts. Our instrumentation is confined within unwind routines and isolated from our thunks.

This arrangement decouples our optimization from the particular stack unwinder used. The trampoline approach requires that trampoline code can interpret stack frames and insert itself in the proper location in the next frame. Our thunks do not perform stack surgery and, thus, can be implemented separately from the main stack unwinding logic, which makes our implementations less complicated than tramploine-based implementations. Stack unwinding APIs only must support writing to return address locations as we unwind the stack, a feature that both libunwind and StackwalkerAPI provide.

Finally, our approach avoids the problem of non-local function exits. In languages that implement exception handling, a routine may return into a routine other than its caller. This event causes control flow to skip the return that would install the trampoline in a lower frame. The trampoline approach thus must instrument all non-local exits to routines, which requires more complex code analysis [10]. Our approach avoids this analysis by simply instrumenting all return addresses. Thus, our scheme already instruments the frames in the prefix of any frame that a non-local exit skips, whereas a trampoline may be skipped entirely by a non-local exit.

5) Counting call-path visits: Our thunks support efficient counting of the number of times that a call path is visited. If a stack-walk step encounters a function in which we have installed a thunk, then this function has not returned since the previous unwind. Had it returned and been called again, its return address would not point to the thunk. This count is only an estimate, because we do not necessarily take a sample at every execution of a given call path, and we do not instrument non-local routine exits. However, this count is a guaranteed lower bound. Further, a unique property of our solution increases the accuracy of this lower bound. We perform stack unwinds not only during arbitrary timer interrupts but also inside every PMPI wrapper. Thus, we frequently unwind call paths that lead to MPI functions. In most cases, these functions always call at least one MPI function every time they are executed, which means that our visit count is likely to be exact. Overall, we provide lower bounds for the visit counts of arbitrary call paths, and exact visit counts for most call paths leading to MPI functions. HPCToolkit similarly tracks the visit count, but does most of the bookkeeping work inside its trampoline, whereas we do this work within our unwind calls.

B. Hybrid Sampling Methodology

Our hybrid profiling approach combines two profoundly different measurement modes. This combination requires that we prevent undesirable interference between their underlying mechanisms. Further, we must integrate their measurements in a consistent and statistically sound manner.

1) Timer interrupts inside MPI: A possible interference between the two modes arises when timer interrupts occur during MPI calls. Although at first glance it may seem reasonable just to pause the timer whenever control is transferred to MPI, the high frequency of MPI calls makes this approach extremely expensive. Thus, we simply ignore interrupts that occur inside MPI functions. We explain below how we correctly account for ignored interrupts and how we avoid inaccuracies due to MPI calls, which are not part of the sampled population.

2) MPI calls inside sample intervals: While we directly measure the time spent inside MPI, we only statistically measure the time spent outside MPI (which we refer to as *computation* in the following). We must separate these two measurement realms so that we represent each as accurately as possible while maintaining the invariant that their sum equals the directly measured overall execution time.

Classic sampling methodology estimates the time t_c consumed by a certain call path c as:

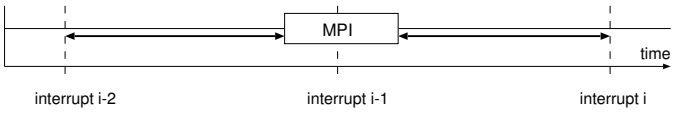
$$t_c = \frac{n_c}{n} * t$$

with t being the total execution time, n being the total number of samples, and n_c being the number of samples exhibiting call path c. We could calculate the execution time within a computational call path either by including the ignored samples into n, as if we were sampling everything, or to rescale the equation variables as if the entire execution consisted only of computation. Either choice could result in inconsistent timings due to the representation of overall MPI time in two different ways, directly via explicit wall-clock readings and indirectly via the number of interrupts that occur inside MPI.

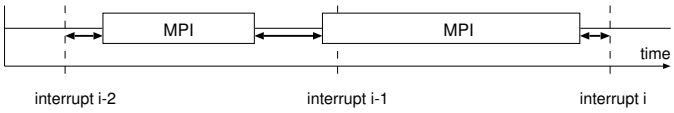
The frequency of MPI calls makes these solutions inadequate. Our experiments confirm that the two representations correspond well, as long as applications alternate between large contiguous phases of computation and communication. However, there are slight deviations when the frequency of MPI calls rises. Further, time assignments may be biased depending on when a call path primarily executes at runtime, because the MPI call frequency may differ not only between applications, but also within phases of the same application.

Classic sampling methodology assumes evenly distributed samples across the execution time with intervals between consecutive samples having about the same length. Our hybrid approach violates this assumption for two different yet related reasons. First, we drop timer interrupts that occur inside MPI calls. Second, the interval between two valid interrupts may include MPI calls, which influence the effective length of the interval. Depending on the duration of these MPI calls, the effective length of this interval can be shorter or longer than the normal timer interval. As illustrated in Figure 2, we define the *effective sample interval length* of a sample taken at interrupt i as the sum of the computation intervals since the last interrupt outside MPI ($i-2$ in the figure). We can calculate this length as the distance between i and $i-2$ minus the intervening time spent in MPI as determined by our direct measurements. Depending on the extent of MPI execution, the effective length can be greater than (Figure 2(a)) or less than (Figure 2(b)) the regular timer interval length.

Instead of excluding the MPI time at a global level by rescaling all measurements by the same factor, we use a more



(a) Effective sample interval length greater than normal.



(b) Effective sample interval length smaller than normal.

Fig. 2. The impact of MPI calls on interval lengths.

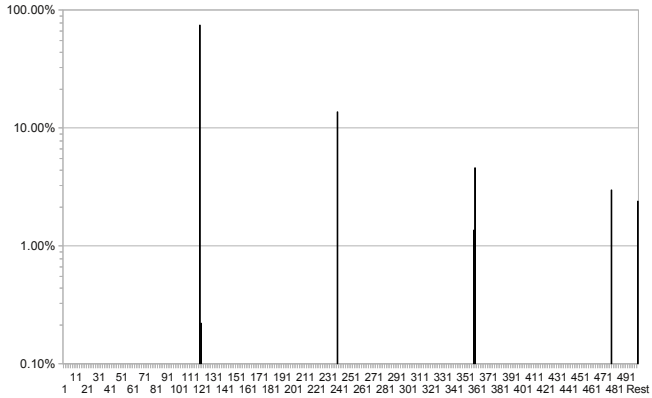
flexible solution that locally rescales the time attributed to a sample. This approach accounts for local fluctuations of the sampling frequency caused by MPI calls. Our solution assigns a weight that corresponds to the length of the effective sample interval to each sample taken outside MPI. Thus, we use a variable sampling frequency that assigns more weight to samples that are taken in regions with a lower *effective sampling frequency* and less weight to call paths in regions with a higher effective frequency. This approach ensures that effective sample intervals and MPI times add up to the overall execution time, preserving consistency as desired.

Figure 3 presents experimental evidence from the SPEC MPI2007 application *128.GAPgeofem* that we require our approach to dropped interrupts. The bins in the histogram in Figure 3(a) represent the sample interval before we subtract the time spent in MPI. 75% of the intervals have normal length, 14% are twice as long, 6% are three times as long, and the remaining 5% are more than three times as long. Figure 3(b) shows the histogram after we subtract the MPI times, resulting in the effective sample interval length. 34% of the intervals have exactly the normal length, which means that they did not include any MPI execution. Another 42% of the intervals have a length between zero and the normal timer interval length. These intervals have the normal length in the previous histogram, but included some MPI execution.

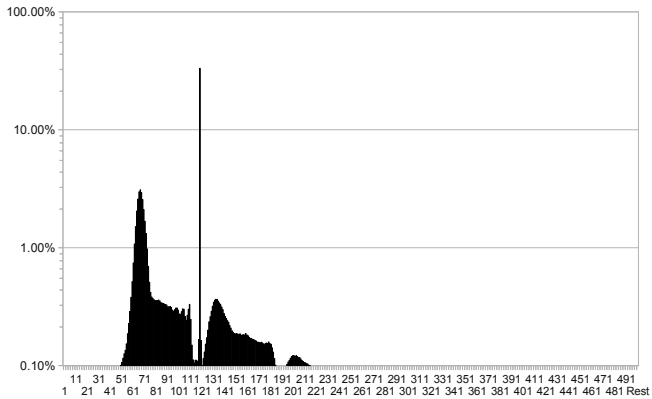
The peak of the histogram indicates that slightly less than half of the sample interval is typically spent in MPI for this application. We also see a dip to the left of the normal interval length as any given interval rarely includes only a small amount of MPI execution. Intervals that comprise the second peak of Fig. 3(a) appear to the right of the normal interval length in Fig. 3(b). These intervals, which must have included MPI execution, are now distributed similarly to the samples from the first peak, but at a different scale (on the logarithmic y-axis). The histogram shows that even though we allow variable and theoretically unbounded sample interval lengths, most of the interval lengths are very close to the normal timer interval. Having many arbitrarily long sample intervals would not invalidate our method, but would slow down its convergence.

C. Implementation Challenges

1) Compiler optimizations: Unwinding the call stack on the x86_64 architecture requires great care. Some compiler



(a) Sample interval length before subtracting MPI time.



(b) Sample interval length after subtracting MPI time.

Fig. 3. Logarithmic histograms of sample interval lengths with 0.0001s resolution buckets on the x-axis for *128.GAPgeofem*; the last bucket contains all intervals that do not fit into the first 500 buckets.

optimizations such as tail calls or even hand-written assembly code may confuse libunwind, leading to incorrect return values [11]. Especially when using thunks, bogus return values from the unwinder will almost certainly lead to a fatal error. To avoid such errors, we validate libunwind return values by ensuring that the preceding instruction was a call to the corresponding function. If it is not, we simply do not install the thunk. StackwalkerAPI did not exhibit this problem.

2) Signal safety: Each function called from our interrupt handler must be signal safe. However, we require that our signal handler has access to the full Scalasca measurement infrastructure, which is not signal safe. We solve this problem through guards to every entry point of the Scalasca library. The guards check a single atomic flag (`volatile sig_atomic_t`) and immediately return if it is set. After the guards, the flag is set, and it is unset at Scalasca's exit points, which eliminates race conditions within Scalasca.

3) StackwalkerAPI: Our approach required extensions to StackwalkerAPI to export information about the *location* of return addresses in unwound call paths. We worked with the StackwalkerAPI developers to add this functionality for x86_64 and PowerPC versions, which also allowed us to decouple our implementation from StackwalkerAPI.

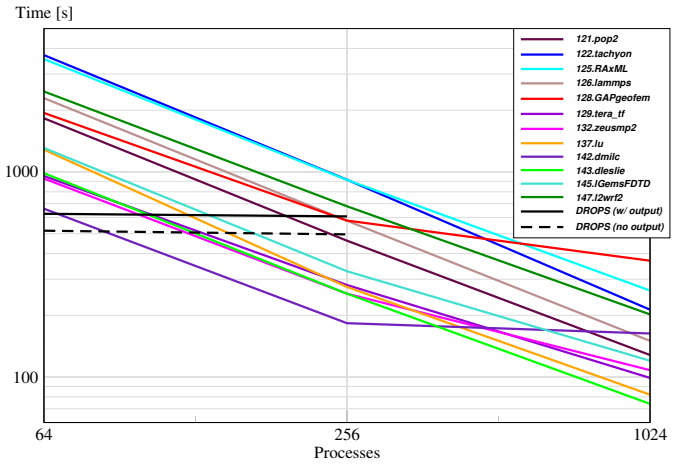


Fig. 4. Application execution times on *Juropa*.

III. Experimental Evaluation

We investigate the benefits of our techniques on a representative set of MPI applications, the twelve applications of the SPEC MPI2007 v2.0 benchmark suite [12] with large reference datasets. We also test with the *DROPS* application since *126.lammps* is the only C++ application in SPEC MPI2007. The *DROPS* CFD software package [13] is an object-oriented framework implemented in C++ that incurs prohibitive measurement overheads [14].

We compile all codes with the Intel 11.1 compilers and run them with ParaStation MPI 5.0 on the *Juropa* system, which has 2208 twin quad-core Intel Xeon X5570 (Nehalem-EP) compute nodes connected with Infiniband QDR and Sun Data Center 648-port switches [15].

Figure 4 shows wall-clock times for executions with 64, 256 and 1024 processes. The programs generally exhibit good scalability for fixed problem sizes: 1024-process runs complete in a few minutes. However, *DROPS* fails to scale even to 256 processes (even with file output disabled), while scalability of *142.dmilc* and *128.GAPgeofem* is poor after 256 processes.

We prefer longer measurement runs since run-to-run variation of several seconds (e.g., due to use of the shared file system) complicates performance comparisons. Thus, we focus on the 256-process executions. With this processor count, we measure each code three times and observe under 3% run-to-run variation in most cases. However, some executions took more than twice as long. To reduce the impact of this variability, we use the fastest of the three runs as the reference time for an uninstrumented execution, which we compare to runs with Scalasca to assess measurement overheads.

We compare two Scalasca measurement sets. The first set uses Scalasca's existing capabilities. The second set uses our new call stack unwinding approach. Both sets employ the Scalasca measurement library in its default runtime summarization mode, which produces an analysis report that integrates call-path profiles from each process. The measurements include a full complement of wrappers for MPI routines, enabling capture of `MPI_Init` and `MPI_Finalize` and all other MPI events according to its runtime configuration.

With the optimized application object files only relinked to include the Scalasca measurement library, the simplest measurement consists of a (flat) profile of MPI routines. We expect this measurement to have the lowest overhead, providing comprehensive MPI communication and synchronization statistics, albeit without call-path contexts. The rightmost (blue) set of bars in Figure 5 shows that, while *143.dleslie* execution took 3.5% longer than the uninstrumented reference, the other applications (including *DROPS*) were dilated less than 1%, confirming the low overhead of basic PMPI profiling with Scalasca. While unnecessary for these applications, we could disable groups of MPI events if required to reduce measurement overhead further.

A. Direct Instrumentation

The Scalasca instrumenter, when used as a prefix to the usual compile and link commands, configures the compiler to instrument the entry and exits from user-level source routines. Although details vary by compiler, almost all contemporary compilers offer such a capability. By observing these entry and exit events for routines, the Scalasca runtime library can track the current stack of instrumented routines as they execute and update its call-path profile accordingly on each process. As MPI events occur, delivered by the PMPI wrapper library, they naturally augment the call paths in the profile.

The leftmost (red) bars in Figure 5 show that nine of the applications have less than 4% measurement overhead, which is sufficiently low for most uses. While *125.RAxML* (8%) is borderline, *142.dmilc* (50%) and *122.tachyon* (237%) have excessive overhead. Finally, the 1770% overhead for *DROPS* confirms that it is extremely challenging for this approach.

When we instrument and measure every user-level source routine, high dilation can occur, particularly for small/short computational routines that execute frequently. We can reduce this overhead significantly by filtering such routines with the black-list approach discussed earlier. In this case, they are not included in the analysis report, as if the compiler inlined them. We typically only need to filter a few user-level source routines to reduce dilation. However, it is often convenient to filter all routines that are not executed on call paths to MPI routines (i.e., those engaged in local calculation).

These filters always reduce overhead, as the middle (green) set of bars in Figure 5 shows. They reduce overhead for *125.RAxML* to 1% and for *142.dmilc* to 4%. Although the *122.tachyon* overhead is down to 44%, it remains a concern: with all overhead due to a single routine, selective instrumentation could be employed to avoid instrumenting it entirely. *DROPS* measurement also benefits from a comprehensive filter (listing more than a thousand routines), yet the dilation of 190% may still result in undesirable distortion. We examine this issue in detail in Section IV. Thus, instrumentation of user-level source routines proves straightforward and effective for most but not all applications and is sometimes inconvenient.

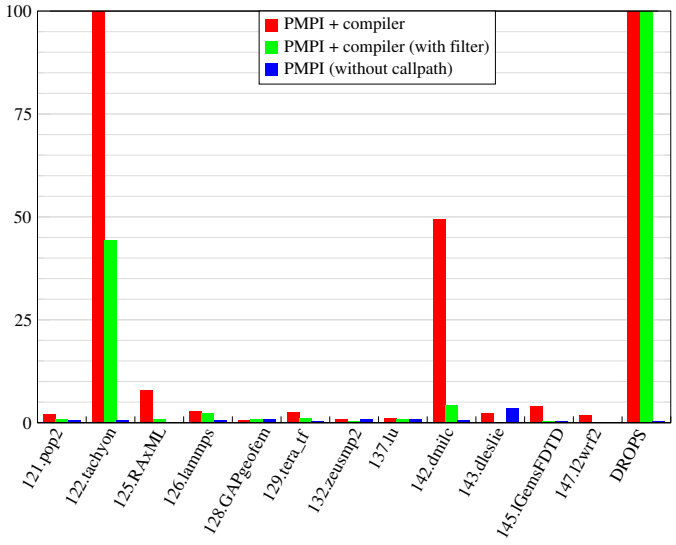


Fig. 5. Measurement overhead percentage for executions with compiler instrumentation of all user-level source routines (also employing filtering) with PMPI call-path profiling, and basic PMPI-only routine profiling.

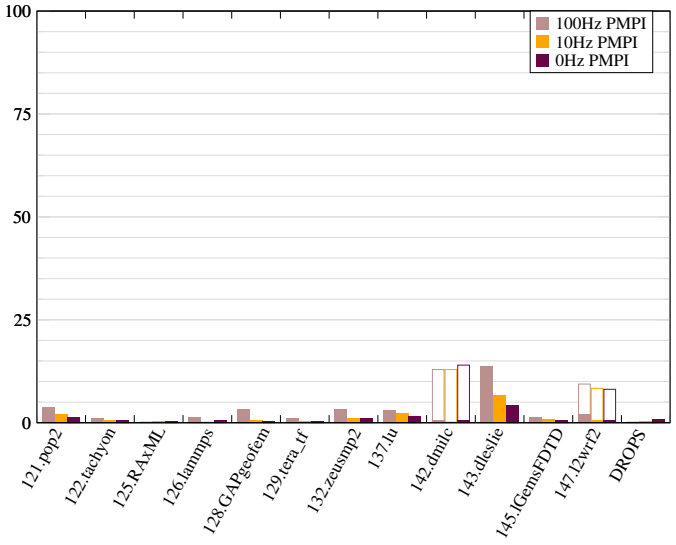


Fig. 6. Measurement overhead percentage with interval timer event samples and PMPI call-stack unwinding.

B. Stack Unwinding

We employ stack unwinding to determine the call path of PMPI events for executables without compiler instrumentation. The resulting call-path profile roughly corresponds to that produced by the filtering scenario that excludes all user-level source routines not on call paths to MPI operations. We expect small differences, e.g., when the compiler changes inlining decisions and from routines that the compiler does not instrument (such as Fortran interfaces to MPI library routines). The rightmost (dark brown) set of bars in Figure 6 shows the measured overhead for direct measurement of MPI routines only with no sampling. For most applications, overhead is under 2% (and under 1% for *DROPS*). Exceptions are *143.dleslie* (4%), *147.l2wrf2* (8%) and *142.dmilc* (14%).

The latter two codes incur more overhead primarily because they frequently invoke certain MPI routines that we do not really need to profile. The Scalasca measurement library can selectively disable measurement of such routines. Although this is unnecessary with direct instrumentation, the cost of frequent stack unwinding from such routines is prohibitive. Disabling the group containing `MPI_Comm_rank` for *142.dmilc* and this group plus the one that contains `MPI_Cart_shift` for *147.l2wrf2* reduces measurement overheads to under 1%. Figure 6 shows the best results with solid bars, and the two cases without this optimization with outlined white bars.

When an interval timer is configured to deliver interrupts at a specified rate to each process, we can use stack unwinding from these signals to augment the call-path profile with non-MPI call paths approximating the profiles produced by (unfiltered) compiler instrumentation. This approach includes additional call paths since signals occur during execution of uninstrumented routines (such as those in system libraries).

The leftmost (brown) and central (orange) sets of bars in Figure 6 are for measurements including one-hundredth of a second (100Hz) and one-tenth of a second (10Hz) timer interrupts. We find slightly higher measurement overhead as expected, but it remains below 4% for all but *143.dleslie*, for which it rises to 14% at 100Hz. Interestingly, measurement overheads decreased for *DROPS*.

We measured the time to perform stack unwinding from PMPI and non-PMPI events for each application, as Figure 7 shows for 10Hz and 100Hz. The PMPI unwind cost (lower/cyan bars) dominates in most cases, equivalent to over 6% of the execution time for *DROPS*, 3% for *128.GAPgeofem* and 2% for *143.dleslie*. Non-MPI unwind costs (middle/orange bars) are negligible at 10Hz and only over 1% at 100Hz for *147.l2wrf2*, *132.zeusmp2*, *137.lu* and *143.dleslie*.

Stack unwinding from PMPI or non-PMPI events sometimes fails, which we categorize as "failed unwinds" in the Scalasca profiles. These failures mostly occur within system libraries, specialized mathematical libraries, and the MPI library. The upper/purple bars in Figure 7 show that the proportion of failures is generally low and roughly consistent regardless of the sampling frequency. For eight of the SPEC MPI codes call-stack unwind failures correspond to under 0.2% of execution time, with the worst cases being *122.tachyon* (4%) and *147.l2wrf2* (7%). For *DROPS* it is also only 0.4%.

Overall, our new approach provides reliable and convenient comprehensive call-path profiling of PMPI and user-level routines. We observe negligible measurement overheads (well under 5%) for all test applications except *143.dleslie* on *Juropa* (and 15% dilation can also still be acceptable).

We show libunwind measurement overheads with and without our thunk optimization in Figure 8(a). *142.dmilc* shows improvement, however, only when we include unwinding from unimportant MPI routines. Although the thunk does not provide a major reduction in overhead, at least on *Juropa*, it still helps to provide more accurate call path visit counts.

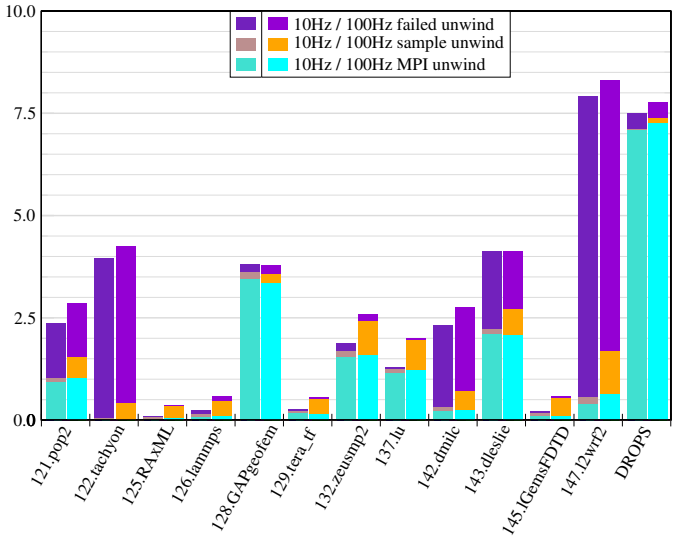
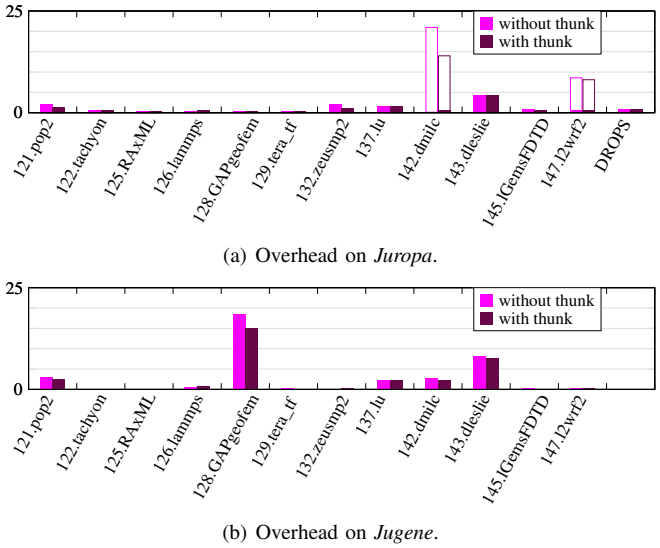


Fig. 7. Measurement distortion due to call-stack unwinding of MPI and non-MPI events including impact of unsuccessful unwinding.



(a) Overhead on *Juropa*.

(b) Overhead on *Jugene*.

Fig. 8. Measurement overhead percentages with and without thunks with StackwalkerAPI stack unwinding from PMPI wrappers.

C. StackwalkerAPI

We implement the thunk optimization for StackwalkerAPI on *Jugene*, a BlueGene/P system [16], which shows that it is portable across platforms and stack-unwinding APIs. Figure 8(b) shows that overheads on *Jugene* are similar to those on *Juropa* with most SPEC MPI applications. Overhead dilation is low with and without the optimization for all applications except *143.dleslie* and *128.GAPgeofem*. The thunk optimization provides observable benefit in the latter case.

IV. C++ Application Example

As already seen in Figure 4, the *DROPS* application execution fails to scale on *Juropa*, and therefore provides a suitable test case for applying the new measurement and analysis capability incorporated in the Scalasca prototype. First we examine the primary performance bottleneck relating to writing an output file. After applying a straightforward remedy for this problem, we then identify a second critical issue in the solver when calculating new parameter distributions.

A. Excessive Flushes Writing Output File

Figure 9 shows views of the Scalasca summary analysis reports of *DROPS* executions on *Juropa* with 256 processes, combining PMPI measurements with 100Hz call-path sampling. Metrics listed in the leftmost panel apply to the entire measurement (e.g., time aggregated for all processes), which can be refined to selected call paths shown as a tree in the middle panel. 66% (1.32 million seconds) of the total time occurs in the `MPI_Allreduce` call on the call path:

```
main
 + DROPS::Strategy<DROPS::ZeroFlowCL>
    + DROPS::SolveCoupledNS<...>
       + DROPS::LevelsetP2CL::GetInfo<...>
          + MPI_Allreduce
```

The over 6000 `MPI_Allreduce` calls encountered during execution occur on 56 distinct call paths. However, the critical call path is readily revealed by following the color-coded markers to the call paths with the largest values and selectively expanding them. We show the time breakdown for each of the 256 processes in the rightmost panel with that call path selected in the analysis report explorer. This time varies considerably (between 5000 and 7000 seconds) across MPI processes with process rank 0 a notable exception with only 0.08 seconds. Clearly the other processes must wait for rank 0 in this synchronizing collective communication. We next identify why rank 0 is delayed.

A computational imbalance heuristic that identifies serial execution isolates most of the serialized execution on rank 0 to a call path in a routine that writes output to file after each solver iteration. Most of the time occurs in the C++ standard IOstream library `std::endl` routine, which inserts a newline character and flushes the buffer for the output stream:

```
main
 + DROPS::Strategy<DROPS::ZeroFlowCL>
    + DROPS::SolveCoupledNS<...>
       + DROPS::Ensight6OutCL::Write
          + DROPS::Ensight6OutCL::putGeom
             + std::endl
```

The 6278 seconds associated with this `std::endl` call path are excessive. The associated source code calls it after writing each line of values, which results in a huge number of calls and corresponding buffer flushes to disk. This performance killer is widely documented, but unfortunately common in C++ codes. We can easily remedy it by substituting explicit newline insertions and only using `std::endl` to flush the buffer at the end of a block of writes. Replacing two instances of `std::endl` reduces the total execution time for *DROPS* more than 11-fold to 174,000 seconds (visible in the background display in Figure 9) and reduces the time in the previous `MPI_Allreduce` bottleneck to 20 thousand seconds (11.7%). While serial file writing remains a significant bottleneck, which varies from run to run, reduced to under 90 seconds its performance impact is no longer quite so crippling.

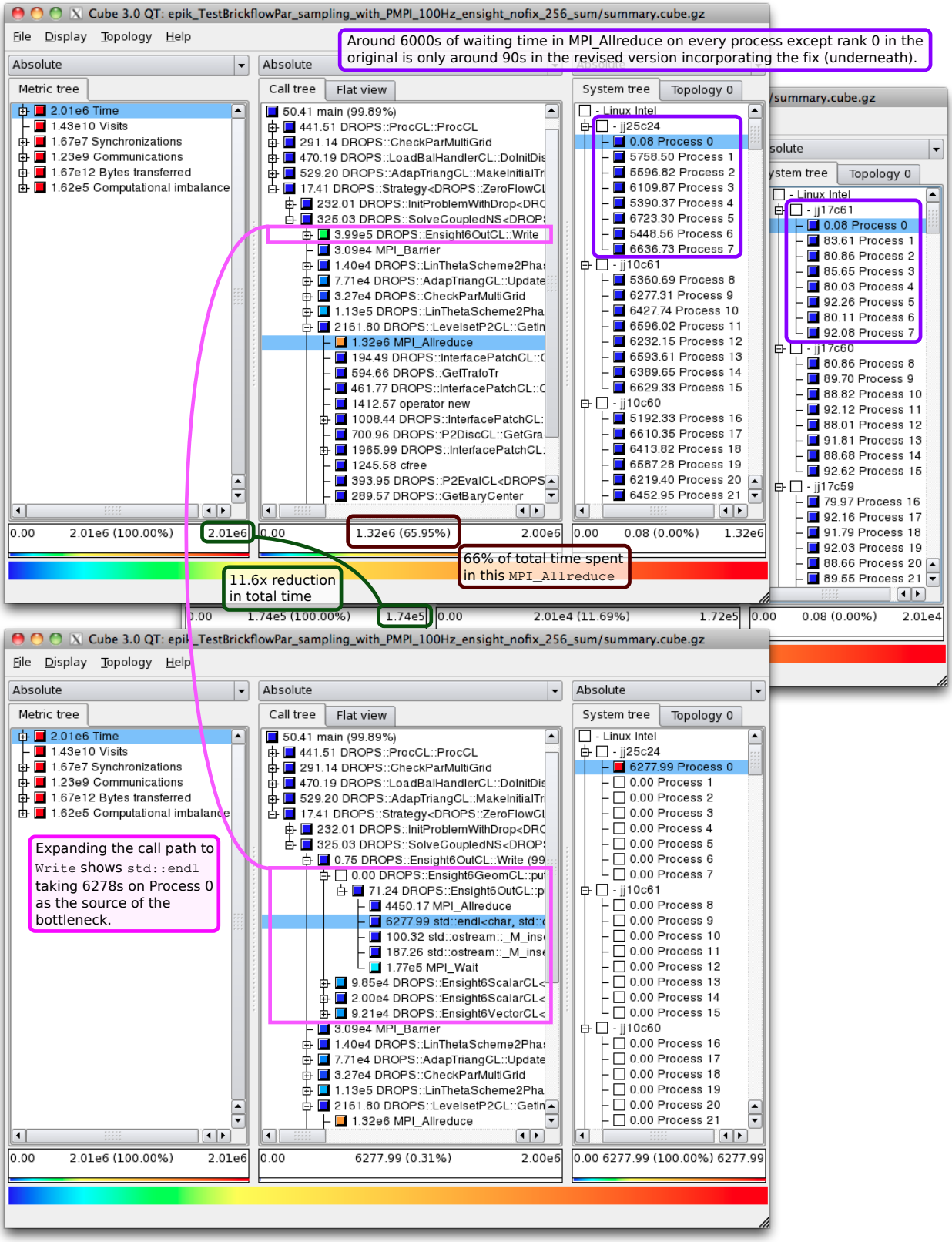


Fig. 9. Scalasca analysis report explorer that shows *DROPS* performance with 256 MPI processes before and after remedying the most serious bottleneck. The upper display shows that the other MPI processes wait in `MPI_Allreduce` 66% of the original total time for rank 0, which is delayed in `DROPS::Ensight6OutCL::Write`, including 6278 seconds in calls to `std::endl` (as seen in the lower display). Eliminating excessive `std::endl` calls (and associated implicit flushes to disk) reduced the serial writing time 75-fold and the other processes wait much less in `MPI_Allreduce` (partially visible in the background), resulting in 11.6 times faster overall execution.

The compiler-based instrumentation approach would not identify this I/O problem as easily. Although we could still find the `MPI_Allreduce` bottleneck, despite the significant measurement dilation, we could only localize the origin on rank 0 to `DROPS::Ensight6OutCL:putGeom` since routines in system libraries (such as `std::endl`) are not instrumented by the compiler. A more detailed localization would require manual instrumentation, which would further exacerbate the measurement overhead.

B. Parameter Redistribution Bottleneck

To investigate the performance of the solver, with less run-to-run variation due to the shared file system, we next performed measurements with writing of the output file disabled, again with 256 processes on *Juropa*. We show analysis reports from a measurement done with full compiler instrumentation (and runtime filtering of routines on non-MPI call paths) and one with direct measurement of MPI calls and 100Hz sampling in Figure 10. We summarize the key contents in Table I.

Although the compiler-instrumented measurement was dilated more than 250%, and overhead for unwinding call stacks in the sampling measurement was almost 8%, both measurements show remarkable agreement in the critical time metrics, such as the 85% of total time that is in MPI. Of course, MPI statistics for the number of synchronizations, communications and bytes transferred match exactly. Compiler instrumentation inhibited inlining of various routines (such as `DROPS::ProcCL::Probe`), and other cases indicate that inlining optimizations were performed differently, complicating comparisons of call paths and call trees.

Over half of the total time is concentrated in `MPI_Probe` calls that block while waiting for a matching message to arrive within `DROPS::LevelsetModifyCL::maybeModify`, which redistributes parameters to improve load balance. Rank 0 is again distinguished from the other ranks. We find a local calculation in `DROPS::FastMarchCL::Reparam` that only rank 0 performs. This calculation accounts for the bulk of the time, creating a serial execution bottleneck that impairs performance. The sampling-based measurement resolves the execution time further than compiler instrumentation can, to a variety of `std::_Rb_tree` operations related to the use of STL sets or maps.

Comparison with the 64-way measurement using the same input confirmed that the time for this serial section is independent of the number of processes. Since it requires more than half of the execution time with 256 processes, it explains the poor scaling for this test case. Better scaling will require efficient parallelization of this part of the application.

V. Related Work

Individually, both statistical sampling and direct instrumentation are used in a wide array of tools. HPCToolkit [1] is one example of a tool that exclusively relies on sampling and uses this data to deliver a comprehensive performance profile to the end user. It implements call-path profiling by gathering stack traces at each sample. It then uses this information to map the profile data back to the user's source code and its dynamic execution path. In order to keep overhead low, HPCToolkit applies a trampoline-based prefix optimization [9], [11] similar to the thunk optimization presented in this paper. We discussed the advantages of our approach in Section II-A4. Arnold and Sweeney present an earlier sampled call-path profiler for Java Virtual Machines [17], but this approach requires VM support for marking return addresses with special bits.

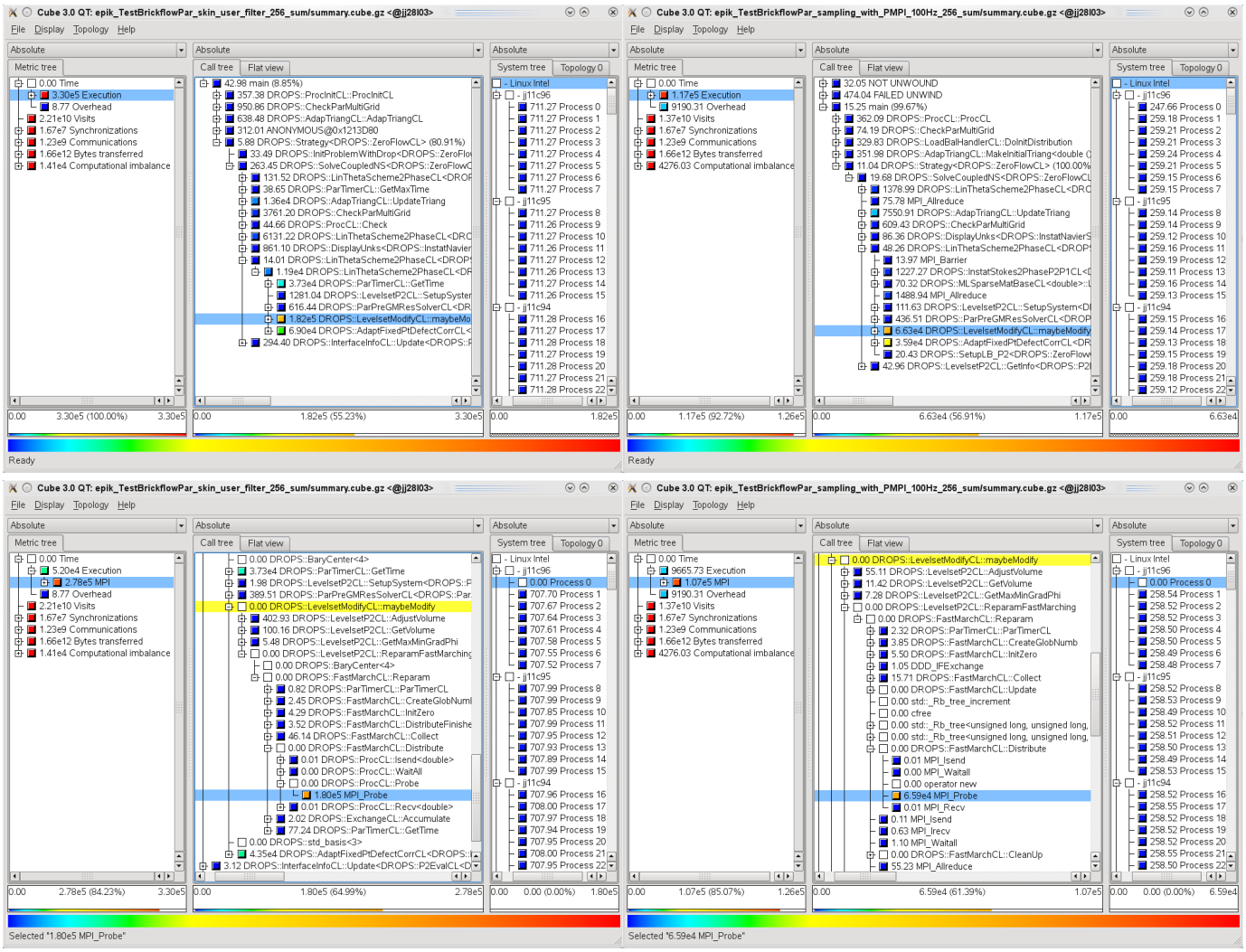
While sampling-based systems can only provide profiles, direct instrumentation can be used for both profiling (i.e., providing aggregate information over the runtime of the process) and tracing (i.e., delivering all events as they occurred during the execution of the program). Closely related to our work is *mpiP* [18], a profiler for MPI operations. Like our system, it uses the PMPI profiling layer to implement its functionality and to gather basic statistics about the usage pattern of MPI, including call-path context.

On the tracing side, tools like *strace* provide per-process trace information. For parallel applications, tools such as Vampir [19], Jumpshot [20] or the Intel Trace Analyzer and Collector [21] record all message transfers between MPI processes and store them to disk with complete timestamps. They can then display the collected trace on a timeline. This approach allows the user to carefully examine each individual message event, but it comes at the price of large storage requirements for the detailed trace files.

So far, little work has combined these two paradigms. However, several tools exist that offer both sampling and direct instrumentation side by side. The standard UNIX tool *gprof* [22] uses direct instrumentation to generate call-path displays for sequential programs in addition to sampling for a separate flat profile. In the area of parallel performance analysis, multiple tool sets offer both approaches in a single environment. CrayPAT [23] has mutually exclusive sampling and direct instrumentation modes. Open|SpeedShop [24] and the Oracle (formerly Sun) Studio Performance Tools [25] simultaneously record sampled and instrumented events, but ultimately present separate analyses. Only recently and concurrently to our work, TAU/ParaProf [26] started to prototype integrating sampling and instrumentation annotation events in a single measurement and to visualize them together [27]. However, the TAU approach records profiles and traces separately for the two sources of information, and is only able to merge them after measurement. By comparison, our work avoids storage and processing overheads of large traces by integrating the two sets of events into call-path profiles as they occur and carefully accounts for their interaction.

VI. Conclusion

We have presented a novel hybrid approach for call path profiling. This approach combines the low overhead of sampling with the detailed measurement of MPI routines possible with direct instrumentation. While the concept is straightforward, we found that several challenges arise in practice. First, we must use stack unwinding even for the direct measurements since we can no longer observe (nearly) all calls to determine



Compiler instrumentation Unwound 100Hz sampling

Fig. 10. Scalasca analysis report explorer summary reports from compiler instrumentation and sampling-based measurements (both with PMPI library interposition) of *DROPS* with 256 processes (where file output was disabled). The upper displays show more than half of the execution time on all processes is in `DROPS::LevelsetModifyCL::maybeModify`, and the lower displays focus on the `MPI_Probe` call path where all but rank 0 must wait.

TABLE I
SELECTED METRICS, CRITICAL CALL-PATH AND PROCESS VALUES FOR COMPILER-INSTRUMENTED AND SAMPLING-BASED MEASUREMENTS OF *DROPS*.

Compiler-instrumented (s)	(%)	Unwound sampling (s)	(%)	Metric / `Call path` / *Processes*
330,000	100.0	126,000	100.0	Time
—		474	0.4	– Failed unwind
—		32	0.0	– Not unwound
330,000	100.0	117,000	92.8	– Execution time
278,000	84.2	107,000	84.9	– MPI
43	0.0	15	0.0	`main`
6	0.0	11	0.0	`+ DROPS::Strategy<DROPS::ZeroFlowCL>`
263	0.1	20	0.0	`+ DROPS::SolveCoupledNS<DROPS::ZeroFlowCL>`
14	0.0	—		`+ DROPS::LinThetaScheme2PhaseCL<...>::DoStep`
11,900	3.6	48	0.0	`+ DROPS::LinThetaScheme2PhaseCL<...>::SolveLsNs`
0	0.0	0	0.0	`+ DROPS::LevelsetModifyCL::maybeModify`
1	0.0	0	0.0	`+ DROPS::LevelsetP2CL::ReparamFastMarching`
704	0.2	60	0.1	`+ DROPS::FastMarchCL::Reparam`
0	0.0	0	0.0	`+ DROPS::FastMarchCL::Distribute`
0	0.0	—		`+ DROPS::ProcCL::Probe`
180,000	54.5	65,900	56.3	`+ MPI_Probe`
0 / 708		0 / 259		*Rank 0 / Others*

call path information. However, the frequency of MPI calls requires several stack unwinding optimizations to keep measurement overhead sufficiently low. Second, we must ensure that the two instrumentation techniques do not interfere with each other. This challenge led us to develop a novel sampling methodology that accounts for the omission of samples that occur during MPI routines.

We implemented our hybrid profiling approach within Scalasca and presented a detailed evaluation of the costs to gather call path profiles with direct instrumentation, sampling and our hybrid approach. We found that all approaches work well for many applications in the SPEC MPI2007 suite. However, direct instrumentation suffers significant overhead for some applications, whereas sampling alone makes gathering some critical information difficult. Our case study of the *DROPS* CFD application demonstrates that the detailed information and reduced overhead of our hybrid approach facilitates analysis of bottlenecks in real applications. Overall, our approach significantly improves on existing techniques and we are working to make it available in a forthcoming release of the open-source Scalasca toolset.

ACKNOWLEDGMENTS

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-468224). Also, funding from the Deutsche Forschungsgemeinschaft (German Research Association) through Grant GSC 111 and the Helmholtz Association of German Research Centers through Grant VH-NG-118 is gratefully acknowledged. Finally, we would like to thank the developers of the StackwalkerAPI for their helpful support.

REFERENCES

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

[2] S. S. Shende and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, 2006.

[3] S. S. Shende, "The Role of Instrumentation and Mapping in Performance Measurement," Ph.D. dissertation, University of Oregon, August 2001.

[4] A. R. Bernat and B. P. Miller, "Incremental Call-Path Profiling," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 11, pp. 1533–1547, August 2007.

[5] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 2.2: Profiling Interface," ch. 14, September 2009, http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[6] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, Apr. 2010.

[7] D. Mosberger. (2010) libunwind. [Online]. Available: http://www.nongnu.org/libunwind/

[8] Dyninst Project. (2010) StackwalkerAPI. [Online]. Available: http://www.paradyn.org/html/stackwalker1.1-features.html

[9] N. Froyd, J. Mellor-Crummey, and R. Fowler, "Low-Overhead Call Path Profiling of Unmodified, Optimized Code," in *Proc. 19th Int'l Conf. on Supercomputing (ICS'05, Cambridge, MA, USA)*. ACM, 2005.

[10] N. Froyd, N. Tallent, J. Mellor-Crummey, and R. Fowler, "Call path profiling for unmodified, optimized binaries," in *GCC Summit '06: Proc. of the GCC Developers' Summit 2006*, 2006, pp. 21–36.

[11] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan, "Binary Analysis for Measurement and Attribution of Program Performance," in *Proc. 2009 SIGPLAN Conf. on Programming Language Design and Implementation (PLDI, Dublin, Ireland)*. ACM, 2009, pp. 441–452.

[12] Standard Performance Evaluation Corporation. (2007) SPEC MPI2007 Benchmark Suite. [Online]. Available: http://www.spec.org/mpi2007/

[13] S. Groß, J. Peters, V. Reichelt, and A. Reusken, "The DROPS Package for Numerical Simulations of Incompressible Flows Using Parallel Adaptive Multigrid Techniques," RWTH Aachen University, IGPM-Report 211, 2002, http://wissrech.ins.uni-bonn.de/research/pub/gross/IGPM211_N.pdf.

[14] C. Iwainsky and D. an Mey, "Comparing the Usability of Performance Analysis Tools," in *Proc. Euro-Par 2008 Workshops on Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 5415. Springer, April 2009, pp. 315–325.

[15] Jülich Supercomputing Centre. (2010) JuRoPA – Jülich Research on Petaflop Architectures. [Online]. Available: http://www.fz-juelich.de/jsc/juropa/

[16] ——. (2010) JUGENE – Juelicher BlueGene/P. [Online]. Available: http://www.fz-juelich.de/jsc/jugene/

[17] M. Arnold and P. F. Sweeney, "Approximating the Calling Context Tree via Sampling," IBM Research Division, Almaden, CA, IBM Research Report 21789(98099), July 7 2000.

[18] J. Vetter and C. Chambreau. (2010) mpiP: Lightweight, Scalable MPI Profiling. [Online]. Available: http://mpip.sourceforge.net/

[19] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources," *Supercomputer*, vol. 12, pp. 69–80, 1996.

[20] A. Chan, W. Gropp, and E. Lusk, "An Efficient Format for Nearly Constant-Time Access to Arbitrary Time Intervals in Large Trace Files," *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, 2008.

[21] Intel. (2010) Trace Analyzer and Collector. [Online]. Available: http://software.intel.com/en-us/intel-trace-analyzer/

[22] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[23] Cray Inc. (2010) Using Cray Performance Analysis Tools. [Online]. Available: http://docs.cray.com/books/S-2376-51/

[24] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008.

[25] M. Itzkowitz and Y. Maruyama, "HPC Profiling with the Sun Studio Performance Tools," in *Proc. 3rd Int'l Workshop on Parallel Tools for High Performance Computing (Dresden, Germany)*. Springer, September 2009, pp. 67–93.

[26] R. Bell, A. Malony, and S. Shende, "ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis," in *Proc. 9th Euro-Par Conf. (Klagenfurt, Austria)*, ser. Lecture Notes in Computer Science, vol. 2790. Springer, Aug. 2003, pp. 17–26.

[27] A. Morris, A. D. Malony, S. Shende, and K. Huck, "Design and Implementation of a Hybrid Parallel Performance Measurement System," in *Proc. 39th Int'l Conf. on Parallel Processing (ICPP, San Diego, USA)*. IEEE Computer Society, September 2010, pp. 492–501.