

Effectively Presenting Call Path Profiles of Application Performance

Laksono Adhianto, John Mellor-Crummey and Nathan R. Tallent

Department of Computer Science

Rice University

Houston, TX

Email: {laksono,johnmc,tallent}@rice.edu

Abstract—Call path profiling is a scalable measurement technique that has been shown to provide insight into the performance characteristics of complex modular programs. However, poor presentation of accurate and precise call path profiles obscures insight.

To enable rapid analysis of an execution’s performance bottlenecks, we make the following contributions for effectively presenting call path profiles. First, we combine a relatively small set of complementary presentation techniques to form a coherent synthesis that is greater than the constituent parts. Second, we extend existing presentation techniques to rapidly focus an analyst’s attention on performance bottlenecks. In particular, we (1) show how to scalably present three complementary views of calling-context-sensitive metrics; (2) treat a procedure’s static structure as first-class information with respect to both performance metrics and constructing views; (3) enable construction of a large variety of user-defined metrics to assess performance inefficiency; and (4) automatically expand hot paths based on arbitrary performance metrics — through calling contexts and static structure — to rapidly highlight important program contexts. Our work is implemented within HPCTOOLKIT, which collects call path profiles using low-overhead asynchronous sampling.

I. INTRODUCTION

Analyzing program performance to find performance bottlenecks is still challenging work, especially for parallel applications. Although performance analysis tools have been around for decades, the complexity of microprocessors continues to increase. Some new challenges include the impact of shared-memory hierarchy such as false sharing, the growing importance of thread-level parallelism, and the widespread use of multi-level parallelism (short vectors, multiple functional units, pipelining, multicore, hardware accelerators, multi-socket and multi-node).

To identify performance bottlenecks effectively, performance tools must gather both accurate and precise measurements and attribute those measurements to source code. Tools may also perform appropriate on-line and off-line analyses to highlight areas of computational inefficiency at the source code level. Finally, tools must present the results in a way that engages the analyst, focuses attention on what is important, and automates common analysis subtasks to reduce the mental effort and frustration of sifting through a sea of measurement details.

In this paper, we focus on techniques for effective presentation of call path profiles. While detailed traces are very useful

for identifying some kinds of bottlenecks, they are difficult to collect accurately for long-running or large-scale executions. As a consequence, we focus on call path profiles for three reasons. First, for modern modular software, calling context is essential for understanding many performance problems. Second, using asynchronous statistical sampling, it is possible to collect accurate and precise call path profiles for only a few percent overhead [1]. Third, call path profiling can be effectively applied to long-running and large-scale applications [15].

To enable rapid analysis of an execution’s performance bottlenecks, we make the following contributions. *First*, we combine a relatively small set of complementary presentation techniques to form a coherent synthesis that is greater than the constituent parts. When united within a single presentation tool, these techniques uniquely and rapidly focus an analyst’s attention on performance bottlenecks rather than on unimportant information. *Second*, we extend existing presentation techniques to facilitate the goal of rapidly focusing an analyst’s attention on performance bottlenecks. In particular, we (1) synthesize and present three complementary views of calling-context-sensitive metrics; (2) treat a procedure’s static structure as first-class information with respect to both performance metrics and constructing views; (3) enable a large variety of user-defined metrics to describe performance inefficiency; and (4) automatically expand hot paths based on arbitrary performance metrics — through calling contexts and static structure — to rapidly highlight important performance data.

Our work is part of HPCTOOLKIT, an integrated suite of open-source tools for measurement and analysis of program performance on computers ranging from multicore desktop systems to large supercomputers [1], [15]. HPCTOOLKIT consists of four primary tools: `hpcrun` for collecting low-overhead high-accuracy profiles using asynchronous statistical sampling, `hpcstruct` for recovering static source code structure, `hpcprof` for correlating dynamic profiles with static source code structure, and `hpcviewer` for interactively presenting the resulting experiment databases. This paper discusses the principles that undergird `hpcviewer`. To demonstrate `hpcviewer`’s capability, we analyze profile data from a mesh generation benchmark and from real-world applications for turbulent combustion and reactive flow.

II. PRINCIPLES OF EFFECTIVE PRESENTATION

Our presentation techniques are all derived from the basic principle that the job of a presentation tool is to *enable an analyst to rapidly pinpoint and quantify performance bottlenecks*. Based on this, we make the following observations.

a) *Presentations should be flexible, showing data from different perspectives*: Calling-context sensitive measurements can be viewed in different ways. For example, the user can emphasize either the caller-callee (top-down) or the callee-caller (bottom-up) relationship. Depending on the nature of the performance problem, one view may be more informative than another. A presentation tool that only supports one view will not be as effective as one that supports multiple views. In Section III, we introduce three different views supported by `hpcviewer` that complement each other.

b) *Presentations should avoid visual clutter that distracts analysts from focusing on real problems*: A set of performance data often includes measurements for procedures that consume very few resources and are therefore unimportant from the perspective of diagnosing performance bottlenecks. A presentation tool should deemphasize this data unless it somehow becomes relevant. As described in Section V, `hpcviewer` forces the user to approach performance data in a top-down fashion. It is designed to keep attention focused on program *scopes* where performance is of interest, where a program *scope* is a program component like a procedure, loop and call site.

c) *Presentations should emphasize potential performance bottlenecks*: Tools should present performance data so as to emphasize potential bottlenecks. For instance, an effective presentation should guide users to rapidly drill down into a context that represents a potential bottleneck. As shown in Section VI, a handful of techniques can effectively locate potential bottleneck.

d) *Presentations should be scalable*: An effective presentation should be able to handle any size of data, from small to very large data, with modest memory requirements and acceptable speed and responsiveness. To handle large amounts of data, we need to process and store some data only when they are needed. The discussion of scalability features implemented in `hpcviewer` is presented in Section VII.

III. THREE COMPLEMENTARY CONTEXT-SENSITIVE VIEWS

This section introduces three perspectives (known as *views*) implemented in `hpcviewer`: Calling Context View, Callers View and Flat View, as well as the integration of static and dynamic program scopes in the three views. `hpcviewer`'s input data consists of static program structure (such as files and loops), dynamic calling context represented by a sequence of <call site, callee> pairs (also called calling context tree or CCT) and "raw" metrics which are sample metrics generated by the profiler. `hpcviewer` needs then to generate callers tree and flat tree, followed by metrics attribution (Section IV). Due to space constraints, the construction of callers tree and

```
1 // recursive function
2 g() {
3   if (...) g();
4   if (...) h();
5 }
6
7 // main routine
8 m() {
9   f();
10  g();
11 }
12 }
```

(a) file1.c

```
1 // recursive function
2 g() {
3   if (...) g();
4   if (...) h();
5 }
6
7 h() {
8   for (...) // 11
9   for (...) // 12
10 }
```

(b) file2.c

Fig. 1: Example of a program that consists of two files: `file1.c` and `file2.c`.

flat tree is not described in this paper, but interested reader can find the details elsewhere [13].

A. Calling Context View

The *Calling Context View* is a top-down view that represents an execution's dynamic calling contexts or call paths. Using this view, we can explore performance measurements of an application in a top-down fashion to understand the costs incurred by calls to a procedure in a particular calling context.

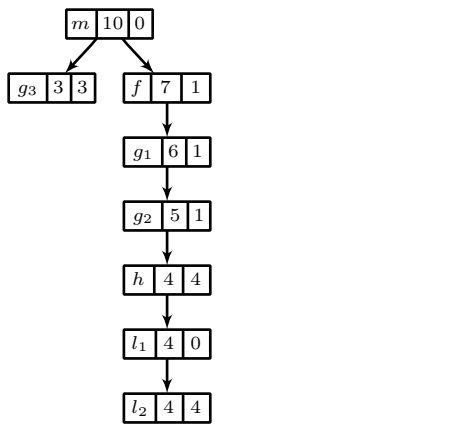
Figure 1 shows an example program that contains some procedures including a recursive procedure. From the execution of this program, we can construct a calling context tree as shown in Figure 2a. In this example, procedure `g` is called from multiple routines, namely `f`, `g` and `m`, where each call represents a distinct calling context, namely g_1 , g_2 and g_3 respectively. Using this view, we can explore performance measurements of an application in a top-down fashion to understand the full contexts in which costs¹ were incurred both across and within procedures.

Our toolkit distinguishes calling context precisely by individual call sites. Figure 3 shows the `hpcviewer`'s Calling Context View. This top-down view attributes the costs incurred within a scope to the scope's full calling context. Scopes represented in the Calling Context View include not only procedures, but also loops and inlined code, as well as individual statements. Each time a scope is sampled, its costs are attributed to a calling context that represents a fusion of the dynamic calling context gathered by our toolkit's profiler, represented by a sequence of <call site, callee> pairs, with information about static program structure.

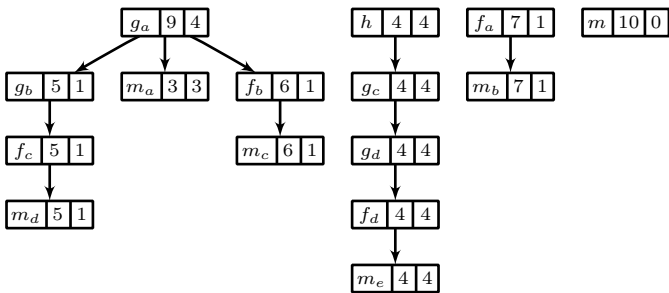
B. Callers View

The *Callers View* is a bottom-up view that enables the user to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are called in more than one context. For instance, a message-passing program may call `MPI_Wait` in many different calling contexts. The cost of any particular call

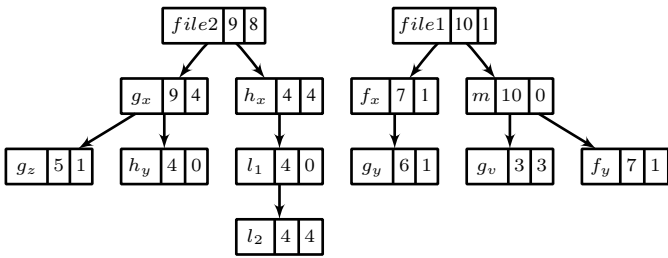
¹We use the term *cost* to include a multiplicity of metrics that are measures of work (e.g., instructions), resource consumption (e.g., bus transactions), or inefficiency (e.g., stall cycles).



(a) Calling context tree (top-down view)



(b) Callers tree (bottom-up view)



(c) Flat tree (static view)

Fig. 2: Three different views representing execution costs for the code in Figure 1. Each node in the tree represents a program scope. Each scope has three columns: the left column is the name of the scope, the middle column is the inclusive cost and the right column is the exclusive cost.

Note: alphanumerical subscripts are used to distinguish between different instances of the same procedure. We use different labels because there is no natural one-to-one correspondence between the instances in the different views.

will depend upon its context. Serialization or load imbalance may cause long waits in some calling contexts but not others.

Figure 2b shows the callers tree constructed from the calling context tree (CCT) in Figure 2a. Note that with the Callers View, procedure *g* in *file2* (Figure 1b), which is represented by node *g_a*, can be clearly shown to have procedure *g*, *m* and *f* as the callers (represented with nodes *g_b*, *m_a*, *f_b* respectively). By using the callers tree, it is easy to identify that the call of *g* from *f* contributes the highest cost in procedure *g*.

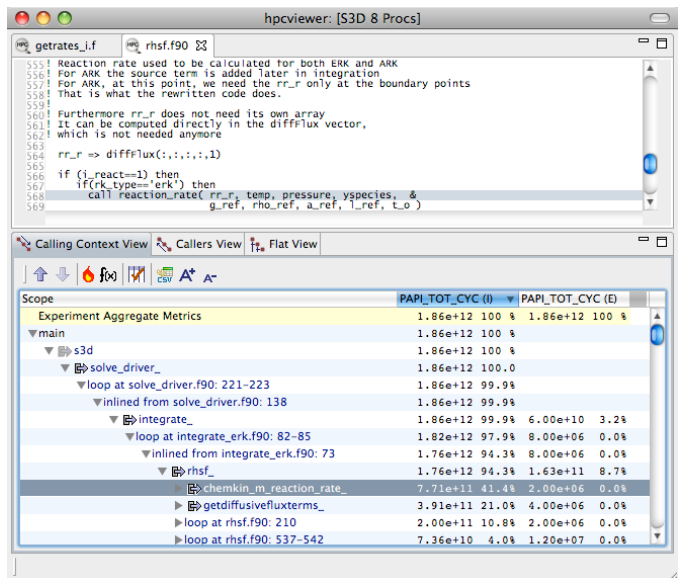


Fig. 3: A Calling Context View of performance metrics for a turbulent combustion application. The highlighted line shows a potential performance bottleneck indicated by “hot path” analysis.

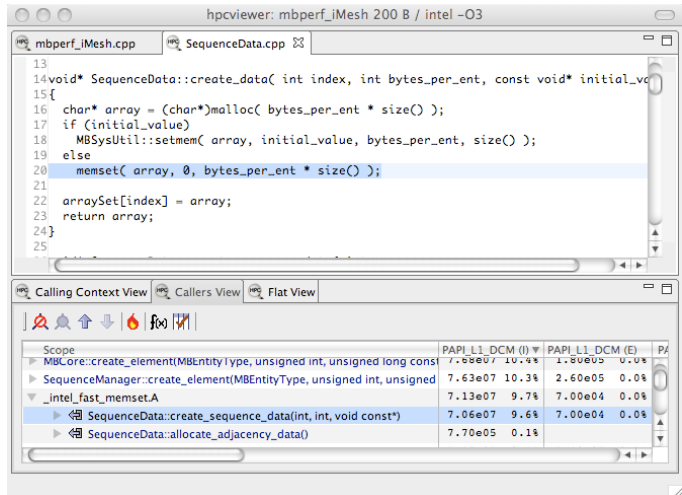


Fig. 4: A Callers View for a sequential mesh generation benchmark.

Figure 4 shows *hpcviewer* presenting a Callers View for the MOAB mesh package, a component for representing and evaluating structured and unstructured mesh data, developed at Argonne National Laboratory. This view shows that the Intel x86-64 compiler replaced calls to *memset* with its own optimized implementation. From the Callers View, we see that the routine *_intel_fast_memset.A* is called from two different callers. Overall, calls to *_intel_fast_memset.A* account for 9.7% of the total L1 data cache misses in the application. Of those, almost all (9.6%) come from the call to *memset* by *Sequence_data::create*.

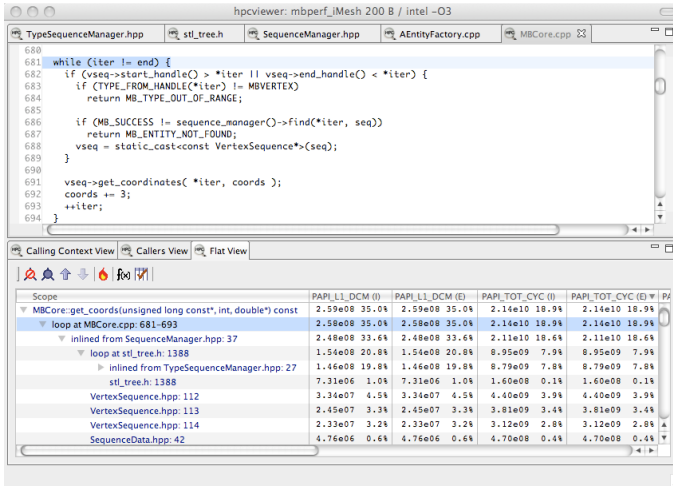


Fig. 5: A Flat View showing the attribution of cache misses and cycles through routines, loops, and inlined code.

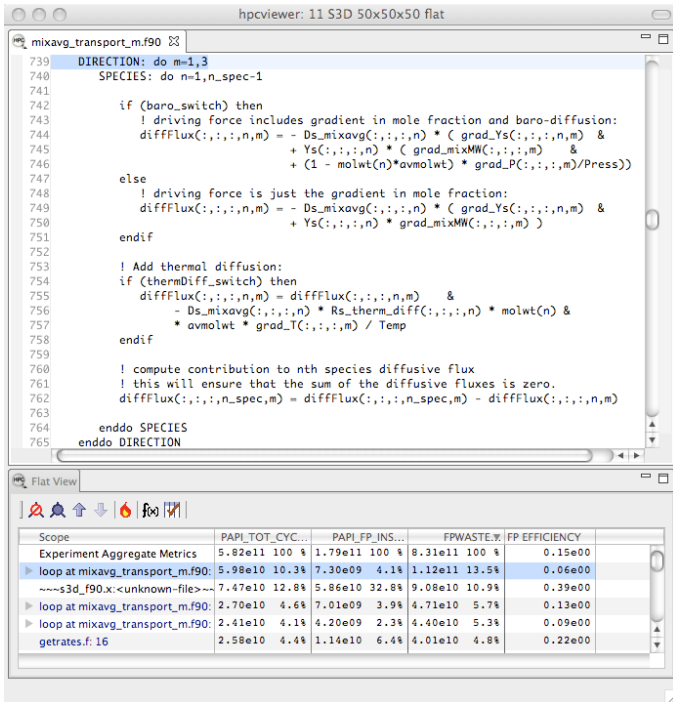


Fig. 6: Using a derived metric of floating-point waste to analyze the performance of loop nests of a turbulent combustion code.

C. Flat View

The Flat View correlates performance data to an application’s static structure such as load module, file, procedure, loop and statement. All costs incurred in any calling context by a procedure are aggregated together in the Flat View. This complements the Calling Context View, in which the costs incurred by a particular procedure are represented separately for each call to the procedure from a different calling context.

Figure 2c shows a flat view tree constructed from the calling context of Figure 2a. With this flat view, we can easily identify

that file2 has the highest exclusive cost over all files in the view; similarly, procedures g and h have the highest exclusive cost among all procedures in the view. Figure 5 shows a real example of using Flat View in a benchmark used to study the performance of the MOAB mesh package. This view shows how the costs associated with the procedure `MBCore::get_coords` are attributed to a loop containing inlined code.

`hpcviewer` also supports a feature that we call *flattening*. Flattening elides a scope and shows its children instead. However, applying flattening to a childless scope (a leaf) has no effect. Since scopes are displayed in a hierarchical fashion, flattening eliminates layers of hierarchical structure (e.g., files and procedures) that prevent making direct comparisons between loops in different routines. Figure 6 shows a Flat View where flattening is used to facilitate comparing costs for loops in different routines.

D. Presenting Calling and Static Contexts

1) *Fine-grain hierarchical attribution of costs to static program structure in the flat view:* Figure 5 uses `hpcviewer`’s Flat View to focus on the performance of a single routine `MBCore::get_coords` of the `mbperf_IMesh` mesh benchmark. The rightmost column of metrics shows the exclusive total cycles attributed to each scope. We can see that all of the cycles spent in the routine (18.9% of the total cycles in the execution) is spent in the highlighted loop nested within the routine. Within the loop, we can track attribution of this cost throughout a hierarchy of inlined code that represents the application of the `find` operation on the `sequence_manager` that is called on line 686. The fourth line of the navigation pane shows an inlined loop to search sequences implemented using the red-black tree implementation from the C++ standard template library. The next line represents calls to the `SequenceCompare` operator that are inlined into this loop. Looking at the second metric column, we can see that applying the comparison operator accounts for 19.8% of the L1 data cache misses in the execution. Selecting any of the other lines in the lower left navigation pane navigates the source pane to show the corresponding source code associated with the file and line numbers. While this example showcases the ability of our performance analysis toolkit to collect and attribute data with astonishing precision in the presence of inlining, that is not the purpose of including it here. For the purpose of this paper, the example highlights the viewer’s presentation capabilities in the Flat View to attribute performance metrics throughout a hierarchy of program structure within a routine that includes nested loops, multiple levels of inlining, as well as individual source lines.

2) *Integration of static and dynamic program structure in the Calling Context View:* Figure 3 shows a Calling Context View of the S3D turbulent combustion code developed at Sandia National Laboratory. The sequence of lines shown in the navigation pane represents a call path through the application. At the top of the call chain is `main` shown in plain black (as opposed to a blue hyperlink) because it corresponds

to routines that have no associated source code. Their implementations are provided in binary-only form in the language run-time library for the compiler. On subsequent lines, this view shows a long call chain from the main routine down into the `chemkin_m_reaction_rate_` routine, where 41.4% of the inclusive cycles is spent computing reaction rates for the chemical species involved in the simulation. It is notable that in this top-down Calling Context View of the metric data for the execution, we see that the call chain presented includes both dynamic context (procedure calls) and the loop nests surrounding these procedure calls. Our toolkit uses information gleaned from the line map of an executable to determine when a call site is nested within a loop. The viewer then presents an integrated view that includes both static and dynamic context.

IV. COMPUTING CONTEXT-SENSITIVE METRICS

Once the Callers View and Flat View are constructed based on the Calling Context View, the next step is to attribute metrics to each scope in the views. We use the term *metric* to represent measurements such as resource consumption (e.g., bus transactions) as inefficiency (e.g., stall cycles). This section begins with a simple method for computing metric values for each view and then introduces a technique to correctly handle recursive programs.

A. Metrics

The Calling Context View shown by `hpcviewer` represents a data structure that we call a canonical calling context tree (canonical CCT). This data structure is synthesized by `hpcprof` by integrating information about *static* program structure into *dynamic* call chains. Consequently, a scope in the CCT is classified as either a *dynamic* or *static* scope. A *dynamic* scope represents caller-callee relationship, while a *static* scope represents static program structure such as load module, file, procedure, loop, line statement or inlined procedure.

We define two types of calling context metrics: *inclusive* and *exclusive*. Inclusive metrics for a particular scope reflect costs for the entire subtree rooted at that scope while exclusive metrics, to some extent, do not. We have found it useful to distinguish between two types of exclusive metric values because although one often thinks of procedure frames in the context of call chains, it is natural to think of loops in the context of a procedure. Therefore, we adopt a hybrid definition of exclusive metrics based on whether metric values for a scope x should be computed with respect to dynamic call chains or a procedure’s static hierarchy:

- 1) Dynamic: sum every *descendant* statement of x that is not across a call site.
- 2) Static: sum every *child* statement of x .

The first rule indicates that the exclusive cost of a dynamic scope is the total of all its descendant statements within procedure frame, while the second rule specifies that the exclusive cost of a static scope is the sum of the exclusive cost of its children statements. For instance, the CCT in Figure 2a shows that procedure h has a direct child loop l_1 , which is a

parent of loop l_2 ; and the exclusive cost of l_1 does not include the cost of l_2 (rule 2) since l_2 is not a statement. Furthermore, the flat tree in Figure 2c shows that procedure h can be both a static procedure as represented by node h_x , and a dynamic call site as represented by node h_y . As a static procedure, h includes the cost of all statements in l_2 (rule 2), but as a dynamic call site scope, its exclusive cost only includes the cost of its invocation as shown in node h_y (rule 1).

The computation of metrics in the viewer consists of three different steps:

- 1) *initialization* : initialize the exclusive and inclusive costs.
- 2) *multiple view creation*: create Callers View and Flat View and compute the metrics attribution.
- 3) *finalization* : refine the metrics for all views. This step is needed to compute metrics of parallel programs [14].

In the *initialization* step, the viewer computes exclusive values and inclusive values. Here we describe how these are computed. Initially, a CCT contains metric values only at sample points, which are typically leaf scopes. We define these values to be exclusive metrics for scopes. For a scope x , the exclusive value $m_E(x)$ for metric m is defined to be the number of samples at x multiplied by the sample period. For any dynamic scope x , we initialize $m_E(x) = 0$. We then compute exclusive values for each node x using the formula:

$$m_E(x) = \begin{cases} \sum_{s \in \text{desc}(x)} m_E(x_s) & x: \text{procedure frame} \\ \sum_{s \in \text{child}(x)} m_E(x_s) & x: \text{other static} \\ m_E(x) & x: \text{dynamic} \end{cases} \quad (1)$$

When x is a dynamic scope, case three simply returns the cost’s initial value. When x is a static scope that is not a procedure frame, the second case uses the Static definition. Here, the function `child(x)` returns every scope that is a child of x . When x is a procedure frame, the first case applies the Dynamic definition. The function `desc(x)` returns every scope s that is a descendant of x and for which the path between x and s contains no call site.

Using exclusive metric values from Equation 1, we define inclusive values for metric m at node x as:

$$m_I(x) = \begin{cases} \sum_{c=1}^{n_C(x)} m_I(x_c) + m_E(x) & x: \text{interior} \\ m_E(x) & x: \text{leaf} \end{cases} \quad (2)$$

This simple inductive definition computes an interior scope’s inclusive metric value from its children’s inclusive values and its own exclusive value. The function $n_C(x)$ refers to the number of children for scope x .

Figure 3 clearly shows the importance of presenting inclusive and exclusive costs of a metric. In this figure, the leftmost column in the table of performance metrics displays the total number of cycles (PAPI_TOT_CYC) consumed by each program scope in the Calling Context View. Seven lines from the bottom of the figure, we see that the data for the

loop at line 82 of file `integrate_erk.f90`. We see that this scope accounts for a significant inclusive number of cycles (97.9%), but the exclusive cost of the loop is negligible, only 0.0%. Most of its work is in `rhsf_` procedure it calls (8.7%) and its descendants, instead of the loop itself.

Call path profile measurements collected by HPCTOOLKIT correspond directly to the Calling Context View. From exclusive metric costs in the Calling Context View, `hpcviewer` derives other views in a step we refer to as *multiple view creation*. For the Callers View, we collect the cost of all samples in each function and attribute that to a top-level entry in the Callers View. Under each top-level function, we can look up the call chain at all of the contexts in which the function is called. For each function, we apportion its costs among each of the calling contexts in which they were incurred. We compute the Flat View by traversing the calling context tree and attributing all costs for a scope to the scope within its static source code structure. The Flat View presents a hierarchy of nested scopes from the load module, file, routine, loops, inlined code and statements.

The *finalization* step is used to scalably compute metrics for large-scale parallel executions; details are described in [14]. In large parallel executions, it is not scalable to store all information for all processes/threads in memory. Instead, HPCTOOLKIT summarizes the profile data using statistical metrics such as arithmetic mean, min, max and standard deviation. The finalization step in `hpcviewer` then assembles intermediate summary metric values into final values.

B. Computing Metrics for Recursive Programs

A subtle issue arises when computing inclusive metrics for the Callers View and the Flat View for recursive programs. If one naïvely aggregates the inclusive costs of a recursive procedure when building either the Callers View or the Flat View, then one will count the inclusive time spent along a chain of recursive calls multiple times.

To avoid this problem, it is necessary to conditionally attribute the inclusive cost of a scope in the Callers View. The following is a sketch our solution; for the full solution, we refer the reader elsewhere [13]. Assume the CCT contains multiple instances of scope x . We define an instance of scope x be *exposed* if it contains no ancestor instance of x . To form the inclusive cost for x within the Callers View, we sum all inclusive costs of x 's *exposed* instances within the Calling Context View.

Figure 1 shows an example of a recursive program separated into two files: `file1.c` (Figure 1a) and `file2.c` (Figure 1b). Routine g (Figure 1b) can behave as a recursive function depending on the value of the condition branch (Line 3-4). Figure 2a shows an example of the call chain execution of the program annotated with both inclusive and exclusive costs. Computation of inclusive costs from exclusive costs in the Calling Context View involves simply summing up all of the costs in the subtree below. In this figure, we can see that on the right path of the routine m , routine g (instantiated in the diagram as g_1) performed a recursive call (g_2) before calling

routine h . Here we can see that g_1 and g_3 are exposed, while g_2 is not since it has g_1 as its ancestor. Although g_1 , g_2 and g_3 are all instances from the same routine (i.e., g), we attribute a different cost for each instance. This separation of cost can be critical to identify which instance has a performance problem.

Figure 2b shows the corresponding scope structure for the Callers View and the costs we compute for this recursive program. The procedure g noted as g_a (which is a root node in the diagram), attributes a different cost to g than to other instances as noted as g_b , g_c and g_d . For instance, on the first tree of this figure, the inclusive cost of g_a is 9, which is the sum of exposed g in CCT (Figure 2a): the inclusive cost of g_3 (which is 3) and g_1 (which is 6). We do not attribute the cost of g_2 here since it is not exposed (in other term, the cost of g_2 is already included in g_1).

Inclusive costs need to be computed similarly in the Flat View, where the inclusive cost is the sum of exposed scopes in Calling Context View. For instance, in Figure 2c, The inclusive cost of g_x , defined as the total cost of all instances of g , is 9 and this is consistently the same as the cost in Callers View. The advantage of attributing different cost to each calling context for g is that it enables user to identify which instance is the responsible for performance losses.

V. FOCUSING AN ANALYST'S ATTENTION

It is not sufficient for a tool to indiscriminately present all performance data. Instead, a tool should guide the user to focus on what is important. In this section, we describe how `hpcviewer` facilitates rapid analysis by (1) encouraging top-down analysis, (2) streamlining repetitive tasks, (3) automatically expanding hot paths, and (4) supporting a rich set of derived metrics.

A. Top-Down Analysis

`hpcviewer` was carefully designed to focus user attention on program scopes that are costly according to a metric of interest. The following design aspects help preserve that focus.

- `hpcviewer` forces the user to approach performance data in a top-down fashion. All access to the program source code is through the navigation pane. For each of the three views of performance data (calling context, callers, and flat), the navigation pane presents a tree or a forest of trees. Scopes at each level of the nesting in the navigation pane are sorted according to the selected metric column.²
- There is no direct access to metric data from the source pane. An earlier design enabled the user to select a scope (e.g., loop nests) in the source pane to call up its associated metrics. We found this to be distracting: it encouraged users to inspect performance data that was often of little or no importance (i.e., its metric values were small). If a scope is important, its metric data will

²While the user typically sorts scopes in the navigation pane by a selected metric column, the user can sort according to the source scopes in the navigation pane itself, but this capability arose from design orthogonality rather than a particular need.


cause it to rank highly in the navigation pane and invite inspection. Our new design forces users to analyze in a top-down fashion.³

- Providing inclusive metrics in the calling context tree view enables casual users of a library to avoid looking deeper than a library’s application programming interface (API) since all of the cost of using API functions is accumulated at the interface functions.
- Performance data is sparse; there is no representation for a scope in `hpcviewer` unless there is a non-zero performance metric or it is a parent of another scope that meets this criteria. This keeps attention focused on scopes where performance is of interest.
- Any metric table cell where data is zero is left blank. Blank cells can be understood at a glance; explicitly representing zeros invites the user to gaze upon cells only to find that they contain no useful information.
- To help the user observe metrics, instead of displaying naively long and painful numbers, `hpcviewer` only displays the metrics with scientific notation with simple and intuitively readable format.

B. Fine Tuning: Removing Distractions / Human Factors

To improve the usability of `hpcviewer`, we fine-tuned its design in two ways to reduce repetitive operations for users that arose when expanding chains of static or dynamic context in the navigation pane.

An earlier design of `hpcviewer` displayed information about a call site on a separate line from that about its callee. While being able to navigate to the source code for the call site in addition to the source code for the callee in `hpcviewer` is valuable, representing callee and call site information on separate lines doubled the length of call chains. In practice, we found opening these long call chains to be tedious. Our current design for `hpcviewer` presents both call site and callee information on a single line in the navigation pane, which shortens the length of the call chains in `hpcviewer` by half and halves the effort to open them a link at a time (Figure 3 for Calling Context View and Figure 4 for Callers View).

The icon of a box with a right-facing arrow () represents a call site; the box represents the call site, and the arrow points at the name of the routine called. Clicking on the call site icon navigates the source pane to the call site. Clicking on the routine name navigates the source pane to the callee. The inclusive cost of a metric for a line representing a call site/callee pair represents the inclusive cost attributed to the callee in that context, namely the cost of the callee and any routine it calls; the exclusive cost on that line represents just the cost attributed to the callee alone in that context (without the routines it calls) plus any cost associated with the call site line itself.

³A side benefit of this omission is that we avoid needing to address the many-to-one mapping problem that arises when the user selects a routine that is called in many places.

C. Hot path analysis.

In programs with layered software abstractions, it can be tedious to individually open each link along a deep chain of calling contexts. Hot path analysis enables the user to instantaneously drill down into a nested context to pinpoint where costs were incurred. To apply hot path analysis, the user selects a column in the metric pane, a program scope in the navigation pane, and then presses the hot path button denoted by the flame. `hpcviewer` will automatically expand scopes along the hot path for the selected metric in the subtree rooted at the selected scope according to the following procedure. Given a program scope x , let $C_{\max}(x)$ return the child of x with the maximum value of metric m_I . Then, let the hotpath $H(x)$ for a scope x be the scopes identified by the following formula:

$$H(x) = \begin{cases} H(C_{\max}(x)) & m_I(C_{\max}(x)) \geq t \times m_I(x) \\ x & \text{otherwise} \end{cases} \quad (3)$$

That is, a hot path $H(x)$ extends from a scope x to include one of its children when the inclusive cost attributed of the child scope accounts for t or more of $m_I(x)$ (the inclusive cost attributed to x). In practice, we found a threshold of $t = 50\%$ to be most useful; the hot path ends at a scope when its inclusive cost is t or less of the parent’s inclusive cost.

Hot path analysis can be quickly applied in different subtrees (to identify the principal consumer of a cost within a subsystem) using different cost metrics; it is not just something that one applies to the root of the calling context tree. In `hpcviewer`, the threshold t can be adjusted through the `hpcviewer` preference dialog box. In Figure 3, hot path analysis detects a potential performance bottleneck in `chemkin_m_reaction_rate_` routine, where 41.4% of the inclusive cycles are spent computing reaction rates for the chemical species involved in the simulation.

D. Derived Metrics

Computer systems today typically provide access to a rich set of hardware performance counters that can directly measure aspects of program performance. Most common are counters in the processor core and memory hierarchy that enable the user to collect measures of work, resource consumption, and inefficiency. Some systems, e.g., Blue Gene P, also contain counters that enable the user to measure other events related to communication traffic. Despite the rich set of counters typically available, values of individual counters are of limited use by themselves. For instance, knowing the count of cache misses for a scope is useless. Only when combined with other information such as the number of instructions executed or the total number of cache accesses does the data become useful. While users do not mind a bit of mental arithmetic and frequently compare values in different columns to see how they relate for a scope, doing this for many scopes is exhausting. To address this problem, `hpcviewer` provides a mechanism for defining derived metrics. A derived metric is defined by specifying a spreadsheet-like mathematical formula

that refers to data in other columns in the metric table by using $\$n$ to refer to the value in the n^{th} column.

Combined with the other capabilities of `hpcviewer`, derived metrics are much more useful than they first appear. First, a column filled with a derived metric value can be used to sort scopes in the navigation pane. Being able to sort by a derived metric is *much* more useful than simply sorting by one of the terms upon which it was based and computing the derived values with mental arithmetic; sorting on derived metrics improves user productivity. Second, derived metrics can focus user attention on tuning opportunities rather than just the raw costs that measured metrics typically represent. For instance, rather than sorting scopes in a scientific program to find out where the most cost was incurred, for tuning it is often more useful to understand where the most important inefficiencies are. A good way to focus on inefficiency is with a derived *waste* metric. Depending upon what the user expects as the rate-limiting resource (e.g., floating-point computation, memory bandwidth, etc.), the user can define an appropriate waste metric (e.g., FLOP opportunities missed, bandwidth not consumed) and sort by that. Specifically, we have used a metric of floating-point waste, which we define as (the total number of cycles spent in a scope) \times (the peak number of floating-point operations per cycle that the processor supports) – (the actual number of floating-point operations executed in the scope) [3]. This metric tells us how many additional FLOPS could have been executed in a scope if we were always computing at peak rate. Sorting by this metric will rank order scopes to show that contain the greatest opportunities for improving overall program performance. This metric may highlight loops where

- a lot of time is spent computing efficiently, but the aggregate inefficiencies accumulate,
- less time is spent computing, but the computation is rather inefficient, and
- scopes such as copy loops that contain no computation at all, which represent a total waste of time according to the metric.

Beyond identifying opportunities for tuning with a waste metric, the user can compute a companion derived metric *relative efficiency* metric to help understand how easy it might be to improve performance. A scope running at very high efficiency will typically be much harder to tune than running at low efficiency. For our floating-point waste metric, we computed relative efficiency by dividing measured FLOPS by potential peak FLOPS. For scopes that rank high according to a waste metric, a relative efficiency metric can indicate the ease of improving the code.

VI. EFFECTIVE ANALYSIS

A. Effective Analysis with Derived Metrics

Here we show how derived metrics help to pinpoint and quantify scalability bottlenecks in context. We compute a derived metric that quantifies scaling loss by scaling and differencing call path profiles from a pair of executions [3].

Figure 6 shows an application of the aforementioned floating-point waste metric to a turbulent combustion code. By computing and sorting by the floating point waste metric, we discovered that the most floating-point waste (13.5%) was attributed to a flux diffusion loop that was streaming data through the memory hierarchy; code for this loop is shown in the source pane. The relative efficiency metric shows that this loop is running at 6% efficiency and represents a fat target for optimization. By using a tool to transform the loop nest to exploit data reuse in cache (by applying loop scalarization, fusion, unswitching, and unroll and jam), we were able to improve its running time by a factor of 2.9. The second scope shown in the figure represents a loop within the math library’s exponential routine.

The relative waste metric indicated that this was running at about 39% efficiency, which means that it is fairly tightly tuned. Using the Callers View view (not shown) to identify some of the contexts in which the exponential routine was used showed that there are opportunities for improving performance by using a vectorized version of the primitive, which makes better use of the instruction pipeline by filling it with multiple independent recurrences for separate computations.

B. Effective Analysis with Multiple Views

Often analysis begins with the Calling Context View to see if there is any calling context in the computation that particularly dominates in terms of cost. This can be done by using hot path analysis on a selected metric which then shows the call path of a potential performance bottleneck (if found). If not, the user typically moves to the Callers View to understand how much cost is incurred by each procedures at the top of the rank ordered list. In this case, the user typically investigates a few of the important contexts. Once the user knows what procedures and contexts are costly, the user can move to the Flat View to understand the costs associated with a procedure along with its loops and inlined code.

The three views play different roles. The Calling Context View provides a context-centric presentation from the callers’ perspectives. The Callers View provides a view of calling context from the perspective of each callee. The Flat View support detailed analysis of all costs incurred by a static context.

C. Load Imbalance Identification

Load imbalance is one of the most common scaling problems in single-program multiple-data (SPMD) scientific applications. It is caused by uneven distribution of work that forces some processes to idle between synchronization points. We have developed a scalable technique to visualize and identify load imbalance of parallel programs by using call path profiles and scalable post-mortem analyses [14].

As a case study, we choose PFLOTRAN, a scientific program for modeling multi-phase, multi-component subsurface flow and reactive transport using massively parallel computers [7]. We ran PFLOTRAN on the Cray XT5 partition of Jaguar, located at Oak Ridge National Laboratory’s National

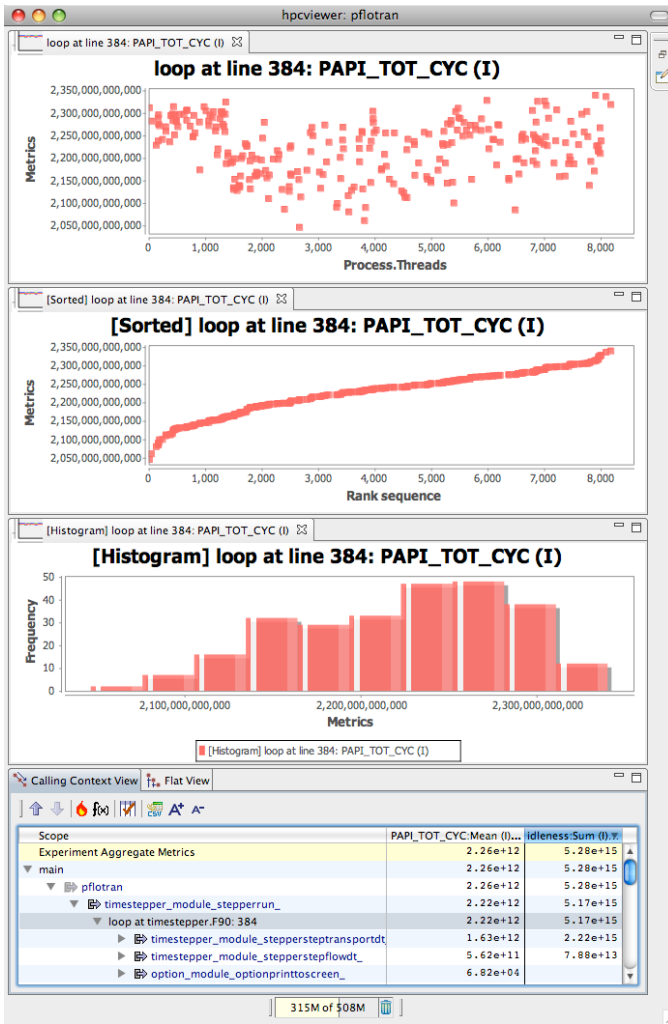


Fig. 7: A Calling Context View of PFLOTRAN’s load imbalance.

Center for the Computational Sciences. The PFLOTRAN test problem was a steady-state groundwater flow problem in heterogeneous porous media on an $850 \times 1000 \times 80$ element discretization with 15 chemical species per cell.

We can identify a load imbalance by sorting by total inclusive idleness summed over all MPI processes and performing hot path analysis to drill down into the potential load imbalance context, which is the main iteration loop at line 384 in `timestepper.F90`. As shown in Figure 7, the first graph from the top shows scattered inclusive total cycles. The second and the third graph shows the sorted metric and the histogram respectively, confirming that there is uneven work partition among processes.

Although this case study of an execution of PFLOTRAN is not an exhaustive analysis, this case study does illustrate how having aggregate idleness metrics attributed to each node of an execution’s canonical CCT can help pinpoint, quantify, and understand sources of performance load imbalance.

VII. SCALABLE PRESENTATION

Scalability is highly critical in any aspects of performance analysis tools, especially when the profile data is taken from a large scale parallel program that may run for days or even weeks on hundred thousands of cores. HPCTOOLKIT was designed with scalability in mind. Its performance measurement has significantly low overhead [15] and its data analysis is able to process thousands of MPI processes [14].

We have designed `hpcviewer` to support scalability as well. Data presentation in `hpcviewer` is based on tree-tabular presentation, which is generally more scalable than a graph-oriented presentation, both in rendering speed and visibility. Using a tree to represent calling contexts is much clearer than a graph for complex applications. Using table to represent metrics allows a user to select which metric to observe and to automatically search for a possible performance bottleneck.

`hpcviewer` is based on Eclipse and configured with *lazy-startup*, which means all components (such as text editors, graph and HTML viewer) are loaded when needed. Furthermore, in order to analyze large-scale application that runs with thousands of processors, we summarize metrics of all processors into mean, covariance, min and max [14], instead of displaying thousands of metrics. Finally, the Callers View is constructed dynamically, ensures scalability for both execution time and memory consumption since we store and process data only when needed.

VIII. RELATED WORK

Presentation of program performance can be either *graphical* or *tabular*. Tools that can be included in the first categories are CrayPat [9], VTune [6], LoopProf [8] and TAU [10]. Tabular-based data presentation tools include gprof [16], Intel PTU [5], parallel performance wizard [11], Apple’s Shark [2], Sun Studio [12] and `hpcviewer`. While graphical presentation style can be more appealing, we have found that tabular style is clearly more scalable and informative.

To the best of our knowledge, there is no other tool that support all three views. gprof [16], Parallel performance wizard [11], Sun Studio [12] and Apple’s Shark [2] all support the Calling Context View with inclusive and exclusive metrics. These tools, however, do not support Callers View. Intel PTU [5] supports Callers View and Flat View (without hierarchical structure), but no inclusive metric is supported.

Some tools support (semi) automatic performance analysis. CrayPat [9] provides a feature to automatically pinpoint and quantify load imbalance in parallel applications. It is also possible to automatically detect scalability problems in TAU by using PerfExplorer [4].

The importance of derived metrics is widely acknowledged. Intel PTU supports a kind of derived metrics to compare data between different experiments. TAU/PerfExplorer allows a much higher flexibility by using a script to define derived metrics and even to automatize repetitive tasks [4].

Some tools only support specific programming models or platform. Parallel performance wizard only supports some

PGAS languages [11]. Intel PTU [5], Intel VTune [6], Apple's Shark [2], Sun Studio [12] and CrayPat [9] supports only for specific platforms. In contrast, our toolkit is designed for language independent, application independent and problem independent.

In summary, while none of the key features of `hpcviewer` is strikingly novel when taken in isolation, our focus has been on what results from their combination. In `hpcviewer`, we have selected and married a relatively small set of complementary presentation techniques to form a coherent synthesis that is greater than the constituent parts.

IX. CONCLUSIONS AND FUTURE WORK

We developed `hpcviewer` as part of HPCTOOLKIT for performance analysis of serial and parallel codes. The features that it supports today grew out of our own experiences of trying to analyze and tune scientific applications. `hpcviewer` was designed based on four principles: (a) support of multiple perspectives which complement each other for observing profile data, (b) avoidance of visual clutter, (c) emphasis on potential performance bottleneck, and (d) scalability. These four principles are critical for guiding the user to perform analysis in a sea of performance data.

To our knowledge, `hpcviewer` is the only tool available that supports all three views of context-sensitive performance data: Calling Context View, Callers View, and Flat View. We have shown that each view complements each other and can discover different performance problems. `hpcviewer`'s top-down approach guides a user to focus on what is important, and its profile data is displayed in such a way as to reduce distractions. Our tool is also unique that it combines both static structure and dynamic call paths as shown in the Calling Context View and Flat View.

`hpcviewer`'s features effectively increase user productivity. Derived metrics quickly highlight tuning opportunities. The hot path analysis streamlines a repetitive task and rapidly highlights important program contexts.

`hpcviewer` is designed to be scalable. Its components are loaded on-demand, it supports dynamic creation of the Callers View, tabular-based data presentation and metrics summarization for parallel applications with large number of processors.

Overall, we believe that we have converged on a useful corpus of features that serves the intended purpose. Using our tools, often the user can pinpoint performance bottlenecks effectively, that need attention in a matter of minutes.

Ongoing work includes additional focus on scalability. As we apply `hpcviewer` to programs with large-scale parallelism, other issues that may be important are replacing our XML format for profiles with a more compact binary format, and enhancing `hpcviewer` so that it need not have data for all processes resident in memory at once.

Other ongoing work includes identifying data reuse patterns and suggesting program transformations to improve program performance. Another item of interest is effectively presenting metrics correlated with object code. Although HPCTOOLKIT

supports a simple text-based presentation of such information, it is cumbersome to use.

X. ACKNOWLEDGMENTS

HPCTOOLKIT would not exist without the contributions of the other project members: Mike Fagan and Mark Krentel.

This research used resources at both Argonne's Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357; and the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725

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*, 22(6):685–701, December 2009.
- [2] Apple Computer. Shark. <http://developer.apple.com/performance/>.
- [3] Cristian Coarfa, John Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. Scalability analysis of `spmd` codes using expectations. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 13–22, New York, NY, USA, 2007. ACM.
- [4] Kevin A. Huck, Oscar Hernandez, Van Bui, Sunita Chandrasekaran, Barbara Chapman, Allen D. Malony, Lois Curfman McInnes, and Boyana Norris. Capturing performance knowledge for automated analysis. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–10, Piscataway, NJ, USA, 2008. IEEE Press.
- [5] Intel Corporation. Intel PTU. <http://software.intel.com/en-us/articles/intel-performance-tuning-utility/>.
- [6] Intel Corporation. Intel VTune performance analyzers. <http://www.intel.com/software/products/vtune/>.
- [7] Richard Tran Mills, Chuan Lu, Peter C Lichtner, and Glenn E. Hammond. Simulating subsurface flow and transport on ultrascale computers using PFLOTTRAN. *Journal of Physics Conference Series*, 78(012051), 2007.
- [8] Tipp Moseley, Daniel A. Connors, Dirk Grunwald, and Ramesh Peri. Identifying potential parallelism via loop-centric profiling. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 143–152, New York, NY, USA, 2007. ACM.
- [9] Luiz De Rose, Bill Homer, and Dean Johnson. Detecting application load imbalance on high end massively parallel systems. In Anne-Marie Kermarec, Luc Bougé, and Thierry Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 150–159. Springer, 2007.
- [10] Sameer S. Shende and Allen D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.
- [11] Hung-Hsun Su, M. Billingsley, and A.D. George. Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.
- [12] Sun Studio. Sun Studio Performance Analyzers. <http://www.sunstudio.com>.
- [13] Nathan R. Tallent. *Performance Analysis for Parallel Programs: From Multicore to Petascale*. Ph.D. dissertation, Department of Computer Science, Rice University, March 2010.
- [14] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path probes. In *The 2010 ACM/IEEE Conference on Supercomputing (to appear)*, New York, NY, USA, 2010.
- [15] Nathan R. Tallent, John M. Mellor-Crummey, Laksono Adhianto, Michael W. Fagan, and Mark Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, pages 1–11, New York, NY, USA, 2009. ACM.
- [16] Dominic A. Varley. Practical experience of the limitations of `gprof`. *Software: Practice and Experience*, 23(4):461–463, 1993.