Bio-inspired call-stack reconstruction for performance analysis

Harald Servat*†, Germán Llort*†, Juan González‡§, Judit Giménez*†, Jesús Labarta*†
* Computer Sciences Department, Barcelona Supercomputing Center - Centro Nacional de Supercomputación
c/Jordi Girona, 31 - 08034 Barcelona, Catalunya, Spain
† Department of Computer Architecture, Universitat Politècnica de Catalunya
c/Jordi Girona, 1-3 - 08034 Barcelona, Catalunya, Spain
‡ IBM T.J. Watson Research Center
Yorktown Heights, NY 10598 USA
harald.servat@bsc.es, german.llort@bsc.es, jgonzale@us.ibm.com, judit.gimenez@bsc.es, jesus.labarta@bsc.es

***Abstract*—The correlation of performance bottlenecks and their associated source code has become a cornerstone of performance analysis. It allows understanding why the efficiency of an application falls behind the computer's peak performance and enabling optimizations on the code ultimately. To this end, performance analysis tools collect the processor call-stack and then combine this information with measurements to allow the analyst comprehend the application behavior.**

Some tools modify the call-stack during run-time to diminish the collection expense but at the cost of resulting in non-portable solutions. In this paper, we present a novel portable approach to associate performance issues with their source code counterpart. To address it, we capture a reduced segment of the call-stack (up to three levels) and then process the segments using an algorithm inspired by multi-sequence alignment techniques. The results of our approach are easily mapped to detailed performance views, enabling the analyst to unveil the application behavior and its corresponding region of code. To demonstrate the usefulness of our approach, we have applied the algorithm to several first-time seen in-production applications to describe them finely, and optimize them by using tiny modifications based on the analyses.

***Index Terms*—performance analysis, multi-sequence alignment, call-stack analysis, sampling, instrumentation**

I. Introduction

It is widely accepted that the applications' performance only reach a fraction of the supercomputers' peak performance. Should anybody recognize the performance flaws and their causes in a piece of software, they will realize that correlating performance and source code is essential. The correlation is not only necessary to understand the application behavior, but also to reduce its execution time. This fact is becoming more relevant as systems and applications grow in complexity. Also, because the scenario in which analysts know little about the applications and have to report on how to improve the applications' performance is becoming frequent.

Performance analysis tools such as TAU [19], Scalasca [25], Vampir [14], HPCToolkit [22] and Paraver [17], help find out application performance flaws and their responsible code counterpart. These tools rely on either instrumentation or sampling techniques to monitor the application as it progresses from one routine to the next. Instrumenting every routine may lead to exaggerated overheads when monitoring fine grain routines [8], [3], and so it is often required an additional execution to blacklist from the collection the tiny but frequently executed routines. Many tools have switched to sampling mechanisms to reduce the overhead while capturing information regarding the representative routines of an application. However, reporting routines without providing their calling context may lead to imprecise analyses [3], [23], and thus several performance tools gather the processor call-stack to provide calling contexts. Capturing the complete call-stack may be costly, and some tools have adopted ways to reduce the monitoring overhead by modifying the call-stack at run-time but at the cost of creating specific non-portable solutions.

We propose a novel offline mechanism that estimates the duration of the representative routines on optimized parallel binaries that include symbolic debug information. Our approach only grabs a small, fixed segment of the top of the call-stack during the application execution ensuring an easy and portable implementation. Then an algorithm inspired by Multiple Sequence Alignment (MSA) algorithms [15] processes the collected samples to estimate the duration of the routines and their chronological order of execution. These algorithms infer biological sequence homology to conduct phylogenetic analyses and study shared lineages from specimens with common ancestors, but in this research they serve as an inspiration to deduce the line of common call-path ancestors. The output of this mechanism is combined afterwards with the framework developed by Servat *et al.* in [18]. This framework takes advantage of the repetitive nature observed in many HPC applications to report instantaneous performance along computing regions, but it lacks correlating the performance with the application code.

The contributions of this paper include:

- a portable technique to reconstruct the call-stack from small segments using an MSA inspired algorithm,
- a procedure to automatically estimate the most representative routines based on the call-stack reconstruction, and
- an extended framework that shows the chronological or-

[§]Most of the contributions made while employed by Barcelona Supercomputing Center - Centro Nacional de Supercomputación.

der of execution of the application source code (including source code lines) and its node-level performance.

The remaining of this paper is organized as follows: Section II contextualizes our method concerning previous existing tools. Then Section III describes the MSA inspired algorithm and the integration of its results into the framework that finely describes the performance evolution of computing regions. Section IV serves to validate our combined approach by comparing the results obtained with other state-of-the-art performance tools. We demonstrate the usability of our work on several in-production applications in Section V. Finally, Section VI draws the conclusions and discusses possible future research topics.

II. Related Work

Call-path profiling is a technique that is crucial to correlate performance and its corresponding code region within the application and consequently this topic has received much attention. Here we describe the related research that collects this information focusing on those approaches that use sampling techniques.

`gprof` [10] is the *de facto* profiling tool, and it combines compiler-assisted instrumentation capabilities to provide function call count and sampling to estimate the function average duration. As a consequence of relying on instrumentation, this tool shows significant overheads (>90%) when instrumenting every routine in the application [8]. Although `gprof` provides a call-graph representation, the time attributed to a function is independent of the calling context. `perf` [4] captures the time, the performance counters and the call-stack at sample points so as to put the blame on application routines for the performance bottlenecks.

Since call-stack inspection imposes certain overhead, some tools have focused on modifying the call-stack for a more efficient call-stack traversal. Whaley extends the execution environment so that the frames contain a bit indicating whether they have been unwound to save time when exploring the call-stack [24]. The introduction of the *trampolines* improves the technique by modifying the call-stack return address to call the measurement system while employing a *shadow* stack to query the call-path prefix [2]. Tools such as Scalasca and HPCToolkit have adopted this mechanism. For instance, Szebeny and others use trampolines to combine sampling mechanism with MPI activity-driven tracing in the Scalasca tool-suite [21]. With this method, they are not only able to identify previously explored stack frames and establish their prefix, but they also benefit from the Scalasca's ability to configure which MPI routines should unwind the call-stack.

The main drawback of the trampolines is that their implementation is system-dependent, resulting in non-portable solutions. Other researchers have explored alternatives that do not modify the call-stack. For instance, ICPP [13] proposes a technique that evaluates the top of the call-stack along with the call-stack size. The mechanism matches the top of the call-stack on a previously generated call-graph created in earlier preparation runs. Perks *et al.* present a heuristic to predict the overlap between consecutively collected call-stacks [16]. With such a heuristic, they estimate the similarity between successive call-stacks, removing the need for collecting the full call-stack.

The latter technique is somewhat similar to the work we propose in which we approximate the duration of the relevant routines within a parallel application by capturing a small, fixed portion of the call-stack through statistical sampling. Then the data is integrated into a framework that detects repetitive code regions using minimal instrumentation (e.g. PMPI [6]) and it later smartly combines coarse-grain sampled with instrumented metrics to provide a detailed progression of the performance metrics within these regions. As a result, our approach exposes chronological execution of the routines (a topic typically omitted in profile-based tools) and correlates with the nature of the performance bottlenecks when collecting few levels of the call-stack.

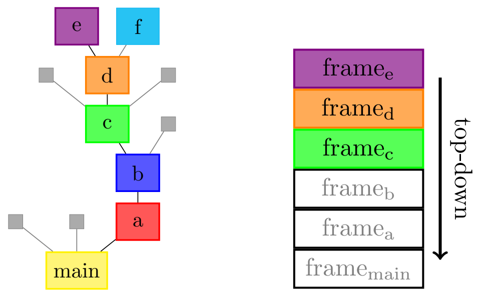
III. Description

This section describes the mechanism we have developed to estimate the duration of the relevant routines. The mechanism consists of two steps. The first step comprises the call-stack collection mechanism that samples the top-most stack frames periodically during the application run. The second step approximates the duration of the routines from the collected data from the collected call-stack data. Finally, we discuss the benefits of integrating this mechanism into a previously existing framework that reports performance metrics at every instant during delimited computing regions.

A. Call-stack monitoring

The call-stack is a data structure maintained by the processor and updated every time the application enters (or leaves) a subroutine by creating (or destroying) stack frames. These frames contain information such as the arguments received, the return address back to the routine's caller and space for local variables. *Call-stack unwinding* refers to the process of inspecting and accessing this data structure, and allows tools to emit relationships between callers and callees. The libunwind library [1] is considered a standard mechanism for call-stack traversals, due to its portability and accessible nature. Then, this call-stack information is converted into human-readable information that includes the routine name, its container filename and the source code line through the compiler debugging information and the binutils package [7].

This step collects a fixed portion of the top of the call-stack using sampling mechanisms to generate a sequence of time-stamped call-stack segments. There are three reasons for a top-down collection (as depicted in Figure 1). First, the collected frames point to functions that are closer to where the activity occurs, helping the tool to point out what routines were executing during the monitoring. Second, it is trivial to implement this collection mechanism when using the libunwind infrastructure, which ensures the portability of this mechanism. Finally, the collection expense becomes reduced and constant irrespective to the call-stack depth.



(a) (b)

Fig. 1: Top-down selection (1b) from a given call-stack tree (1a) when routine e is active.

B. Estimate the routine duration

This step is divided into two phases. The first phase searches for common call-stack frames in the sample time-series by aligning the frames using a process similar to MSA and generates a matrix that represents the active routines during the execution. The second phase filters out those routines that are very short because they may not be meaningful for the analysis and also because presenting many small routines may overwhelm the tool user.

1) Call-stack alignment: Despite the benefits from collecting the top of the call-stack, this approach presents difficulties when identifying routines in the call-stack because it evolves in conjunction with the routine entries and exits; therefore the depth for a particular routine varies along the application activity. Figure 2a exemplifies this problem showing a timeline of the captured top call-stacks related to the application with the call-tree depicted in Figure 1a. In Figure 2a, the first sample (s_1) has been interrupted while executing routine f_d that was invoked by f_c, and at the same time, f_c was called by f_b. Since the depth for a particular routine (e.g. f_b) varies over samples (and time), and it might not even appear sometimes, the call-stack analysis must be aware of this issue.

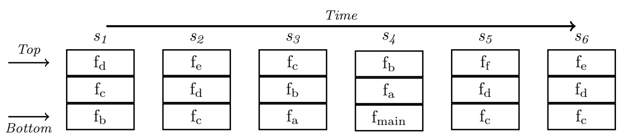
The first phase consists of reconstructing the call-stack based from the reduced captured frames contained in the samples by aligning the shared frames on a common symbol. To implement it, we have developed an algorithm inspired by MSA methods. Sequence alignment is a mechanism to arrange sequences of biological molecules (*i.e.*, DNA, RNA or proteins) and identify regions of similarity between the sequences. The simplest alignment involves a pair-wise sequence comparison [20], but this idea is extended to multiple sequences, in which the mechanism searches for the best matching on any number of sequences. The *naïve* alignment of n sequences of length m implies applying a pair-wise alignment of sequences and it results in a $n \times m$-dimensional matrix where each row represents a sequence. Our approach is similar to MSA but in our case the sampled call-stacks substitute the sequences. Honoring this metaphor, Figure 2a represents the starting point of an alignment of six call-stack samples. Thus, the mechanism arranges time-stamped routines pointed by frames of the call-stack.

There is a fundamental difference comparing the two approaches when applying the pair-wise alignment: the presence of mutations. In the biology field, mutations are either point mutations (*i.e.* molecules have been replaced by other molecules) or indels (insertions or deletions of molecules in a sequence). Our approach considers neither point nor deletion-derived mutations because the call-stack frames are not subject to modifications and because the instrumentation package captures a consecutive fragment of the call-stack. However, the approach considers inserting mutations to extend the portion below the common call-stack between two samples if they have a common call-stack portion. Consider samples s_1 and s_2 from Figure 2a, in which f_d and f_c are common and intuitively one can extend s_2 to include f_b at the bottom. These insertions (henceforth gaps) are denoted as $\circ(X)$ where X refers to the frame to the routine X. A two-step process implements this approach. Algorithm 1 describes the first step. The Algorithm searches for a common routine (named *pivot* hereafter) among the samples and then applies a pair-wise alignment to consecutive samples that contain the pivot. The second step applies a similar algorithm (not shown) to the remaining samples not containing the pivot but at least share one symbol with those samples containing the pivot and thus increasing the number of used samples in the process. This step could be repeated until no more samples are added, but in our implementation we only execute it once.

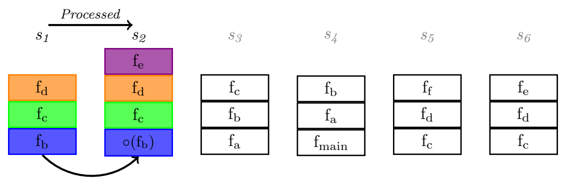
Algorithm 1 Pseudocode for the call-stack alignment.

```
procedure CALLSTACKALIGNMENT(vs)
Require: vs: vector(Samples) sorted by timestamp
    $pivot \leftarrow vs.searchMostFrequentRoutine()$
    $vtmps \leftarrow vs.contain(pivot)$
    for $i \leq vtmps.size()$ do
        $c\_c \leftarrow vs(i).getCallstack()$            ▷ Current call-stack
        if $i > 1$ then
                                                        ▷ Previous call-stack
            $p\_c \leftarrow vs(i-1).getCallstack()$
            $h\_cc \leftarrow c\_c.height(pivot)$
            $h\_pc \leftarrow p\_c.height(pivot)$
            if $h\_cc < h\_pc$ then
                $gaps \leftarrow \{p\_c.subset(h\_pc - h\_cc)\}$
                $c\_c.InjectGaps(gaps)$
                $mat(i) \leftarrow c\_c$
            else if $h\_cc > h\_pc$ then
                $gaps \leftarrow \{c\_c.subset(h\_cc - h\_pc)\}$
                for $j < i$ do
                    $mat(j).InjectGaps(gaps)$
                end for
            end if
        else
            $mat(i) \leftarrow c\_c$
        end if
    end for
    return $mat$
end procedure
```

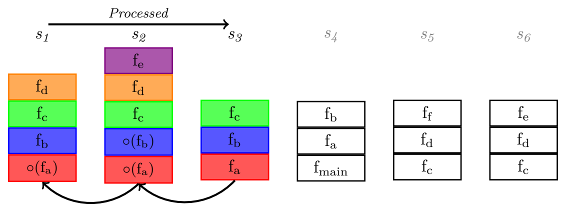
Figure 2 illustrates how the algorithm generates the alignment matrix. First, the pivot in the example is f_c because it is the most frequent routine and it serves as a reference for the alignment. Next, the process selects the samples that



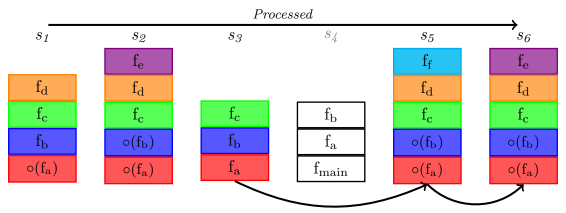
(a) Samples as collected. f_c is the pivot.



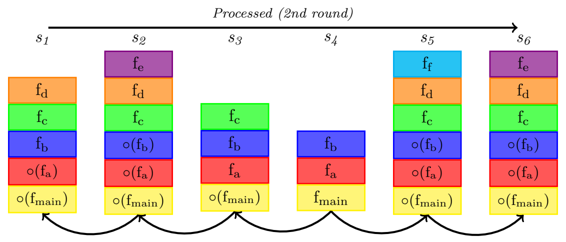
(b) Alignment of s_1 and s_2. Gap injection into s_2.



(c) Alignment of s_2 and s_3. Gap injection into previous samples.

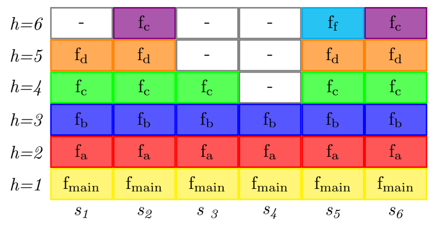


(d) Alignment of s_3 and s_5, s_6. Double gap injection into s_5 and s_6.

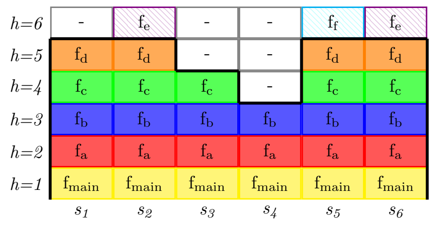


(e) Processing the remaining samples s_4 that do not contain f_c. Need to inject gaps into the other samples.

Fig. 2: Example of alignment of a set of samples.



(a) Matrix representation of the call-stacks depicted in Figure 2e.



(b) Selection of routines of interest with a threshold≥2 samples.

Fig. 3: Post-processing the alignment samples to choose the routines of interest.

contain the pivot within their call-stacks (s_1, s_2, s_3, s_5 and s_6), and then aligns these samples. The process applies a pair-wise align to consecutive samples and injects as many gaps as needed to line the pivot up. Consider the alignment of samples s_1 and s_2 from Figure 2a. In this case, both samples contain the pivot routine at a different depth. The pivot in s_1 is above when compared to s_2, so the algorithm adds a gap at the bottom of the call-stack of s_2 according to the condition in line 11 of the Algorithm (and as illustrated in Figure 2b). When aligning s_2 and s_3 the algorithm behaves differently because the pivot in s_2 is below compared to s_3. In this case (controlled by the conditional statement in line 15), the gaps are back-propagated to those samples that occurred before s_3 as depicted in Figure 2c, to propagate the common call-path. Due to space restrictions, the consecutive alignment for samples s_3 and s_5, and s_5 and s_6 is depicted in Figure 2d. The second step aligns the samples that do not contain the pivot but share a symbol of the call-stack with the previously treated samples. In the example, sample s_4 in Figure 2e has been ignored because it does not contain the pivot, but since its call-stack has routines in common with other processed samples (f_a and f_b) it is aligned in this second step. In this case, the call-stack data from s_4 shows that its bottom frame points to the routine `main`, so the alignment injects a gap into the rest of the samples.

2) *Selecting representative routines:* After generating the alignment matrix, the second phase explores the matrix to identify the representative routines. In this direction, we consider the representative routines those that are closer to the top frames that span at least for a user-given duration (either in time or number of consecutive samples). This selection ensures that the mechanism focuses on identifying a small set of routines that does not overwhelm the analyst providing lots of tiny routines that may be invoked lots of times. This search processes the aligned call-stacks as if they were stored in a matrix where columns refer to samples ordered by their collection time and rows point to the depth of the call-stack frame. The algorithm looks for the number of consecutive frames at the lower level (h=1) that point to the same routine. If the number of consecutive samples surpasses the given threshold, then the process is applied recursively to the upper level (h=h+1) within the range defined by samples that contain the frames pointing to the same routine. For instance, consider the matrix shown in Figure 3a which shows the results from Figure 2e and apply the selection with a threshold of two consecutive samples. The matrix analysis starts at level

h=1 and since all the samples point to the same routine in the interval $s_1 - s_6$ and the number of samples is higher than the threshold, the next level (h=2) explores the same interval. Similarly, the process explores levels h=2 and h=3. When processing the frames at h=4, the mechanism splits the recursive analysis into two parts (one involving samples s_1-s_3 and the other involving s_5-s_6). The analysis finishes at h=6, where routines $\mathrm{f_e}$ and $\mathrm{f_f}$ are ignored because they only last one sample. In the end, the selected portion of the matrix represents the active call-stacks with a particular granularity, as well as, the top of the selection denotes the active routines. Figure 3b illustrates the result using a solid color and a black line to show the selected routines and their heights.

C. *Correlating source code and performance metrics*

We have enhanced the framework described in [18] with the bio-inspired algorithm to associate the source code information to its performance. The resulting framework generates views that detail the performance progression along computing regions, yet the framework also generates trace-files. The computing regions can be manually delimited using instrumentation, or automatically detected by the framework. In the latter case, a computing region is defined as the user code in between successive parallel programming calls (such as MPI or OpenMP) and a clustering algorithm groups these regions according to their performance metrics (typically the number of instructions and the instruction rate) [9]. Then, the framework applies a mechanism named folding that combines coarse grain sampled and instrumented information to provide detailed node-level performance within a computing region. In the context of the folding process, the samples are gathered into a synthetic region by preserving their relative time within their original region so that the sampled information determines how the performance evolves within the region. Consequently, the folded samples represent the progression in shorter periods of time no matter the sampling frequency.

The combination benefits both parts creating synergies between them. First, the selection of the pivot directly influences the results because it determines which samples are aligned and which are ignored. This fact becomes critical in applications that progress through many routines and share a few calling routines in the captured samples. In this sense, it is advantageous to apply the alignment to code regions with similar performance characteristics because they are likely to refer to the same code and help to identify the call-stacks. Second, the monitoring period also affects the results because the lower sampling rate, the lesser call-stack activity captured and the call-stacks captured may differ substantially. In this direction, the folding mechanism ensures that the aligned samples refer to adjacent code references and consequently adding the minimum gaps. Finally, the framework gains the ability to depict the instantaneous progression of the source code in conjunction with the node-level performance. This progression does not only cover the temporal progression of different routines, but also the source code line progression within them.

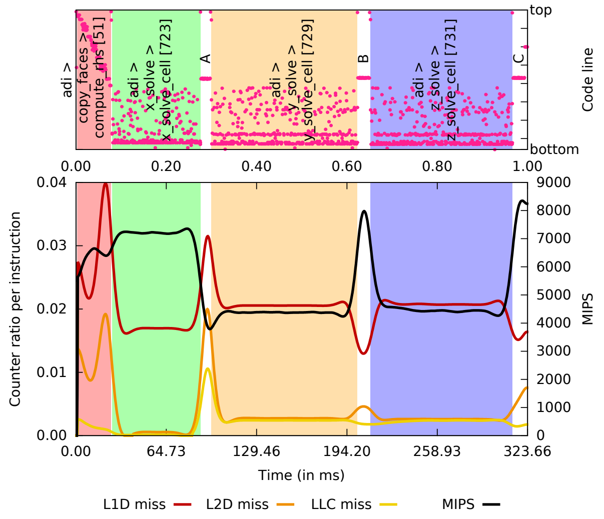
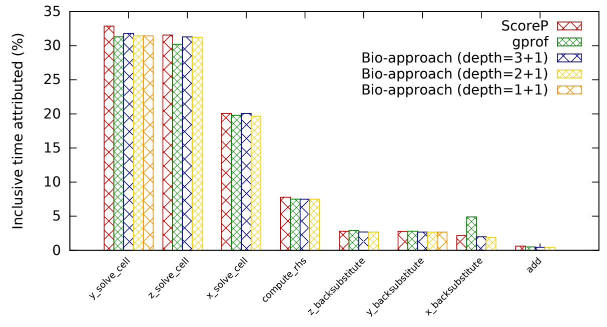


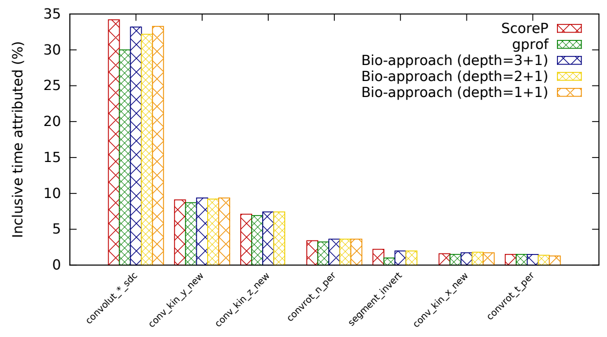
Fig. 4: Source code references and performance metrics collocated for the MPI version of the NAS bt.A benchmark executed with one process.

1) Practical example: We use the parallel MPI NAS bt.A benchmark to exemplify the usage of the framework. The application has been executed in an Intel Xeon E5-2670 processor and sampled three levels of the call-stack and performance counters at 20 Hz. Despite the benchmark is parallel, and the clustering tool can identify computing regions in between MPI calls, we opt to instrument manually the main loop (the `adi` routine) for clarification purposes so that the folding results represents the whole iteration.

Figure 4 shows the results for the complete framework. The plot contains two subplots representing the progression along the instrumented region for the routines and a profile of source code lines (at the top), and the node-level performance metrics (at the bottom). That is, the X-axis refers to the time since the start of the delimited region (`adi`) until the region ends. Both plots share a distinctive background color according to the active routine. Due to plot rendering limitations, the short routines (as in the transition from $\mathrm{s_3}$ to $\mathrm{s_4}$ in Figure 3b) are depicted with a white background, yet they can be analyzed using the trace-file. Similarly for size rendering limitations, the plot at the top displays the name of the active routines and up to two calling ancestors (in the form of X >Y >Z [n], where Z is the active routine, X and Y refer to the ancestors and n is the most observed code line within the active routine), yet an expanded calling context could be retrieved. For instance, the second phase (shown in green) represents the routine `x_solve_cell`, which is called by `x_solve` and invests most of its time on line 723. This plot also shows pink points representing a time-based profile of the code lines within the active routine (where the top refers to the upper of the file that contains the routine). The solvers (`*_solve_cell`) present almost a random line progression limited to the half bottom of the plot, which indicates the presence of a loop that covers the lower half of the file and that spans for the whole execution of the routine. We observe



(a) NAS bt.B benchmark.



(b) BigDFT application.

Fig. 5: Comparison of the code attribution with other performance tools and using different levels of call-stack unwind when applied to different applications.

that `x_solve_cell` lasts less than the rest and that most of the samples point to one line at the bottom of the source file while other solvers have mainly sampled two lines at the bottom of the file. Finally, routines `*_backsubstitute` (depicted in white and manually labeled as A, B and C) invest most of their time in one line, which observing the source code corresponds to a single statement within the five-nested loop that comprises the routines. Concerning the performance, the black line refers to the MIPS rate and it is shown on the right Y-axis while the remaining lines depict the ratio of cache misses per instruction at different levels of the cache hierarchy and are displayed on the left Y-axis. The MIPS rate of the solvers is uniform, being higher on `x_solve` due to less cache miss ratios in L2D and Last-Level Cache (LLC). This behavior is the opposite on the `*_backsubstitute` routines, where `x_backsubstitute` (A in the plot) runs slower due to an increase in the L2D and LLC miss ratios.

IV. Validation

We have evaluated whether the results provided are similar to those obtained with other performance tools that rely on different measurement techniques (Score-P [12] using direct instrumentation, and gprof using compiler-based instrumentation and time-based sampling running at 100 Hz). The validation aims at detecting how the depth of the call-stack

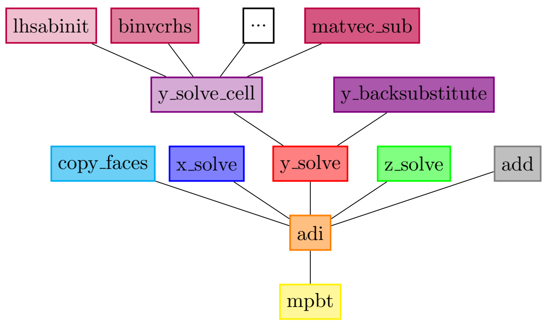


Fig. 6: Summarized call-graph for the NAS MPI bt benchmark.

unwound influences the results. We use the NAS MPI bt.B benchmark and the BigDFT application and some additional experiments indicate that their call-stacks are 6-7 and 15-20 levels deep, respectively, in most of the situations. The NAS benchmark has been executed on a cluster based on Intel Xeon E5-2670 processors using one process to avoid network noise while BigDFT has been executed using 1,024 MPI ranks in a Blue Gene/Q system. The comparison determines the most representative routines (concerning percentage of time) using gprof and Score-P instrumentation. Despite Score-P supports automatic instrumentation on the Blue Gene/Q installation, the installation in the Intel cluster lacks such feature. Thus in the latter system, we manually instrument the routines identified by gprof using the Score-P's API. The values reported by these monitoring systems are compared to the results of our approach when monitoring the application using MPI instrumentation and sampling it with a coarse frequency (25 Hz) and collected different segment lengths.

Figure 5 shows the results of the comparison on plots that show applications routines on the X-axis and the time attributed to each routine in the Y-axis. Notice that the call-stack depth is expressed as $X+1$, meaning that the sampling handler has unwound X frames of the call-stack and also has emitted the interrupted PC address provided by the sampling handler. In general, the reader may observe that the results from our approach using $depth{=}2{+}1$ are similar to those obtained by Score-P and gprof. Using a smaller value for *depth* does not attribute time to some of the routines, yet provide good approximations for many cases. For instance, we notice in Figure 5a that our approach only provides measurements for `y_solve_cell` and `y_backsubstitute` when $depth{=}1{+}1$. This issue occurs because the routine `y_solve_cell` becomes the pivot in the algorithm and because of the structure of the application (depicted in Figure 6). Since our approach applies to those call-stacks containing the pivot, when $depth{=}1{+}1$ the mechanism only finds samples containing invocations from `y_solve`. Then, the second part of the algorithm (that looks for common frames) adds `y_backsubstitute` to the common frames. However, it ignores `adi` because the collecting mechanism has not captured samples at `y_solve` (invoked from `adi`); thus the mechanism does not explore the call-stacks that contain this

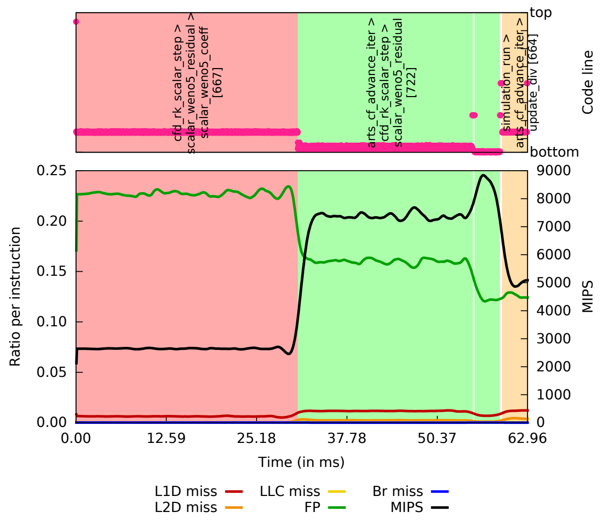
TABLE I: Applications analyzed

	Arts_CF	**Nemo** v3.4
# Processes	512	128
Processor type	Intel®Xeon$^{\text{TM}}$E5-2670 at 2.60 GHz (3.30 GHz with TurboBoost)	
Application size	338 Klines 564 files	221 Klines 484 files
Compiler	Intel Compiler v14.0.2	Intel Compiler v13.0.1
Compiler flags	-O3 -xAVX -g	-O3 -g
Sampling frequency	20 Hz	

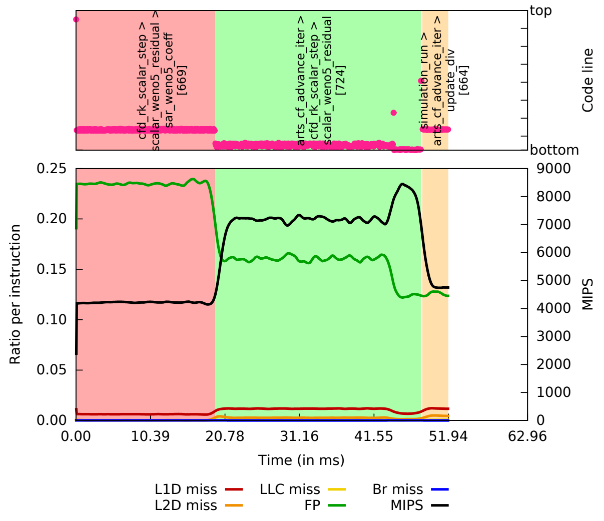
routine. Similarly occurs in BigDFT when $depth$=1+1 which only discloses five from the seven most representative routines. It is also worth mentioning that BigDFT experiences significant overheads instrumenting every routine through gprof and Score-P because of the huge amount of visits to certain routines, so we have used the Score-P filtering capabilities to reduce the expense. Nonetheless, the duration reported for the most time-consuming routines are similar comparing all approaches. The exception is gprof when reporting the duration of `convolut_kinetic_slab_sdc`, which is slightly shorter than the rest most probably due to the higher overhead imposed during the application run.

V. Experimental analysis

We have evaluated the performance of two first-time seen in-production applications to demonstrate the usefulness of our solution. We have used the framework to analyze these applications, including the automatic detection of the most time-consuming computing regions delimited by consecutive MPI calls, applying the folding mechanism, and executing the discussed alignment algorithm. Using the enhanced framework we are not only able to identify the nature of the performance bottlenecks using the performance counters but also correlate these bottlenecks with the associated code. The first application is Arts_CF [5], which implements a variable density, conservative and arbitrarily high order finite difference method to simulate flows in complex geometries with cylindrical or cartesian non-uniform meshes. The second application is Nemo [11]; an ocean model that includes several components besides the ocean circulation, including sea-ice and biogeochemistry. Table I contains some application characteristics, their compilation flags, the execution characteristics, and the sampling frequencies. We outline that both Arts_CF and Nemo are large in terms of lines of code containing more than 100K lines. They have been compiled using aggressive compiler flags and contain debugging information so that the binaries allow translating call-stack frame addresses into full source code references. Concerning the monitoring, we have captured performance measurements at MPI points and sampled the performance counters and three frames of the call-stack at 20 Hz ($5x$ smaller than the gprof sampling rate, 100 Hz). Despite the monitoring captures many performance counters using multiplexing techniques, we only report those that are meaningful for the analysis. The metrics include, at least: the instruction rate (depicted in black and referenced on the right Y-axis), and the L1D, L2D, LLC miss ratios per instruction (on different colors and shown on the left Y-axis).



(a) Original version.



(b) Modified version.

Fig. 7: Source code and performance correlated for one of the most dominant computing regions in Arts_CF. Both time-lines are at the same time-scale to ease the comparison.

A. Arts_CF

We focused our analysis on one of the most dominant computing regions of the application according to the clustering tool, which represented up to 17.6% of the application execution time. Figure 7a depicts the combination of the call-stack and performance counter results for this region. The Figure indicates that the computing region comprises three phases during its execution. The code line profile indicates that a small number of lines are responsible for the performance behavior observed. The first phase, colored in red, is mainly devoted to line 667 within the routine `scalar_weno5_coeff`, lasted approximately 30 ms and ran at 2,600 MIPS (less than 20% of the peak performance). The second phase, depicted in green, ran for 26 ms at 7,300 MIPS and correlated with line 722 in the routine `scalar_weno5_residual`, mostly. The last phase, colored in yellow, executed for 3 ms while running at 4,800 MIPS, and correlated with the routine `update_div`.

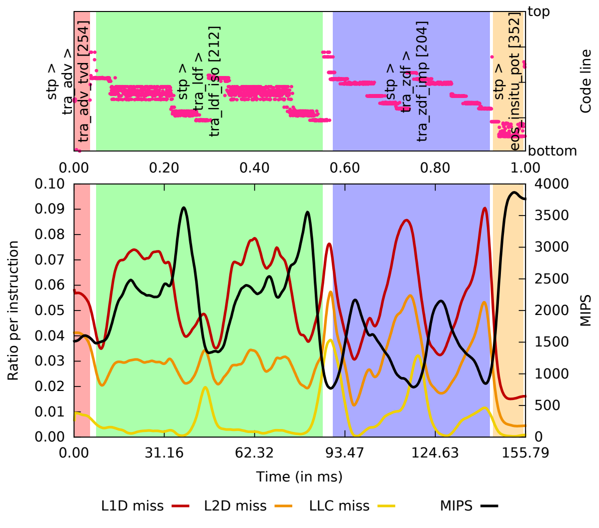
The analysis of the performance metrics in the first phase indicates that neither the cache hierarchy nor the branch predictor is limiting the performance because less than 2% of the instructions miss at L1D cache and less than 1% of the instructions fail on the branch predictor. However, the results point out that most of the instructions are floating-point instructions. The analysis of the stalled cycles and their categories (not shown due to space restrictions) indicates that 75% of the cycles stalled, and 40% of the stalled cycles are due to lack of re-order buffer entries. As we did not have access to the sources, we asked the developer to search for this code region. The exploration of the code showed that line 667 was the innermost sentence within a four-nested loop that included a floating-point division in which the divisor was invariant. It is well-known that floating-point divisions take longer to complete than multiplications, so we suggested to calculate the reciprocal of the divisor before this loop and multiply by the reciprocal within the loop.

A similar performance analysis (not shown) was applied to another computing region that represented an additional 18.3% of the computation time. Approximately 80% of this computing region ran uniformly at 2,600 MIPS, and the source code pointed to the innermost statement of the loops in lines 539, 545, 600 and 606 within the routine `scalar_weno_coeff`. Since the performance symptoms were similar to those detailed in the first phase of the previous region, we proposed the same code changes to the loops pointed in this region.

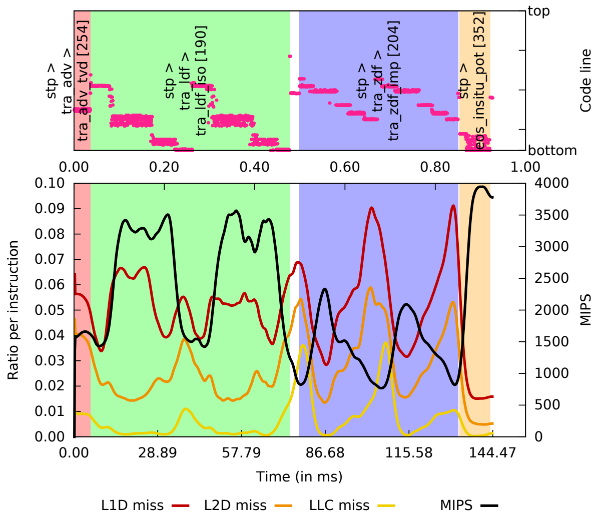
We re-applied the analysis to the modified binary and the results are depicted in Figure 7b. The duration of the first region decreased by 17.4% because the instruction rate in the first phase increased to 4,200 MIPS approximately. A similar increase in the instruction rate occurred in the second computing region explored, leading to a 28.3% of reduction on its duration. In overall, the application execution time was reduced by 9.0%.

B. Nemo

This analysis focuses on the most time-consuming region of Nemo, which took approximately 14% of the total execution time. Figure 8a shows the combined performance and source code reference results generated by the mechanism for this region. The results indicate that most of the time was invested in `tra_ldf_iso` and `tra_zdf_imp` routines (colored in



(a) Original version.



(b) Modified version.

Fig. 8: Source code and performance correlated for the most dominant computing region in Nemo. Both time-lines are at the same time-scale to ease the comparison.

green and blue) in lines 212 and 204, respectively. The code line profile displays a pattern of two iterations on both routines. With respect to `tra_ldf_iso`, it mostly executed a loop within lines 173-313. This loop contained a nested loop (in lines 207-253) in which 7% of the instructions missed at L1D cache due to the access to one 4D matrix, eleven 3D matrices, and several 2D matrices. Regarding `tra_zdf_imp`, the source code line profile indicated that most of the computation time was invested in the loop delimited by lines 161-280. Within this loop, there was a nested loop in lines 271-277 that achieved the worst performance (less than 1,000 MIPS) because of the high L1D cache miss ratio (8%). The code within this nested loop updated a 4D matrix where each element accessed to four matrices, and one these matrices is accessed twice in two different planes.

After analyzing the application code in `traldf_iso.f90`, we found that the loops in lines 245-251 and 303-311 did a similar job: they calculated the divergence of fluxes. These calculations accessed to the same data structures, performed similar operations and updated the same 4D matrix. Since these structures remained untouched from one loop to the next, we fused them with the aim to reduce the number of instructions executed and improve the temporal locality of the data. The loop in lines 207-253 calculated two 2D matrices (`zdkt` and `zdk1t`) before entering into the nested loops but the loop bodies only accessed to the elements (i, j), $(i{+}1, j)$ and $(i, j{+}1)$. Therefore, we changed the code to calculate these three elements and stored the results into scalar variables to reduce the pressure on the cache. When exploring the results for this modified version, shown in Figure 8b, we found out that the region took 7.2% less time to execute and increased its average instruction rate from 1,950 to 2,200 MIPS. The last plot also unveils that the code structure in `tra_ldf_iso` was different from the original version as a result from the modifications and the ran was faster in terms of MIPS because of the lesser L2D misses.

VI. Conclusions and Future work

We have presented a mechanism that estimates the duration of the representative routines, shows their chronological order and displays the evolution within the source code. The mechanism is inspired by MSA algorithms to align reduced segments call-stack segments captured by a portable process that impacts with a uniform, reduced cost during the measurement. This mechanism has been integrated into an existing framework allowing analysts to correlate the code and the nature of the performance bottlenecks easily without requiring them to know the application *a priori*. We have evaluated the accuracy of the mechanism using a set of benchmarks and have observed that collecting segments containing three call-stack frames suffice to attribute the time spent on the most time-consuming routines. We have also demonstrated the usefulness of the extension by finely characterizing the performance of the most dominant regions of two first-time seen in-production optimized applications. While these applications are hundreds of thousands of lines long, the characterization has pointed us to a tiny number of lines of code and to the bottlenecks they experience. Despite the applications had already been compiled with aggressive optimization flags, the analyses and simple code transformation derived from the framework results resulted in gains up to 9.0%.

We believe that there are further research opportunities using this framework. For instance, it could be worth to explore the results of this extended framework to identify the loops within the time-based profile automatically. Also, the framework could be extended by adding expert systems that guide non-expert users by providing insights on which code transformations may improve the application performance.

Acknowledgments

We thankfully acknowledge Mathis Bode for giving us access to the Arts_CF binaries, and Miguel Castrillo and Kim Serradell for their valuable insight regarding Nemo. We would like to thank Forschungszentrum Jülich for the computation time on their Blue Gene/Q system. This research has been partially funded by the CICYT under contracts No. TIN2012-34557 and TIN2015-65316-P.

References

[1] The libunwind project. https://savannah.nongnu.org/projects/libunwind.

[2] M. Arnold and P. Sweeney. *Approximating the Calling Context Tree Via Sampling*. Research report (International Business Machines Corporation). IBM T.J. Watson Research Center, 2000.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, 1996.

[4] A. C. de Melo. The new linux "perf" tools. In *Linux Kongress*, 2010.

[5] O. Desjardins et al. High order conservative finite difference scheme for variable density low mach number turbulent flows. *Journal of Computational Physics*, 227(15):7125 – 7159, 2008.

[6] M. P. I. Forum. *MPI: A Message-Passing Interface Standard*, chapter 8: Profiling Interface, pages 198–203. 1994.

[7] Free Software Foundation. GNU Binutils. http://www.gnu.org/software/binutils.

[8] N. Froyd et al. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th International Conference on Supercomputing*, pages 81–90, 2005.

[9] J. González, J. Giménez, and J. Labarta. Automatic Detection of Parallel Applications Computation Phases. In *IPDPS'09: 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.

[10] S. L. Graham et al. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[11] G. Madec. Nemo ocean engine. Technical report, 2008. http://www.nemo-ocean.eu/content/download/5302/31828/file/NEMO_book.pdf.

[12] D. Mey et al. Score-P: A Unified Performance Measurement System for Petascale Applications. In *Competence in High Performance Computing 2010*, pages 85–97. 2012.

[13] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. In *24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 175–190, 2009.

[14] W. E. Nagel et al. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[15] C. Notredame. Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, 3(1):131–144, 2002.

[16] O. Perks et al. Exploiting spatiotemporal locality for fast call stack traversal. In *26th International Conference on Supercomputing*, 2012.

[17] V. Pillet et al. Paraver: A tool to visualize and analyze parallel code. *Transputer and occam Developments*, pages 17–32, April 1995.

[18] H. Servat et al. Unveiling internal evolution of parallel application computation phases. In *International Conference on Parallel Processing, 2011*, pages 155–164, 2011.

[19] S. S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[20] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[21] Z. Szebenyi et al. Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs. In *Proc. of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 640–648, 2011.

[22] N. Tallent et al. HPCToolkit: performance tools for scientific computing. *Journal of Physics: Conference Series*, 125(1):012088, 2008.

[23] D. A. Varley. Practical experience of the limitations of gprof. *Softw. Pract. Exper.*, 23(4):461–463, 1993.

[24] J. Whaley. A portable sampling-based profiler for java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87, New York, NY, USA, 2000. ACM.

[25] F. Wolf et al. Usage of the SCALASCA for scalable performance analysis of large-scale parallel applications. In *Tools for High Performance Computing*, pages 157–167, 2008.