Automatic Instrumentation Refinement for Empirical Performance Modeling

Jan-Patrick Lehr*, Alexandru Calotoiu†, Christian Bischof* and Felix Wolf†
*Institute for Scientific Computing, †Laboratory for Parallel Programming
Technische Universität Darmstadt, Darmstadt, Germany
{jan.lehr, christian.bischof}@sc.tu-darmstadt.de
{calotoiu, wolf}@cs.tu-darmstadt.de

***Abstract*—The analysis of runtime performance is important during the development and throughout the life cycle of HPC applications. One important objective in performance analysis is to identify regions in the code that show significant runtime increase with larger problem sizes or more processes. One approach to identify such regions is to use empirical performance modeling, i.e., building performance models based on measurements. While the modeling itself has already been streamlined and automated, the generation of the required measurements is time consuming and tedious. In this paper, we propose an approach to automatically adjust the instrumentation to reduce overhead and focus the measurements to relevant regions, i.e., such that show increasing runtime with larger input parameters or increasing number of MPI ranks. Our approach employs Extra-P to generate performance models, which it then uses to extrapolate runtime and, finally, decide which functions should be kept for measurement. Also, the analysis expands the instrumentation, by heuristically adding functions based on static source-code features. We evaluate our approach using benchmarks from SPEC CPU 2006, SU2, and parallel MILC. The evaluation shows that our approach can filter functions of little interest and generate profiles that contain mostly relevant regions. For example, the overhead for SU2 can be improved automatically from 200% to 11% compared to filtered Score-P measurements.**

***Index Terms*—automatic instrumentation, performance modeling, high-performance computing, performance analysis**

I. Introduction

The analysis of application performance is an important step during the development and throughout the life cycle of high-performance computing (HPC) applications. Many of the established simulation frameworks and applications used today have been in development for more than 10 years and outlived several generations of hardware. Every hardware generation posed other and new requirements on applications, so their behavior needed constant inspection and improvement.

To benefit from, e.g., increasing parallelism, two main objectives are of importance: (1) for which functions in the target does the amount of work increase with input size, and, (2) for which functions in the target does the amount of synchronization increase with added parallelism. The first one searches for regions that benefit from (additional) parallelism, commonly referred to as kernels. The second one searches for regions in which the amount of synchronization needs to be reduced to more efficiently use the parallelism available.

The identification of kernels is usually important when an application needs to expose more parallelism, e.g., porting it to another hardware architecture. The code is then searched for regions that show increasing amounts of work with increasing input size. Hence, it is of interest to understand which kernels, e.g., functions, in the application depend on the input parameters, e.g., grid size or number of iterations. Likewise, for code regions of which the runtime increases with the degree of parallelism, it is of interest which relation the runtime increase has w.r.t. the number of processes used.

One approach to identifying regions that show interesting behavior depending on one or multiple input parameters is performance modeling, i.e., constructing a mathematical model that expresses properties, such as wall-clock time, as a function in the parameters of interest. In particular, empirical performance modeling, as implemented in Extra-P [1], has gained attention, as the approach relies on measurement data rather than the time-consuming manual model construction [2].

As empirical performance modeling is based on runtime measurements, the method is prone to jitter in the measurement data arising from the profiling itself. Jitter is the consequence of perturbation in the target application's (from here simply *target*) execution caused by the measurement system – noticeable as runtime overhead. Also, an increasing number of modeling parameters requires increasingly many measurements of the target. It is, therefore, preferable to base the model construction on measurements that do not impact the target execution unreasonably heavy.

To mitigate the perturbation and resulting long experiment time, the instrumentation configuration, i.e., which functions are instrumented, needs to be adjusted to reduce the impact of the measurement system on the target. Currently, this process requires different degrees of manual configuration.

We can summarize the challenges currently arising in empirical performance modeling as follows. (1) Performing the required measurements can take a considerable amount of time, partly, due to runtime overhead. (2) The measurement

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. The work has been supported in part by the German Research Foundation (DFG) through the Program "Performance Engineering for Scientific Software" (BI 714/6-1), CRC 1194 (Project Z-INF) "Interaction between Transport and Wetting Processes", and the *ExtraPeak* project, and by the US Department of Energy under Grant No. DE-SC0015524. Calculations were performed on the Lichtenberg Hochleistungsrechner at TU Darmstadt.

influence on runtime interferes with the modeling itself, potentially, leading to misleading results. (3) The number of functions in current software can be large and leads to too many functions presented to the user at once.

In this paper, we propose an automated iterative approach for the creation of reasonable instrumentation configurations using empirical performance modeling. We extend our previous work on PIRA [3] to include Extra-P's modeling approach and implement new filter techniques that automatically focus the measurements to regions that show input-parameter dependent runtimes. In particular, the approach uses the models to extrapolate the runtime of the functions recorded in the measurements, to refine the instrumentation towards only those functions that will consume the most runtime with increasingly large input parameters.

We validate our approach with sequential codes from SPEC CPU 2006, the computational fluid dynamics code SU2 [4], and a multi-parameter study of the parallel MILC application.

Our main contributions in this paper are:

1) An approach to automatically reduce the runtime overhead of direct instrumentation for the creation of performance models with Extra-P.
2) An implementation of the proposed approach that streamlines the task of repetitive measurements for the generation of empirical performance models.

The paper is structured as follows: first, an introduction to empirical performance modeling is given in Section II and its implementation in Extra-P in Section III. We introduce PIRA in Section IV, before we elaborate on the improvements to the automation and the analysis component used to steer the filter process in Section V. The explanation is followed by the evaluation in Section VI, and the discussion in Section VII. We set our approach into perspective to related work in Section VIII. Finally, we conclude in Section IX and outline future research directions in Section X.

II. Empirical Performance Modeling

Empirical performance modeling is a performance analysis technique that owes its existence to key insights into the cost-effectiveness of performance analysis approaches. Analytical performance modeling has been proven many times to provide unparalleled understanding into the behavior of applications, being able to identify performance bottlenecks [5], explain runtime performance [6]–[8], and even predict performance [9]–[12]. The difficulty in the analytical approach lies in the complexity and time required to model even simple codes. Therefore, to be of widespread practical use, a performance modeling method must be as accessible as possible. Empirical performance modeling provides the type of insightful models previously confined to analytical modeling by instrumenting and measuring performance relevant metrics, e.g., runtime or number of floating operations, while varying certain configuration parameters such as number of processes or the problem size per process and determining how the metrics change with respect to one or more such parameters. This information can be presented to the user as human-readable functions, for example showing that the runtime of a routine **`foo`** can be expressed as a function of the number of processes: $t(p) = 100 + 10 \cdot p^2$ seconds.

III. Extra-P

Extra-P [1] is a state of the art tool that can leverage fine-grained measurements, e.g., provided by Score-P, to generate performance models for multiple parameters [13]. The core concept relies on the fact that the complexity of algorithms implemented in both sequential and parallel applications with respect to most relevant configuration parameters is most commonly polynomial, logarithmic, or some combination of the two. This has led to the introduction of the performance model normal form (PMNF), which expresses the effect of a number of parameters x_i on a metric as a sum of terms consisting of products of polynomial and logarithmic expressions in the parameters x_i. The expression is formalized in Equation 1.

$$f(x_1, \ldots, x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \qquad (1)$$

Given a set of measurements, the performance models are identified in an iterative process. We first model the effects of each separate parameter, and then test all possible combinations of the selected single-parameter models to determine the multi-parameter model that fits the measurements best.

An important assumption of the modeling approach is that there is one behavior to the modeled application across the entire parameter range. Should this not be true, for example due to the MPI collective communication algorithm changing with increasing number of processes, then the resulting model may be misleading. A method has been developed that can automatically detect such an occurrence and if necessary suggest additional measurements [14].

Extra-P has been successfully used to detect scalability bottlenecks and evaluate the performance of many libraries and scientific applications [1], [15], [16].

While the modeling process itself has been streamlined and is both efficient and accurate, the approach relies on the measurements themselves to provide useful results. Gathering high quality measurements with Score-P or other instrumentation tools is difficult and time consuming, and noise can have a negative impact on the quality of the resulting models. An approach which would remove unimportant code regions from the analysis altogether will not only reduce the measurement overhead but also improve the quality of resulting models, and therefore allow the developers to find and focus on the true issues in their applications quicker.

IV. PIRA

Many performance analysis tools [1], [17], [18] rely on the Score-P [19] measurement infrastructure to obtain the required performance profiles. Since Score-P by default uses automatic compiler instrumentation to obtain measurement data, it may perturb the target's runtime quite significantly, when no filtering is applied [20].

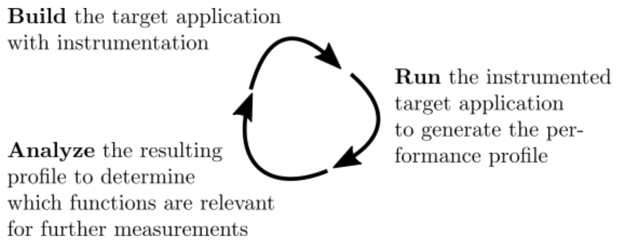
Build the target application with instrumentation
Run the instrumented target application to generate the performance profile
Analyze the resulting profile to determine which functions are relevant for further measurements

Fig. 1. The build–run–analyze cycle: build the application in a certain configuration, running it to generate measurement data, analyze the resulting profile to generate some insight.

Particularly challenging for instrumentation-based measurements are codes that consist of many small functions, e.g., **`operator[]`** in C++. Since the amount of work such functions incorporate is comparably small, the negative effects of instrumentation are very prominent. The runtime overhead introduced by unfiltered direct instrumentation due to such small functions can become larger than $100x$ easily [21].

To reduce the effect, Score-P implements a compiler plugin for GCC that automatically filters functions marked **`inline`**. In addition, analysts reduce overheads by manually creating filter lists and employ them at runtime or at compile time, to filter out the specified functions. For some applications, creating these filter lists is relatively straight forward. First, the analyst performs a measurement with a full instrumentation. Then, the list of functions, sorted according to their runtimes and call counts is inspected, and, the functions that show the highest call counts are manually added to the filter list. This is repeated until a satisfying overhead reduction is achieved.

To reduce the time spent by an analyst creating these filter lists, we developed PIRA [3], which fully automatically constructs such lists. It automates the build–run–analyze cycle, shown in Fig. 1, as an iterative instrumentation refinement. It, first, performs baseline measurements, i.e., running the target without any instrumentation, to compute the overheads generated from the instrumentation applied in every iteration. The iterative refinement process starts with an initial, statically determined, guess based on the approach presented in [22]. We improved the strategy to adjust the threshold value used for filtering based on statistical measures of the static code features, i.e., the number of statements per function. After the initial iteration, PIRA refines the instrumentation using heuristics that consider both static source-code features and runtime measurements gathered in the previous iteration.

The initial PIRA version (PIRA I) carried out all measurements for one input configuration. It refined the instrumentation towards runtime hot-spots in the target by analyzing raw runtime in the profiles. This process is of course sensitive to input data and the result is specific to that data set.

In this work, we replace the prior heuristics with a new scheme that filters out functions based on empirically determined and automatically built performance models.

V. PIRA II

Empirical performance modeling with Extra-P requires multiple data points as input. Therefore, we add the possibility to run the target with multiple input configurations to PIRA. Also, we introduce the notion of *repetitions*, i.e., running the target with the same instrumentation and input configuration multiple times. This is required by Extra-P, to account for noise in the measurement data and to support the curve fitting of the PMNF.

As another result of the integration of Extra-P, we improve the storage organization of the performance profiles generated during the PIRA process. All profiles are stored with a naming scheme conforming to Extra-P's requirements, alleviating the need for additional scripts. This allows a user to manually use Extra-P with the profiles generated by PIRA in each iteration after the overall PIRA process finished.

In subsequent sections, we present more detail about the design and the components required to enable the automation. We outline how PIRA implements its instrumentation refinement, and, also, we explain how PIRA extrapolates and evaluates the obtained performance models to decide which functions to prune from subsequent measurements. Lastly, we explain how PIRA statically expands the instrumentation. Please note, in the remainder of this paper, PIRA refers to PIRA II unless stated otherwise.

A. Automation

As in PIRA I, baseline measurements are performed before an initial, statically determined instrumentation is generated.

In each iteration, the target is built using the instrumentation generated in the previous iteration. The necessary commands are generated by the framework and depend on the measurement tool-chain backend. In our case, it is set to Score-P, as Extra-P consumes profiles generated in that format.

As large application builds can take a considerable amount of time, PIRA II allows to select between the two modes (1) compile-time filtering, and, (2) runtime filtering. The first increases the time required per iteration, because it needs to rebuild the target at the beginning of each iteration. However, it only introduces the overhead generated by the measurement hooks actually requested. On the other hand, with runtime filtering the target is built once, with all functions instrumented. The functions are then filtered at runtime by employing Score-P's filter-file mechanism. The full instrumentation can significantly slow down the target. Thus, in this work we only present numbers for compile-time filtering.

After the target has been built, the correct invocation of the target is required. The arguments are provided in the configuration file along with the other necessary information. The extended format includes the definition of parameter series for both command line options and input files.

After all repetitions for the different combinations of input parameters have run, PIRA invokes the analysis engine, in which the new instrumentation is generated. The profiles are given to the analysis engine and it generates a list of functions to be instrumented in the next iteration.

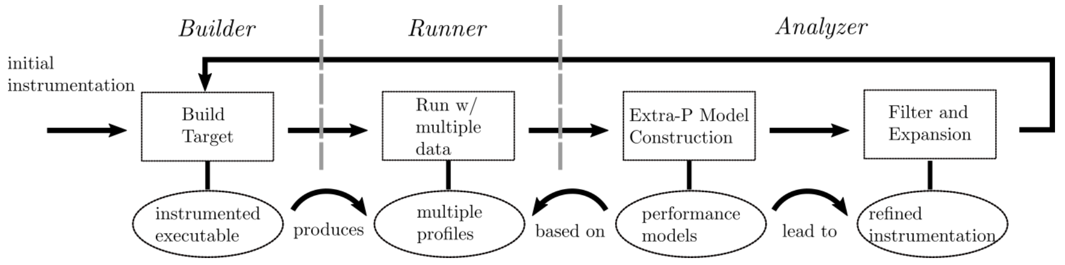

Fig. 2. PIRA process: Initial instrumentation is determined by source-feature heuristics, then carry out the build–run–analyze cycle. The rectangular boxes show the actions happening, the oval shaped boxes outline the generated artifacts, e.g., all necessary profiles to use Extra-P for empirical performance modeling.

In its current version, PIRA iterates for a fixed number of iterations before stopping. While this is not optimal and the stopping criterion should be formulated in terms of, for example, code coverage or measurement perturbation, it already shows the benefits of an automated iterative refinement approach to generate filtered performance profiles.

B. Instrumentation Refinement

This work extends the analysis engine to support Extra-P's automatic empirical performance modeling. PIRA implements two different refinement strategies: first, a filter-only mode, and, second, the filter-and-expand mode, which uses the application's call graph to add functions to the instrumentation.

To generate the initial instrumentation, PIRA uses the fully static strategy as used before. In subsequent iterations, i.e., profile data exists, the analysis engine receives as its input all profiles generated for the target in the previous iteration. It constructs the performance models using Extra-P, and attaches the models to the function nodes of the statically built call graph. The call graph is used to (1) find and mark all possible paths from `main` to a function selected for instrumentation, and, (2) add new functions to the instrumentation, when PIRA is set to filter-and-expand mode.

1) Call Graph: To apply the heuristics, PIRA requires a whole-program call graph, which is built in a pre-processing step using a Clang-based tool. The nodes in the call graph are annotated with the number of statements contained within each function. A statement corresponds to one C/C++ construct that ends with a semicolon (*cf.* Listing 1).

Listing 1
PIRA'S NOTION OF STATEMENTS: SHOWING FOUR STATEMENTS.

```
int compute(int a) {
  int b = 2 * a; // one statement
  for (int i = 0; i < a; ++i) {
    b += i; // one statement
  } // one loop statement
  return b; // one statement
}
```

The call graph is over-approximated w.r.t. edges for virtual functions and pointers to function, i.e., in both cases all potential call targets are considered. Hence, if any of the two cases is present in the target, the call graph has more edges than can occur at runtime. Since, we use (1) runtime information as the primary source of information to filter, and, (2) use static code information only in aggregate form as explained in Section V-B4, we do not consider this as a major problem.

2) Performance Models and Extrapolation: After each instrumented iteration, the performance models are generated by Extra-P and attached to the corresponding function nodes in the call graph. To use the performance models generated, the respective model functions need to be evaluated at a point of interest, i.e., values for the function's parameters.

PIRA uses extrapolation whenever the analysis engine needs to decide whether a function is relevant. Currently, it bases the extrapolation on the data points provided as the original input to the target. It calculates the average difference d between two adjacent input parameter values in the sorted sequence of given input parameters as $d = \frac{1}{n}\sum_{i=1}^{n} ||p_i - p_{i-1}||$. The value obtained for d is added to the last user-provided value to result in the extrapolated value $p_{ext} = p_n + d$. This value is then used to evaluate the performance model and decide if the function is relevant, i.e., if it is above a specified threshold.

3) Filter Only: In filter-only mode, PIRA visits all nodes in the call graph and marks those for instrumentation for which the evaluated model is above a specified threshold. After a node is marked for instrumentation, PIRA finds all paths from this node to the `main` function and adds them to the instrumentation. In the example shown in Fig. 3, the dotted circles indicate that a performance model was constructed. It is then evaluated and tested if it qualifies to be kept for the next iteration. Assuming that the nodes C and F are selected for instrumentation, PIRA reconstructs the paths from both nodes to the root node A, and marks them for instrumentation, as indicated with the gray color. Hence, for all relevant functions, also their calling contexts are preserved, as they may behave differently in different contexts. This is important for an analyst when using the generated measurements to determine tuning potential in the target.

4) Filter and Expand: In the filter-and-expand mode, shown in Fig. 4, PIRA first applies the model-based filtering, before it performs an expansion step using the already explained static statement aggregation heuristics using a locally determined threshold. It estimates the amount of work of a function based on the number of statements within the respective function itself and all its children in the statically collected call graph. In the aggregation step, every node in the respective sub tree is visited once, thus, functions are not counted multiple times

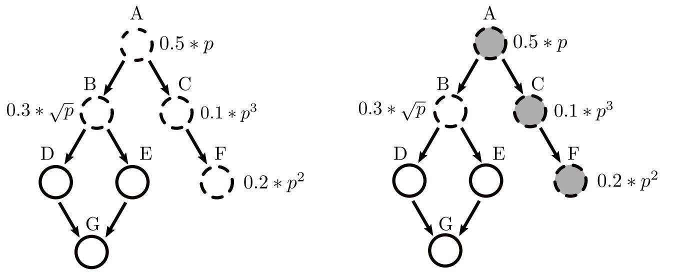

Fig. 3. Filter-only mode: The available performance models are attached to the corresponding call-graph node. The model is evaluated at an automatically determined point p_{ext} to determine if a node is kept for subsequent measurement based on whether its value satisfies a set threshold criterion.

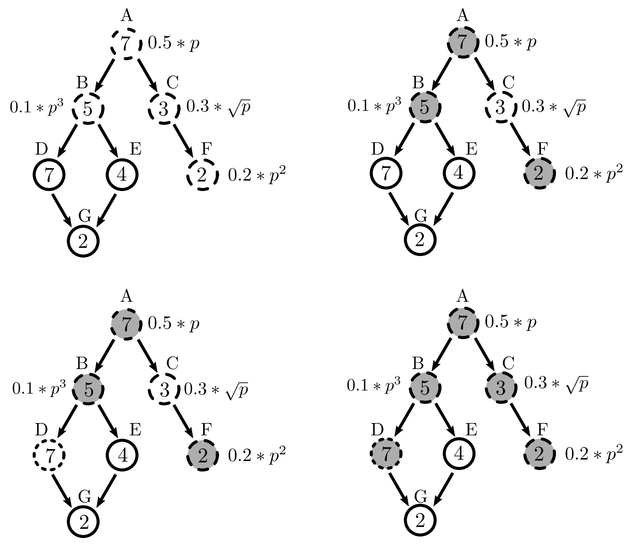

Fig. 4. Filter-and-expand mode: First, the filter strategy is applied, marking nodes A, B, and F. Thereafter, starting from those nodes, a static source-code criterion is evaluated on all its child nodes. For node D, the number of statements contained in the function satisfies the selection criterion. Hence, D is added to the instrumentation. Node C is added to the instrumentation as it is on a call path from F to the root node.

in the presence of cycles. For all selected nodes, all paths to **`main`** are added to the instrumentation.

5) Instrumentation: The resulting instrumentation is passed to a modified version of the Clang compiler, which emits the instrumentation while lowering to LLVM IR. Generally, our approach is independent of the compiler, as long as the compiler supports selective instrumentation at the function level. Unfortunately, no major compiler, currently, supports this feature off the shelve, although the new Score-P version offers function-level filtering as a GCC plugin.

VI. Evaluation

In our evaluation, we consider two scenarios: sequential, single-parameter studies (SPEC CPU 2006 433.milc, 473.astar, and SU2), and parallel, multi-parameter study (su3_rmd).

In Section I, we outlined three particular challenges that we address in this work. Limiting the number of functions instrumented addresses all three challenges. Consequently, we are interested in whether PIRA is able to significantly reduce the overhead and guide the instrumentation towards relevant regions, i.e., such that depend on the input parameters. Therefore, our evaluation focuses on (1) the runtime overhead posed on the target application by the instrumentation, (2) the number of instrumented functions in the final PIRA configuration, and, (3) if PIRA keeps relevant functions in the instrumentation. We use PIRA in the filter-and-expand mode, and compare our measurements to measurements performed with Score-P with and without compile-time filtering of all **`inlined`** functions.

All measurements are carried out on nodes of the Lichtenberg cluster at TU Darmstadt. Each node consists of two Intel Xeon E2670 and 32 GB of main memory. The sequential codes are run with a fixed clock frequency of 2.6 GHz, thread pinning and disabled HyperThreading. The MPI benchmark is run without fixed frequencies, due to a technical limitation when invoking MPI job from within Python scripts.

We report the median over five repetitions as the runtime and, first, present the evaluation results for the sequential and single parameter applications, before we show the results we obtained for the MPI parallel multi parameter study. Table I lists the accumulated time required to perform all necessary executions of the target application if no instrumentation is applied, i.e., the *Vanilla* runtime, in column one. If a user instruments the code base using a default Score-P instrumentation without filtering, the executions of the measurements required for the Extra-P modeling add up to the value given in the column *Score-P*, and the respective overhead. The next column shows the runtime and the overhead generated when using the standard Score-P with the compile-time filtering, i.e., all **`inline`** marked functions are filtered. Column seven lists the total time required to run the iterative refinement process with PIRA (*PIRA Total*). This time includes the times spent in the different stages, and results in an instrumentation that is used for further measurements. The last column denotes the time required when the final PIRA instrumentation is used to perform the Extra-P measurements.

A. Serial Application / Single Parameter

In the case of SU2, we vary the number of iterations in the solver and use 100, 500, $1k$, $2k$, and $5k$ as its inputs. For the SPEC CPU 2006 codes, we use data sets from the test and train size. For *433.milc*, we vary the number of flavors to compute from 1 to 5 and for *473.astar*, we vary the number of paths in the map with the values $5k$, $10k$, $25k$, $50k$, $75k$.

The final instrumentation results in a significantly faster execution of all our benchmarks, when compared to a full Score-P instrumentation. Moreover, the full PIRA process is faster than a full unfiltered Score-P measurement and sometimes even en par with the filtered measurement. The final PIRA instrumentation generates less overhead than both Score-P versions. In particular, the PIRA process needs to be run once to generate the instrumentation and can be, in addition, much faster than a full Score-P measurement, e.g., PIRA requires 18,132 seconds to generate an instrumentation for SU2, whereas a full unfiltered Score-P measurement needs

TABLE I
ACCUMULATED TIME (SECONDS) TO EXECUTE MEASUREMENTS FOR EXTRA-P: NO INSTRUMENTATION (*Vanilla*), FULL INSTRUMENTATION (*Score-P*), PIRA I INSTRUMENTATION (*PIRA I Final*); PIRA II WITH ALL ITERATIONS (*PIRA II Total*), AND THE FINAL PIRA II CONFIGURATION (*PIRA II Final*).

Application	*Vanilla*	*Score-P w/o Filter*	*Overhead Score-P*	*Score-P w/ Filter*	*Overhead Score-P*	*PIRA I Final*	*Overhead PIRA I*	*PIRA II Total*	*Overhead PIRA II*	*PIRA II Final*	*Overhead PIRA II*
SU2	2,835	202,675	7,049%	8,865	212%	3,092	9.1%	18,132	539%	3,144	10.9%
433.milc	10,041	32,304	221%	31,010	208%	11,854	18%	17,905	78%	10,048	0.0%
473.astar	1,034	18,190	1,659%	4,941	377%	1,602	54.9%	4,741	356%	1,219	17.8%
MILC	66	2,611	3,856%	333	405%	347	425%	3,180	4,718%	272	312%

TABLE II
NUMBER OF FUNCTIONS: STATICALLY IN THE CODE BASE, CAPTURED IN PROFILE WHEN USING SCORE-P, CONTAINED IN THE FINAL PIRA CONFIGURATION, AND EXECUTED IN THE FINAL PIRA CONFIGURATION.

Application	Code Base	Score-P w/o Filter	Score-P w/ Filter	PIRA II Marked	PIRA II Executed
SU2	15,775	8,107	281	95	37
433.milc	353	236	113	22	21
473.astar	326	128	95	11	11
MILC	505	64	47	88	30

TABLE III
THE VALUES USED AS INPUT FOR MILC IN THE MULTI-PARAMETER STUDY. ALL COMBINATIONS OF THE PARAMETERS ARE REQUIRED.

Num. MPI Procs.	Grid Size
8	16 x 16 x 32 x 16
16	32 x 32 x 32 x 16
32	64 x 64 x 32 x 16
64	128 x 128 x 32 x 16
128	256 x 256 x 32 x 16

202,675 seconds. The speed up is roughly $10x$ for SU2, $2.2x$ for 433.milc and almost $14x$ for 473.astar. Although Score-P measurements can usually be sped up considerably with compile-time filtering, 433.milc did not profit from it considerably. Overall, PIRA creates instrumentations that significantly reduce the runtime overhead of direct instrumentation.

We ran the same measurements given the instrumentation generated by PIRA I. The overhead generated by PIRA I is larger if the identified kernels are deeper in a high-frequency call chain. In the other cases, the higher runtime for PIRA II is a result of it instrumenting all paths to a target function, whereas PIRA I instruments only one call path.

In Table II, we compare three different types of occurrences of functions in the measurement data, i.e., (1) the number of functions in the code base, (2) the number of functions marked for instrumentation by PIRA, (3) the number of functions executed in the final PIRA run, and, (4) the number of functions executed in a fully instrumented target. We can see that, particularly for larger applications, PIRA significantly reduces both the number of functions marked for instrumentation and the number of functions actually executed. In the case of SU2, for example, the final PIRA instrumentation results in 95 different functions being instrumented from which only 37 are being executed, compared to 8,107 being executed when using Score-P. Many of the functions recorded by Score-P in the unfiltered version are functions that do very little work, e.g., return the file name of the input file. These functions do not add significant insight to the understanding of the performance behavior of the target, and are mostly marked inline. Consequently, also the Score-P compile-time filter reduces the number of functions recorded to only 281.

As no quality measure for the generated performance models exists, we manually inspected that mostly relevant functions from the target are preserved in the final profile. We found that for some of the benchmarks, PIRA keeps functions that are constant, but still evaluate to values larger than the specified threshold. Also, some functions that do not add significant amounts of runtime are instrumented, because they are on a call path to a relevant function.

B. Parallel Application / Multi Parameter

Our second scenario is a multi-parameter study in which the combination of varied input parameters and varied number of MPI processes is investigated. We use fairly small data sets in our experiments to account for the memory footprint with smaller numbers of MPI processes, i.e., the largest grid size and eight MPI processes consumes ~4.5 GB per process.

In the multi parameter case, all combinations of the input parameters need to be run in order to allow Extra-P to generate the performance models (*cf.* Section II). Table III lists the values used to generate the required 25 measurements.

The runtimes obtained are listed in the last row of Table I and show that the average impact of the final PIRA instrumentation is considerably large (312%). However, compared to the influence of the Score-P measurement overhead of 3,856%, it is about a factor of $10x$ smaller.

In the multi parameter setting, the total time PIRA takes to construct the final configuration is larger than the time necessary to conduct all measurements with Score-P. However, the influence of the measurement system in the selectively instrumented binary generated by PIRA should be lower than for the fully instrumented one, decreasing its overall influence on the modeling process. Also, the number of functions presented to the user is reduced by PIRA. although the number of functions in the full profile is not overwhelmingly large.

VII. Discussion

In our experiments we found that the approach worked well in most cases. An important goal it achieved is to limit the number of functions for performance modeling and final presentation to the analyst. The benchmarks taken from SPEC CPU 2006 are comparably small in code size and number of functions. Thus, the filter-and-expand strategy does add nearly no functions in the expand phase, and, the overhead of the initial instrumentation cannot be reduced much further without a final filter step. However, the SU2 case shows that also for larger programs, the approach is able to greatly reduce the number of functions instrumented when compared to both the unfiltered and the compile-time filtered Score-P versions.

The filtering reliably reduced the number of functions marked for instrumentation in subsequent measurements. This significantly reduces per-run runtime overhead. While the total experiment time was not reduced in all cases, compared to the filtered Score-P version, the final configuration showed significantly reduced overhead for the sequential applications.

The extrapolation for the filtering worked reasonably well for the sequential targets. However, in the multi-parameter case, the simple linear interpolation seems to be too limited to extrapolate the number of processes adequately. This can be improved by making the extrapolation configurable or by adding support for user-defined extrapolation functions.

The approach is still sensitive to the inputs: the extrapolation-based filter uses a threshold to determine which functions to instrument, i.e., in the current implementation large constant models can end up in the profile as well. However, as the modeling is based on runtime measurements, it will always be sensitive to input data.

Since we commonly started with many functions in the initial instrumentation configuration, the expansion step did not contribute as much to the refinement process. This is particularly interesting, as the lacking loop information for instrumentation expansion is not as severe in this case.

VIII. Related Work

In [23] the authors define filters to configure which parts of a target should be instrumented. Opposed to our approach, these filters have to be constructed manually by taking into account, e.g., loop nesting and cyclomatic complexity, to define which parts of the target to instrument. The target is then instrumented at the binary level by employing binary rewriting. Compared to our approach the selection is fully based on static features, whereas our approach mainly relies on runtime information. However, the additional consideration of loops in the static expansion could greatly benefit the expansion and lead to better initial guesses in the instrumentation.

The work presented in [20] also applies call-graph analysis prior to instrumentation. However, the authors rely on direct instrumentation only for MPI calls and employ call-stack unwinding to restore the call context for every instrumented call to an MPI function. As a result, the communication behavior can be restored with full information, while the remaining execution is sampled. Compared to our approach, the goal is to capture communication behavior, whereas, our approach tries to find an instrumentation configuration that enables Extra-P to produce performance models for the functions that are most likely relevant when increasing the data sets. Using sampling to reconstruct the call paths to the main function is interesting, and might be considered further in future work.

The approaches presented in [24]–[26] consider the dynamic adjustment of the instrumentation at runtime. Our approach on the other hand requires post-mortem analysis of several measurements for the performance model construction, thus, it cannot be adjusted on the fly.

In addition, many more frameworks and tools to implement instrumentation filters and analyses exist. TAU [27] is used for many projects and has shown its capabilities to implement program analyses based on its Program Database Toolkit (PDT) [28]. PathWAY [29] is another framework that enables developers and analysts to define workflows and implement program analyses together with instrumentation transformations. COMPASS [30] is a framework for automatic performance modeling and prediction. Compared to our approach, the framework relies on source code scanning, whereas our approach employs performance measurement to derive the performance models.

IX. Conclusion

We extended the tool PIRA to automatically construct instrumentation filters, using empirically determined performance models, to reduce jitter in performance measurements used for empirical performance modeling. In most of our test cases, the resulting instrumentation filters significantly reduced the runtime overhead for the experiments required.

We showed that the iterative filtering and refinement can reliably reduce the number of functions marked for instrumentation, when compared to a Score-P instrumentation. Applying performance modeling during the iterative refinement of the instrumentation configuration helps to identify relevant regions w.r.t. the particular parameters varied in the modeling. In this work, we varied the application's input parameters to find those functions of the target that consume increasing amounts of runtime with larger input data. PIRA filters the target application's functions based on metric values obtained from evaluating the performance model at extrapolated points based on the user-provided input parameters.

Extra-P requires that all processes generate the same call paths to successfully construct a performance model. Thus, the analyst needs to be aware of the interplay of input parameters and MPI processes when setting up the experiments.

PIRA is somewhat sensitive to input data in the sense that the modeling is based on runtime measurements. Hence, the input data should be chosen carefully enough and the results still should be validated by the analyst after PIRA has finished.

In conclusion, we showed that PIRA effectively reduces the number of functions instrumented, thus, reducing the runtime influence of the measurement system and, consequently, can be helpful to obtain at least a reasonable starting point for adequate measurements for empirical performance modeling. Obtained results should, however, be inspected for validity.

X. Future Work

Currently, the static analysis takes not into account loops, although knowledge of their presence can improve the static selection step. Moreover, identifying loops without a constant or pre-determined loop bound is of interest, and can be integrated into the call-graph collector.

Allowing a more flexible notion of region to instrument is of interest. This would enable the generation of performance models for aspects of the code not tied to a single function.

Since PIRA already records the overhead generated by the instrumentation in much detail, the modeling capabilities of Extra-P can be used to identify a particular correspondence between input parameters and instrumentation overhead. This would enable the analyst to very specifically control how much overhead a measurement would introduce.

References

[1] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 45:1–45:12.

[2] G. Bauer, S. Gottlieb, and T. Hoefler, "Performance modeling and comparative analysis of the milc lattice qcd application su3_rmd," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 652–659.

[3] J.-P. Lehr, A. Hück, and C. Bischof, "Pira: Performance instrumentation refinement automation," in *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, ser. AI-SEPS 2018. New York, NY, USA: ACM, 2018, pp. 1–10.

[4] T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso, "Su2: An open-source suite for multiphysics simulation and design," *Aiaa Journal*, vol. 54, no. 3, pp. 828–846, 2015.

[5] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive performance and scalability modeling of a large-scale application," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC'01)*. ACM, 2001, p. 37.

[6] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proc. of the ACM/IEEE Conference on Supercomputing*, ser. (SC '03). ACM, 2003, p. 55.

[7] M. M. Mathis, N. M. Amato, and M. L. Adams, "A general performance model for parallel sweeps on orthogonal grids for particle transport calculations," College Station, TX, USA, Tech. Rep., 2000.

[8] G. Bauer, S. Gottlieb, and T. Hoefler, "Performance modeling and comparative analysis of the MILC lattice QCD application su3_rmd," in *Proc. of CCGrid*, May 2012.

[9] L. Carrington, A. Snavely, X. Gao, and N. Wolter, "A performance prediction framework for scientific applications," in *Proc. of the 2003 Intl. Conference on Computational science: PartIII*, ser. ICCS '03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 926–935. [Online]. Available: http://dl.acm.org/citation.cfm?id=1762418.1762520

[10] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 284–294.

[11] X. Wu and F. Müller, "ScalaExtrap: Trace-based communication extrapolation for SPMD programs," *ACM Transactions on Programming Languages and Systems*, vol. 34, no. 1, April 2012.

[12] K. Davis, K. J. Barker, and D. J. Kerbyson, "Performance prediction via modeling: a case study of the ORNL Cray XT4 upgrade," *Parallel Processing Letters*, vol. 19, no. 4, pp. 619–639, 2009.

[13] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf, "Fast multi-parameter performance modeling," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2016, pp. 172–181.

[14] K. Ilyas, A. Calotoiu, and F. Wolf, "Off-road performance modeling – how to deal with segmented data," in *Proc. of the 23rd Euro-Par Conference, Santiago de Compostela, Spain*, ser. Lecture Notes in Computer Science. Springer, Aug. 2017, pp. 1–12.

[15] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf, "How many threads will be too many? on the scalability of openmp implementations," in *European Conference on Parallel Processing*. Springer, 2015, pp. 451–463.

[16] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf, "Exascaling your library: Will your implementation meet your expectations?" in *Proc. of the International Conference on Supercomputing (ICS), Newport Beach, CA, USA*. ACM, Jun. 2015, pp. 165–175.

[17] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[18] S. Benedict, V. Petkov, and M. Gerndt, "Periscope: An online-based distributed performance analysis tool," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–16.

[19] D. an Mey, S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. Shende, M. Wagner, B. Wesarg, and F. Wolf, "Score-p: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010*. Springer Science + Business Media, 2011, pp. 85–97.

[20] Z. Szebenyi, T. Gamblin, M. Schulz, B. R. de Supinski, F. Wolf, and B. J. Wylie, "Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs," in *2011 IEEE Intl. Parallel & Distributed Processing Symposium*. Institute of Electrical & Electronics Engineers (IEEE), 2011.

[21] C. Iwainsky, J.-P. Lehr, and C. Bischof, "Compiler supported sampling through minimalistic instrumentation," in *2014 43rd Intl. Conf. on Parallel Processing Workshops*. Institute of Electrical & Electronics Engineers (IEEE), 2014.

[22] C. Iwainsky and C. Bischof, "Call tree controlled instrumentation for low-overhead survey measurements," in *2016 IEEE Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW 2016)*, 2016.

[23] J. Mußler, D. Lorenz, and F. Wolf, "Reducing the overhead of direct application instrumentation using prior static analysis," in *Euro-Par 2011 Parallel Processing*. Springer, 2011.

[24] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, Nov 1995.

[25] J. K. Hollingsworth and B. P. Miller, "Dynamic control of performance monitoring on large scale parallel systems," in *Proceedings of the 7th international conference on Supercomputing*. ACM, 1993, pp. 185–194.

[26] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque, "Mate: Monitoring, analysis and tuning environment for parallel/distributed applications," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 11, pp. 1517–1531.

[27] S. S. Shende, "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[28] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen, "A tool framework for static and dynamic analysis of object-oriented software with templates," in *Proc. ACM/IEEE 2000 Conf. Supercomputing*, Nov. 2000, p. 49.

[29] V. Petkov, M. Gerndt, and M. Firbach, "PAThWay: Performance analysis and tuning using workflows," in *2013 IEEE 10th Intl. Conf. on High Performance Computing and Communications & 2013 IEEE Intl. Conf. on Embedded and Ubiquitous Computing*. Institute of Electrical & Electronics Engineers (IEEE), 2013.

[30] S. Lee, J. S. Meredith, and J. S. Vetter, "Compass: A framework for automated performance modeling and prediction," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 405–414.