Exploring the Use of Learning Algorithms for Efficient Performance Profiling

Shoumik Palkar*, **Sahaana Suri***, **Peter Bailis, Matei Zaharia**
Stanford DAWN Project
`{shoumik,sahaana,pbailis,matei}@cs.stanford.edu`

Abstract

A key challenge in profiling programs for identifying performance bottlenecks is balancing accuracy and runtime overhead. While statistical profilers are canonically used in production settings due to their low overhead, they treat all events equally by sampling at a uniform frequency. Thus, infrequent events are rarely sampled if the sampling rate is low (low accuracy), and low-variance events that take little time may be sampled too often if the rate is too high (high overhead). The challenge is that event frequency and runtime is unknown a priori. Tracing profilers circumvent this problem by instrumenting code and deterministically recording each event, but this model has traditionally come at the cost of high performance penalties. We explore the use of *learning profilers* that systematically trace a program and learn which parts of a program to profile. Our results show that our technique reduces the overhead of profiling a production HTML parsing workload by $1.5\times$ compared to Python's built-in tracing `cProfile` module, and provides significantly more accurate results based on the KL-divergence of the estimated runtime distributions compared to a popular open-source statistical profiler, for the same overhead.

1 Introduction

Profilers enable developers to identify and repair performance bottlenecks in production systems by providing a runtime breakdown in an executing program or service, generally at the granularity of *function calls* [7]. To reduce the overhead of profiling, modern profilers are often statistical [13, 15, 16, 17, 19, 21, 22]. Rather than instrumenting the entire program and timing each function call (as in a tracing profiler), statistical profilers sample the program at a regular interval using operating system interrupts and probe its call stack to estimate where time is spent. This estimation allows profilers to analyze programs with little intrusion in many cases.

While statistical profilers work well in practice for some applications, they sample the target program at a regular interval even though events of interest do not occur with a uniform distribution: different parts of a function take different amounts of time. Thus, to accurately capture short or infrequent events given a fixed sampling rate, the rate must be set to a high value (which adds overhead) or the program itself must be long-running (e.g., a web server). Unfortunately, existing alternatives to statistical profilers—tracing profilers that instrument code— are usually far too expensive to run in production environments because of the immense performance penalty that they add [7, 14, 19].

In response, we investigate the design of a new profiler that accurately traces programs with low overhead by *learning which parts of a program to profile*. Our learning profiler leverages the fact that, if users only care about the relative distribution of runtimes in their system to *identify bottlenecks*, we only want to profile the longest running, highest variance parts of the program aggressively. Functions with low variance and that take relatively little time can be profiled fewer times to reduce overhead.

[*] The authors contributed equally to this work.

1st Workshop on ML for Systems at Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

The key challenge is to adaptively determine which function calls require more profiling and to quickly reject those that do not, as this is unknown a priori.

As a first step towards this opportunity for learning profilers, we explore two algorithms. First, we apply a racing algorithm where the goal is to select the best (i.e., most expensive, highest variance) function calls to profile and dynamically discard the rest. We also evaluate a multi-armed bandits formulation based on the Successive Rejects [2] algorithm, which provides similar results, albeit with higher overhead. In the terminology of the multi-armed bandit literature, we treat each profiled function call as an arm and iteratively remove arms that have low variance and relative impact on performance. Although our profiler traces the program's execution, it only imposes the overhead of measuring a function if the algorithm chooses to profile it. This reduces overhead as our algorithms cause functions that have little impact on overall performance to be profiled fewer times.

We evaluate a learning profiler prototype via a new profiler for Python, Paikana. Preliminary results are promising: when profiling a production Python HTML parser, Paikana added $1.5\times$ less overhead compared to the built-in Python `cProfile` module, which traces the full program and produces a deterministic profile. Our profiler returns accurate results as well, correctly identifying the bottleneck lines and providing a KL divergence of 0.04 when comparing the runtime distributions of Paikana and `cProfile`. We also tried benchmarking Paikana against two open-source statistical profilers for Python [17, 21], but found that they give highly inaccurate results for this workload compared to `cProfile` (KL divergence of 2.38). We thus believe that a learning-based tracing system is a promising approach and hope to further explore it in future work.

2 Related Work

There are several open-source statistical profilers used in a variety of domains [13, 15, 17, 18, 21, 22]. These profilers generally use POSIX timers to interrupt the running program and analyze the executing stack frame to determine where running time is being spent, or use hardware support to profile based on hardware events. In Python, `PyFlame` [19] uses the Linux `ptrace` utility to achieve the same goal. In contrast to sampling frames at a regular interval, our goal is to explore a new algorithmic methodology to reduce profiler overhead by requiring fewer samples for the same accuracy. These techniques could also lead to improvements in existing statistical profilers.

SWAT [9] extends the framework from burst tracing [10] to use *adaptive* program tracing to find bugs in the target program. Our algorithm has a similar design with a different goal: to *stop* profiling lines of code as quickly as possible. Several researchers have proposed using various hardware counters to further optimize profiling [1, 6, 8]. These techniques are largely orthogonal to selecting *which* lines to profile, and could be used with Paikana as well.

3 Methodology

When executing long-running programs, users wish to efficiently and accurately identify which operations are performance bottlenecks. A core challenge in developing a profiler to enable this is trading-off profiler overhead with accuracy. While a profiler would ideally focus effort on program segments with the highest runtime and variance, this information is unknown a priori—hence the need for performance profiling. To combat this challenge, we propose Paikana, a profiler that uses a racing algorithm to efficiently and accurately profile programs by tracing execution and dynamically learning which function calls to profile and discard. Paikana outputs a sorted list of the function calls in the target function and their estimated percent contributions to the total running time.

3.1 Paikana: Choosing Function Calls to Profile

To frame the problem of dynamically selecting lines to profile, we refer to the terminology in the stochastic multi-armed bandit (MAB) literature. In the stochastic MAB setting, an agent must iteratively hedge against K competing actions (arms) to maximize cumulative reward. Each action admits differing and initially unknown reward drawn from some probability distribution. *By mapping each function call in the profiled program to an arm with reward equal to its running time, identifying bottleneck functions reduces to identifying the top arms.*

Traditionally used for model selection, racing algorithms such as Hoeffding or Bernstein racing [11, 12] minimize the number of actions required to identify the top arm to a given confidence level. In Paikana, we implement a racing algorithm that iteratively identifies the function calls with the lowest runtime and removes them from future consideration when confident of their relative performance (using a default 95% confidence interval). Specifically, the function call with the lowest average runtime ($i = \arg\min \boldsymbol{\mu}$) is considered for removal. If the upper bound of this function's confidence interval is smaller than the minimum of all other functions' lower bounds ($UB[i] < \min_j LB[j]$), this function is no longer considered for future profiling. For long-running programs, Paikana reruns the full algorithm periodically at a (configurable) one second interval.

3.1.1 Alternative Algorithms

Paikana is modular, and easily allows for algorithmic changes in how to select which functions to profile. Thus, we also evaluate algorithms under the pure exploration framework for MAB [4], where the goal is to optimize the quality of the final recommended action(s), subject to a fixed budget size of actions. In traditional MAB, the objective is to maximize cumulative reward. As a result, classic algorithms such as Upper Confidence Bound or ϵ-greedy strategies [3] typically trade-off between exploration (trying new actions) and exploitation (focusing on seemingly high-reward actions). In contrast, the pure exploration framework separates the exploration and exploitation phases —i.e., at the end of profiler execution, the programmer requires only an accurate estimation of relative runtime distribution in exchange for low additional computational overhead.

As validation for this approach, we implement the Successive Rejects algorithm from [2] modified for top m-arm identification. While SR is as accurate as our racing algorithm, it requires a user-specified profiling budget that dramatically affects the resulting overhead but is difficult to tune a priori; in our evaluation, we set this value to be the number of iterations racing takes to converge. A follow-up Successive Accepts and Rejects (SAR) algorithm is tailored for multiple arm identification, but as it would add additional overhead on top of SR, we defer its evaluation for future work [5].

3.2 Implementation

Paikana is a Python loop profiler implemented as a C extension. We use Python's system `setprofile` function to trace function calls to profile. This allows our C module to run custom code before each function call and after each function `return`.

Listing 1 Example of profiling a loop. Users wrap the iterated value with the `paikanafor` function.

```
for line in paikanafor(f):
  line = line.strip()
  token = line.split(",")
```

Users enable profiling by wrapping an iterator used in a `for` loop with the `paikanafor` function (Listing 1). Before yielding a new item on each iteration of the loop, the custom `paikanafor` iterator runs an update function to run the algorithm for choosing which lines to profile during the current execution. When the loop body invokes a function, the callback will check whether to profile it: if not, the function returns immediately. If the function is to be profiled, we start a clock and register that the function is being profiled: when the function returns, the clock is stopped and the runtime is recorded. We envision that Paikana could also be run on general Python functions or in other domains where a low overhead tracing mechanism is available (e.g., in distributed systems [20]).

4 Preliminary Results

To evaluate Paikana, we profiled a production Python script that parses nested lists in an HTML object into a tree of Python objects. The HTML document represents a flame graph of Java performance traces: the script is used to convert the traces into a programmatically queryable representation. Because of the regular structure of the HTML, the script uses Python's regular expressions and string functions to extract values rather than a full HTML parser.

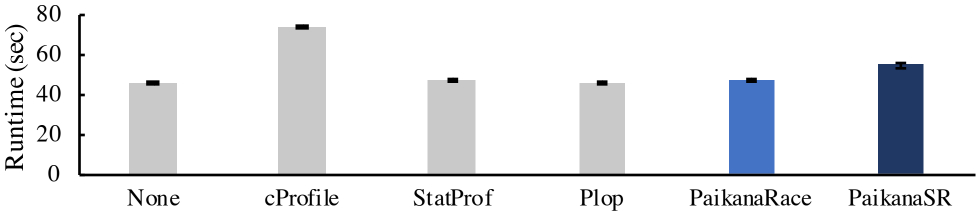

Figure 1: **Performance of HTML parsing with various profilers.** Paikana shows around 5% overhead over the baseline, which has no profiling enabled. `cProfile` profiles every call deterministically, but at the cost of high overhead (almost 50%).

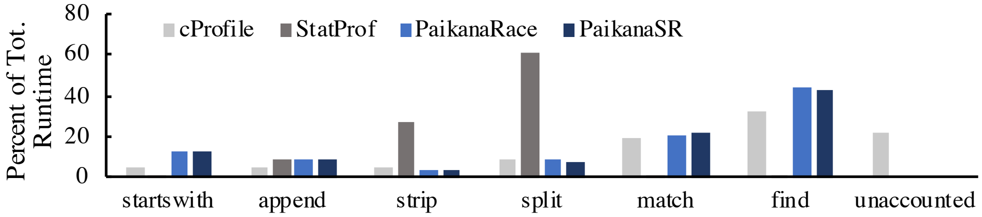

Figure 2: **Breakdown of the profile returned by `cProfile`, `statprof`, and Paikana.** Our method returns a runtime distribution with a KL divergence of 0.04 compared to `cProfile`'s.

Our benchmark ran on a 500MB HTML file. Figure 1 shows the end-to-end running time without any profiling, Paikana with the racing algorithm and the Successive Rejects (SR) algorithm, Python's built-in `cProfile` module, and two open source statistical profilers: `plop` [17] and `statprof` [21]. Both statistical profilers were run with their default sampling rate of 1000 Hz. End-to-end, Paikana reduces the running time during profiling by up to 1.5× when compared to `cProfile`, and on average adds 5% overhead compared to execution without profiling.

Although the overhead of our profiler was similar to that of the statistical profilers, we were not able to obtain accurate profiles from either `plop` or `statprof`. The former only reported the total running time of the full parsing function (as opposed to providing a profile of each subcall). The latter measured subcalls, but highly overestimated the contribution of the one of the calls. We verified this by looking at the output of `cProfile`, manually instrumenting the overestimated call by adding timing functions around it, and by removing the call and observing the delta in running time. We suspect that the overestimation occurs because many of the other expensive calls are invoked in a loop, and `statprof` seems to undercount the running time of each function within a loop [21]. We did not observe any improvements or difference in results after increasing the sampling rate of the profilers to 10000 Hz, but observed a 30% slowdown in execution time.

Figure 2 shows the accuracy of our method by comparing the reported runtime contribution of each function for `cProfile`, `statprof`, and Paikana. Despite profiling for only a fraction of the program, Paikana produces results that correctly identify bottleneck lines. The runtime distribution is accurate to a KL divergence of 0.04 compared to `cProfile` using racing, and 0.05 for SR. In contrast, `statprof` misidentifies even the top bottleneck line, and provides a KL divergence of 2.38. Overall, in our benchmarks, Paikana estimates bottlenecks in the program more accurately than the statistical profilers we used, but with the same overhead. Paikana also improves performance over the de facto default Python tracing profiler and returns similar runtime estimations.

5 Conclusion and Future Directions

We believe that machine learning based approaches are a promising avenue for optimizing online program analysis. For example, in addition to profiling, these methods could also have applications in efficient program debugging and testing. Our prototype technique is only one possible design for selecting "interesting" regions of code in an executing program: other data-driven approaches (e.g., deep learning) would also be interesting to explore with access to large codebases and production system deployments.

Acknowledgments

This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Intel, Microsoft, NEC, Teradata, SAP, and VMware—as well as DARPA under No. FA8750-17-2-0095 (D3M), NSF CAREER grant CNS-1651570, NSF Graduate Research Fellowship DGE-1656518, and industrial gifts and support from Toyota Research Institute, Keysight Technologies, Northrop Grumman, and the Okawa Research Grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[2] J.-Y. Audibert and S. Bubeck. Best arm identification in multi-armed bandits. In *COLT-23th Conference on Learning Theory-2010*, pages 13–p, 2010.

[3] S. Bubeck, N. Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[4] S. Bubeck, R. Munos, and G. Stoltz. Pure exploration in finitely-armed and continuous-armed bandits. 2011.

[5] S. Bubeck, T. Wang, and N. Viswanathan. Multiple identifications in multi-armed bandits. In *International Conference on Machine Learning*, pages 258–265, 2013.

[6] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 79–90. IEEE Computer Society, 2003.

[7] The Python Profilers. `https://docs.python.org/2/library/profile.html`.

[8] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302. IEEE Computer Society, 1997.

[9] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Acm Sigplan Notices*, volume 39, pages 156–164. ACM, 2004.

[10] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, pages 117–126, 2001.

[11] P.-L. Loh and S. Nowozin. Faster hoeffding racing: Bernstein races via jackknife estimates. In *International Conference on Algorithmic Learning Theory*, pages 203–217. Springer, 2013.

[12] O. Maron and A. W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Advances in neural information processing systems*, pages 59–66, 1994.

[13] Microsoft Common Language Runtime. `https://docs.microsoft.com/en-us/dotnet/standard/clr`.

[14] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 198–208. IEEE Computer Society, 2007.

[15] Oracle Performance Analyzer. `https://www.oracle.com/technetwork/server-storage/solarisstudio/features/performance-analyzer-2292312.htm://www.oracle.com/technetwork/server-storage/solarisstudio/features/performance-analyzer-2292312.html`.

[16] Perf. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[17] Plop: Low Overhead Profiling for Python. `https://blogs.dropbox.com/tech/2012/07/plop-low-overhead-profiling-for-python/`.

[18] pprofile. `https://pypi.org/project/pprofile/`.

[19] Pyflame: Uber engineering's ptracing profiler for python. https://eng.uber.com/pyflame/.

[20] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Technical report, Google, Inc, 2010.

[21] statprof. https://pypi.org/project/statprof/.

[22] Intel vTune. https://software.intel.com/en-us/vtune.