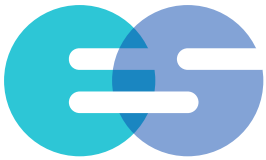
TUHH
Hamburg
University of
Technology



Debugging - CPU profiling

Seminar Report

submitted by
Lennart Brandin

Debugging – Methodology instead of Confusion
Hamburg University of Technology

May 2025

1st examiner Danylo Kozak
2nd examiner Prof. Görschwin Fey

Abstract

Performance profiling is a method of analyzing where execution time is spent. Using profiling tools, it is possible to locate performance hotspots (bottlenecks) . To achieve this, the written user code is executed and recorded as a call graph, i. e. the relationship between a function and those it is calling. Such a recording can provide information about the number of calls and time spent in each function. This report aims to describe different techniques of profiling, including their benefits, limitations and possible inaccuracies. Additionally a conceptual overview of a predictive profiling tool, as proposed in [1], is given and compared to the usage of “traditional” profiling tools.

Contents

Glosarry / Acronyms Index

***heisenbug* – Heisenberg bug**: A bug that occurs only while observing the program

***IDE* – Integrated Development Environment**: A toolset for writing, analysing, testing and executing software.

***PC* – Program Counter**: A processor register that stores the pointer to the next instruction

***bottleneck* – performance hotspot**: sections of code that, if optimized, would yield the best overall speed-up [2]

***PMU* – Performance Monitoring Unit**: A hardware implementation of monitoring functions

***Predictive Profiling* – Predicting Profiling results**: A method of predicting performance based on previous profiles

***TUHH* – Hamburg University of Technology**

***VM* – Virtual Machine**

1 Introduction

Improving the performance of a computational program is a constant goal in software development. While underlying platforms and abstractions are becoming more optimized, the computational tasks are growing increasingly complex.

Provided with the task of improving runtime, a software designer might take an educated guess or create specific tests to narrow down the hypothesis. Both lack qualitative assurance: do the tests accurately reflect the performance issue, or do they describe an entirely different one? [2] CPU profiling offers quantitative measurements of runtime associated with specific user symbols. The profiling techniques vary in degree of intrusion, detail, and accuracy of measurement.

This report will describe an overview of different profiling tools and their uses. Those presented are based on an iterative code-improvement cycle:

- Execution using a profiler
- Analyzing results for bottlenecks
- Inspecting hotspot code for optimization

The concept of predictive profiling, i.e. predicting the profiling results without compiling the source code, as part of an IDE , will be summarized. [1]

1.1 Motivation

In the process of iterative code-improvement, badly designed code may be visible early on, for example, a long-running function with little computational complexity. Manually finding bottlenecks in low-level abstractions, for example, data structures or wrapping functions, may be less obvious or simply infeasible due to the size and spread of function calls in complex programs. While it is possible to manually measure function runtime by printing time stamps, it is tedious, error-prone, and does not provide information about the summed or average execution time of all collective calls to a function. Profilers aim to ease spotting these issues by collecting statistics during execution and providing a profile used for visualizing the performance distribution over the program.

Profilers do not require any prior implementation, so they can be applied to any program. Since the profile represents a statistical measure of the program's runtime, profiles can be accumulated and compared to identify changes between runs. This could be used to spot performance regression over multiple versions of a program. When runs are performed with different inputs, the profiles could be used to predict the application's runtime growth.

Monitoring this information provides a way of identifying and solving a broad spectrum of performance issues before they occur in a critical manner. [2]

1.2 Structure of the report

This report will introduce the method of profiling, how it can be achieved using different approaches, and how the results can be further used.

Furthermore, multiple tools will be presented and compared. This list is not exhaustive nor representative of popular tools and is limited to the range of experimentation I conducted while writing this report.

As a conclusion to the summary of the State of the Art, visualizations and informational views of profiling results will be presented.

In the second part, an external paper will be briefly summarized and reviewed.

1.3 Usage of electronic tools

The following tools were used to design, research, and write this report to the described extent.

1.3.1 Typst - Typesetting

Typst is a typesetting language, used here to simplify the design process by providing an underlying system for using templates, styles, and citations. This report was designed using the Hamburg University of Technology (TUHH) typst ies-report template .

1.3.2 VSCode - IDE

Visual Studio Code is a general-purpose editor, mainly used here to write the report and provide macros and debugging features for Typst.

1.3.3 Git/Jujutsu - Version control

Jujutsu is a git-based version control system, used for tracking changes, proof of work, and backing up the report.

1.3.4 Google Scholar - Research

Google Scholar is a search engine used to find the papers and articles this report is founded upon.

1.3.5 Generative AI - LLMs

Different large language models were used in the creation process.

Different models were also used for general text conversion processes (e.g., converting between languages and formats).

- ChatGPT ChatGPT's "Deep Research" tool was used to generate a broad, sourced overview of the topic and to find articles covering the topic. It was also used to generate a quick summary of the paper [1].
- Github Copilot is a code completion tool, used as a more powerful alternative to verb suggestion, spell checking or refactoring tools.

2 State of the art

The concept of profiling, i.e., the idea of measuring code performance for finding bottlenecks , has been well established for decades. Many of the current papers discuss the usage of profiling (especially focusing on Performance Monitoring Units (PMUs)) in higher abstractions, such as Virtual Machines (VMs) or other cloud deployments. Given the growing importance of GPU computing, profiling GPU performance is also a focus.

The paper [1] proposes to integrate dynamic Predicting Profiling results (Predictive Profiling) into the code writing process, as opposed to measuring existing code and predicting runtime per input growth.

This report (and presentation) aims to convey the idea and usage of profilers in a simple context in order to provide the other students with foundational knowledge of these tools.

3 Preliminaries

This section contains definitions and background information that will be used in the report.

3.1 Profiling

In profiling, for understanding the program's performance distribution, the following statistical measurements of function calls are collected:
Call count (Total) Collective count of all calls to this function
Call duration (Total & per call) CPU time inside the call including children
Call time usage (Total and per call) CPU time used exclusively in the call
Call relationship/graph Call stack, for hierarchy visualization.

3.1.1 Instrumentation Method

Using this method, code is *instrumented* by altering existing functions. Depending on the tool, this includes some form of call counter, recording the number of calls to this function, often separated per individual caller. Timing information can also be approximately collected but is "complicated on time-sharing systems" [3]

By recording each profiled function, high-frequency functions (e.g., wrapping low-level calls) add significant overhead. Since the program is altered for measuring purposes, its behaviour changes in terms of performance or, in extreme cases, also introduces a Heisenberg bug (heisenbug) [2]

For some of this information, there exist hardware implementations such as PMUs that externalize the collection process. This is less intrusive and more performant.

3.1.2 Sampling (Statistical) Method

Sampling profilers interrupt the program at regular time intervals and inspect the Program Counter (PC) and call stack. Execution time of individual functions can be inferred by distributing the total execution time over the accumulated samples, functions that occur a multitude of times are given a high execution time approximate.

Since this method inspects only a subset of all function calls, the added overhead is negligible in comparison to other methods.

Given the statistical approach of sampling, it is less accurate in providing exact timings. To acquire a representative profile, it is important that the sampling period is chosen according to program runtime. If a program finishes in only a few samples, the execution time distribution might become inaccurate. If a program is sampled with a frequency so that the actual program is interfered with, the execution time might not be representative anymore. Additional error is also introduced by the execution time of the interrupts. [3]

3.1.3 Method Comparison

Method	Overhead	Call timing accuracy	Performance distribution accuracy
Instrumentation	High	Precise	Low - added overhead, heisenbugs
Sampling	Low	Approximate	High - statistical distribution

Table 1: Comparison of presented methods

Depending on which attributes, mentioned in Section 3.1, are of interest, the accuracy of the profiler should be considered. Using sampling profilers, the program will run near natively, and a representative profile of performance distribution is obtained. This is useful for identifying bottlenecks .

Using instrumentation profilers, the program will be significantly slowed in its execution, but one can obtain precise individual function runtimes and call counts. This is useful for quickly verifying implementations.

Table 1 summarizes core differences between the mentioned methods.

3.2 Performance prediction

This is an advanced usage of profiling. While previously the momentary performance was measured, performance prediction aims to apply accumulated profiles of different program iterations to predict future performance.

Implementing this idea allows identifying performance issues before they become critical.

3.2.1 Input based prediction

Given a program that processes inputs of different sizes, it would be beneficial to predict the performance growth in relation to the input size.

Individual measurements of different input sizes might be, if badly chosen, unnoticeably slower for the executing party. Automating these measurements allows identifying the performance growth and making an accurate prediction of the performance for larger inputs.

3.2.2 Performance regression

Code is often changed over time, either by feature expansion, bug fixes, or other changes. While usually the changes are made with improvement in mind, they might introduce performance regression that is overlooked in the current usage of the program.

These regressions often accumulate over time and are only attended to when performance has significantly degraded.

With automated profiling, releases can be directly compared to their predecessors for varying input scenarios. This allows one to directly notice and identify the cause of the performance regression. [2]

3.2.3 Predictive Profiling

Similar to Section 3.2.1, predictive profiling aims to provide performance indications without executing the program. Instead of basing on previously recorded iterations of the entire program, predictive profiling as proposed in [1] is based on learned runtimes of individual code snippets, predicting not the complete runtime but only the runtime of new snippets.

This is achieved by using machine learning on datasets that include C code snippets and their measured runtimes.

The goal is to give early hints about the performance of new code without going through the “traditional profiling” workflow.

4 Tools

In this section, a selection of profiling tools is shown and compared.

4.1 Profiling

Using the tools usually involves compiling the code with flags so that the function names can be recognized in the profile, and running the executable with the profiler or attaching the profiler to a running process. This produces a profile which can later be visualized, either by the profiler itself or an external tool.

4.1.1 gprof

GProf is a profiler using both sampling and instrumentation methods. It provides a precise call count, call relations, and approximate call duration from which call time can be inferred.

The tool was presented in the paper [3] and is currently actively maintained.

The instrumentation method is used to track the exact call count and call-site callee relationships. These are tracked using an in-memory hash table and can be used to visualize a call graph. In order not to interfere with program execution, the call timings are collected using sampling and are presented as accumulated-(total) and individual-, i.e., without its descendants (self), time. Additionally, average times per call are calculated.

4.1.1.1 Limitations and Drawbacks

The added instrumentation (`-pg` flags) still adds significant overhead (which can be observed by profiling the instrumented binary as in Section 4.1.2).

GProf is not suited for programs "that exhibit a large degree of recursion" [3] as well as multithreaded applications.

4.1.1.2 Example

The example Listing 1 is a profile of a small raycasting project, which calculates the distances of a position in a 2D grid to the nearest wall and prints a column with the corresponding height, resulting in a 3D view.

For collecting the profile, the program was compiled with `gcc` and `-pg` flags and run as `gprof ./solution gmon.out`.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 47.75      0.94     0.94      544     1.73     3.29  raycast
 25.91      1.45     0.51 51479843     0.00     0.00  get_object
 12.70      1.70     0.25 51481719     0.00     0.00  valid_coordinates
  5.08      1.80     0.10      544     0.18     0.18  render_frame
  4.06      1.88     0.08                             _init
  1.52      1.91     0.03   122400     0.00     0.00  construct_frame_column
  1.52      1.94     0.03      544     0.06     0.06  print_field
  0.51      1.95     0.01   367622     0.00     0.00  deg_to_rad
  0.51      1.96     0.01   122400     0.00     0.00  v_dist
  0.51      1.97     0.01   122400     0.00     0.00  v_length
  0.00      1.97     0.00   122400     0.00     0.00  v_sub
  0.00      1.97     0.00   110666     0.00     0.00  set_terminal
...
```

Listing 1: GProf output of a raycasting project, Flat profile sorted by self seconds - proportional to time%

Legend:
Time % of total time
Cumulative (sec) accumulated CPU time in this function and all listed above
Self (sec) accumulated call time
Calls Number of calls
self ms/call Average call time
total ms/call Average call duration (call time + children)
name Function name

The profile Listing 1 shows that high amounts of execution time are spent in `raycast`, `get_object`, and `valid_coordinates`. This (knowing the program) indicates that optimizing the `raycast` loop would speed up runtime, and (knowing) that `get_object` is essentially a wrapper calling `valid_coordinates`, which is a sanity check that should not occur; both of these functions might be entirely removed or at least optimized.

After any optimization efforts, the program should be profiled again to verify improvements.

4.1.2 linux perf/perf_events

Perf is a profiler that is part of the Linux kernel; it uses syscalls to collect application (or system-wide) CPU performance statistics. It does not require any additional compilation flags.

For collecting a profile, the program is run with `perf record ./solution` and the profile is later visualized using `perf report`. The standard output is similar to gprof, excluding the call graph, but offers detailed insight on the functions' durations by their assembly instruction. Perf does provide broad access to different hardware events, such as cache statistics or context switches.

4.1.2.1 Example

As we can observe in Listing 1 and Listing 2, the bottlenecks are still `raycast`, `get_object`, and `valid_coordinates`. In relation, they show the same performance difference as in Listing 1, yet the total % time differs.

This is caused by:

- Perf profiles more external events, while gprof is limited to the code that was compiled.
- GProf does not account for its own instrumentation code, while Perf includes the performance of the syscalls.

- Generally, the output here is being truncated to display only the major functions.

```
Samples: 24K of event 'cycles:P', Event count (approx.): 5875520885
Overhead  Command   Shared Object         Symbol
  24.53%  solution  solution              [.] raycast
  13.33%  solution  solution              [.] get_object
   9.50%  solution  libm.so.6             [.] 0x000000000007eef4
   9.49%  solution  solution              [.] valid_coordinates
   5.00%  solution  [kernel.kallsyms]     [k] syscall_return_via_sysret
   4.24%  solution  [kernel.kallsyms]     [k] entry_SYSRETQ_unsafe_stack
   2.93%  solution  libm.so.6             [.] 0x000000000007eef0
   2.73%  solution  [kernel.kallsyms]     [k] entry_SYSCALL_64_after_hwframe
   2.48%  solution  libc.so.6             [.] putchar
   1.91%  solution  solution              [.] floor@plt
   1.79%  solution  libc.so.6             [.] _IO_file_overflow
   1.77%  solution  [kernel.kallsyms]     [k] entry_SYSCALL_64
   1.46%  solution  solution              [.] render_frame
   0.90%  solution  [kernel.kallsyms]     [k] n_tty_write
   0.86%  solution  solution              [.] construct_frame_column
   0.77%  solution  libm.so.6             [.] 0x000000000007eefa
   0.69%  solution  [kernel.kallsyms]     [k] n_tty_set_termios
...
```

Listing 2: Example of `perf report`

Legend:
Overhead % of total time
Command profiled command (here just the binary)
Shared Object Function source
Symbol Function name (Or address in case of missing names)

```
Samples: 24K of event 'cycles:P', 4000 Hz, Event count (approx.): 5875520885
raycast  /home/lennart/terminal-raycasting/solution [Percent: local period]
   1.47 │       mov       -0x20(%rbp),%rax
   2.50 │       movq      %rax,%xmm0
   2.00 │     → call      floor@plt
  14.82 │       cvttsd2si %xmm0,%ecx
   0.22 │       mov       -0x68(%rbp),%rax
   1.52 │       mov       %ebx,%edx
   0.17 │       mov       %ecx,%esi
   2.36 │       mov       %rax,%rdi
   1.97 │     → call      get_object
   3.28 │       mov       %rax,-0x30(%rbp)
   6.42 │       mov       -0x30(%rbp),%rax
  25.16 │       movb      $0x1,0x3(%rax)
   0.43 │       mov       -0x30(%rbp),%rax
   1.32 │       movzbl    0x1(%rax),%eax
        │       test      %al,%al
   1.01 │     ↑ jne       106
   0.22 │       movsd     -0x20(%rbp),%xmm2
   0.03 │       movsd     -0x18(%rbp),%xmm0
   0.01 │       mov       -0x70(%rbp),%rax
...
```

Listing 3: Example of assembly performance analysis

4.2 Visualization

Different visualizations offer specialized insights.

Flame graphs show an accumulated overview of function calls and their hierarchy, which gives a general clue about what the program is doing as the program execution progresses.

A *table*, listing the functions and their runtime statistics, provides condensed information about how to spend optimization efforts.

4.2.1 Tables

A tabular representation like the flat profile Listing 1 offers represents how much time is spent in which symbols. This is useful for quickly identifying bottlenecks .

These can be obtained directly from `gprof` or `perf report`.

Due to the number of symbols, low-time functions might be hidden or entirely missing due to the statistical nature of sampling.

4.2.2 Flame graphs

Flame graphs can be produced using an external tool, such as Brendan Gregg's FlameGraph tool (Description & Instructions), which operates on an already collected profile, such as the one from `perf` or `gprof`.

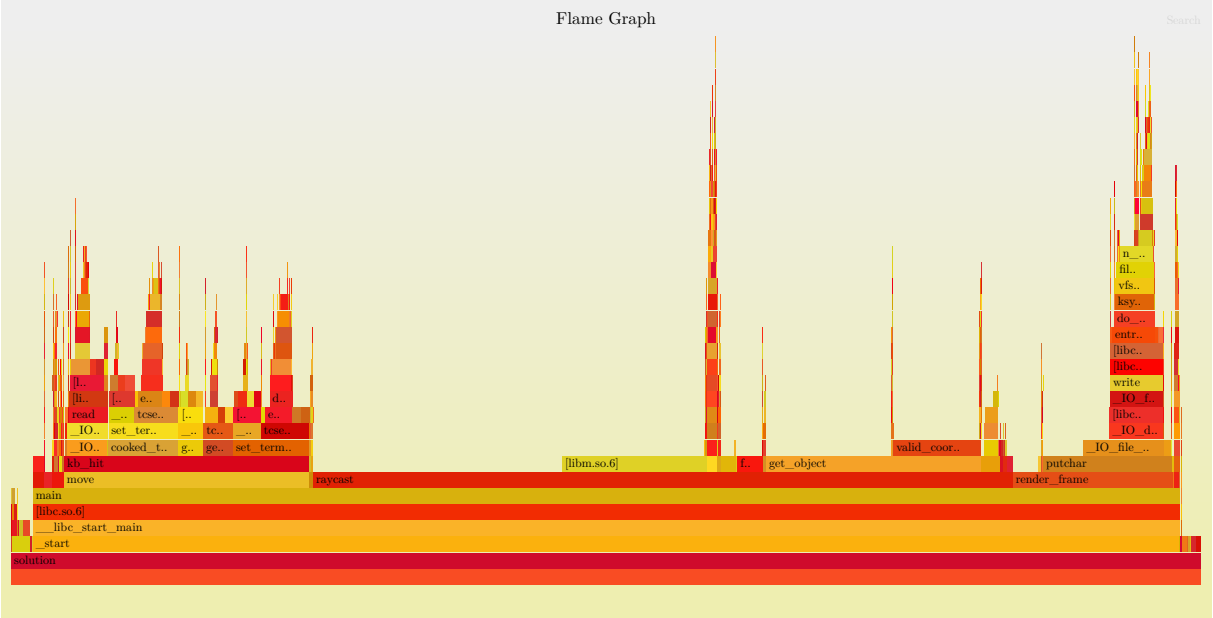


Figure 1: Raycasting flamegraph, showing the call hierarchy and time distribution. The coloring is merely aesthetic and not informational.

They give a hierarchical overview of the call stack, accumulating time spent in each child function. The width represents the amount of time spent in the function (and children), while calls made are stacked on top.

This allows to identify bottlenecks , as wide bars with no children indicate that the function is consuming a lot of time. The flamegraph in Figure 1 shows bottlenecks in the user code `raycast`, `get_object`, and `valid_coordinates` and in external code `[libm.so.6]`. (Knowing the code) and looking (or asking ChatGPT) at the assembly code Listing 3, it becomes clear that double to int conversion, flooring operations, and calls to short functions (`get_object` and `valid_coordinates`) take a lot of overhead.

4.2.3 Call graphs

Call graphs are similar to the tabular representation, but given per function each. The rows (per entry) contain all parents (above the function itself), the function itself (the row providing index and time%) and all children (below the function).

It is possible to see which children make up most of the time, and for how many of the total number of calls the parent is responsible for.

index The number given to the function (sorted by time%)
time% % of program runtime spent in this function and children
self time (seconds) of runtime spent in this function
children time (seconds) of runtime spent in this function and children
called (self/total) Number of calls to this child (self/total), or by this parent
name Child/Parent name

```
granularity: each sample hit covers 2 byte(s) for 0.66% of 1.51 seconds

index % time    self  children    called     name
                                                <spontaneous>
[1]     94.7    0.00    1.43                 main [1]
                0.59    0.73     501/501         raycast [2]
                0.10    0.00     501/501         render_frame [5]
                0.00    0.01     501/501         move [13]
                0.00    0.00     776/50876       kb_hit [12]
                0.00    0.00       1/1           initialize_game [19]
                0.00    0.00     590/590         ms_mov [24]
                0.00    0.00     501/501         turn [25]
                0.00    0.00       1/1           open_files [30]
                0.00    0.00       1/1           init_screen [29]
                0.00    0.00       1/1           destroy_screen [27]
                0.00    0.00       1/1           free_field [28]
                0.00    0.00       1/1           close_files [26]
-----------------------------------------------
                0.59    0.73     501/501         main [1]
[2]     87.4    0.59    0.73     501         raycast [2]
                0.46    0.16 41959892/42537746     get_object [3]
                0.04    0.00  112725/112725      construct_frame_column [7]
                0.03    0.00  112725/112725      v_dist [8]
                0.02    0.00     501/501         print_field [9]
                0.01    0.00  338175/338575      deg_to_rad [10]
                0.00    0.00     501/501         reset_ray_hits [15]
-----------------------------------------------
                0.00    0.00     201/42537746     valid_position [17]
                0.00    0.00  288576/42537746     reset_ray_hits [15]
                0.00    0.00  289077/42537746     print_field [9]
                0.46    0.16 41959892/42537746     raycast [2]
[3]     42.1    0.47    0.17 42537746         get_object [3]
                0.17    0.00 42537746/42539525     valid_coordinates [4]
```

Listing 4: Partial output of Gprof's call graph

E.g in Listing 4 `main [1]` all children are only called exclusivly by itself, while `get_object [3]` is called mostly by `raycast [2]` and a few times by `valid_position`.

5 Relevance in modern applications

(Also to answer the presentation question of wheter profiling can be applied for machine learning applications)

While most tools presented here are focusing on C similar languages and neglecting optimizations to keep the topic moderately simple, the concepts of profiling are still relevant in modern applications.

Many performance-critical applications are written in C-like languages, yet also higher abstractions such as Python are possible to profile. (Especially when using C extensions)

6 On-the-Fly profiling

This section will describe a conceptual development tool that builds upon profiling. The idea, implementation, usage, and limitations of this tool will be reviewed.

6.1 Motivation

The process of traditional code profiling is an efficient workflow for identifying bottlenecks individually, but especially for large programs that are not feasible to build and run iteratively, this workflow can become slow.

The paper [1] offers a complementary workflow: inspecting routine performance *On-the-Fly* (i.e., without any compilation or execution, while coding, visualized by the IDE).

6.2 Introduction

With increasing layers of abstraction, it becomes harder to have a deep understanding of the performance impact of specific decisions. Profilers, used correctly, can give a general notion of how much time is spent in what area of the implementation. As with any other toolset, profilers must be learned to be used efficiently.

Even then, optimizing a bottleneck might not be a clear task. Say the goal is to optimize appending data to a list; if the list is sorted, perhaps the sorting comparison criteria take a lot of time. It could also be the case that the CPU cannot be fully utilized, as linked-list traversal stages require memory fetching to complete before the next comparison can take place. This requires a deep understanding or precise profiling to figure out where time is spent inefficiently.

On-the-Fly profiling aims to support the developer in this aspect, showing *how* time is spent. This is based on categories introduced in [4].

Retiring Useful work (arithmetic, logic, reads/stores)
Bad-speculation Wrong speculative execution, e.g., branch mispredictions
Frontend-Bound CPU is waiting for instructions, e.g., cache misses
Backend-Bound CPU is waiting for data, e.g., slow memory fetches

6.3 Implementation

The prediction of performance distribution over the categories is based upon machine learning that takes roughly as input the C-code and provides a percentage distribution over the predicted CPU time distribution.

The model was trained using snippets of C code and the actual recorded CPU usage (by Intel VTune, another profiler) given over the categories.

6.3.1 Training

As the snippet as a string is not representative of how the code is executed, the paper uses different representations of the code (Abstract Syntax Tree, Control Flow Graph, and Data Flow Graph) as input for the model.

The model was developed using a dataset of 240 C projects, collected from GitHub (star rating was used as an indicator of quality).

Of these, only 83 were used for training, as many lacked documentation or a correct build process.

6.3.2 Accuracy

Testing the model was done by using the actual measured CPU usage of the project and comparing it to the predicted usage. The prediction was within 10% of the actual usage in 80% of the tests, and within 5% in 50% of the tests.

6.4 Limitations

The model and training process do not take compiler optimizations into account. The paper has tested 5 projects using optimization, and the accuracy was within 10% for 70% and within 5% for 40% of the tests.

6.5 Comparison to traditional profiling

Given the accuracy of the model, it is a viable alternative to profiling individual functions (in speed of applying the tool). The On-the-Fly method gives an idea of how the function might misperform, while traditional profiling gives you the time spent for itself and its children. Both require an understanding of how the code is executed (to understand why the yielded result is slow).

Yet a big part of profiling is the overview of the entire program, *what parts* are slow and *how much* time is spent. Identifying a bottleneck is more relevant than knowing how to optimize it.

The information that can be derived from profiles (performance regression, input growth, etc.) cannot be obtained using the On-the-Fly method. A new option is to see how the predicted distribution of an individual function changes over time, which could imply performance regression.

Aspect	On-the-Fly	Traditional Profiling
Function Analysis	Fast, predictive, mostly accurate	Slow, measured, precise
Program Overview	No	Yes
Information Derivation	New options, but no use-case	Established methods

Table 2: Comparison of On-the-Fly and traditional profiling

7 Conclusion

Profiling plays an essential role in measuring and optimizing large-scale applications or any performance-critical application. The tools can be used while actively looking to improve an application or in an automated fashion, generating a performance history which allows one to prevent regression or simply predict future performance.

The On-the-Fly method shows an interesting approach to effortlessly integrating the optimization stage into the code-writing process. While this method seems promising based on the provided statistics, it does not substitute for many aspects and use cases of traditional profiling tools. Should a usable tool be developed, it would be beneficial to use both methods at different stages in the development cycle.

Bibliography

[1] X. Hu *et al.*, "Towards On-The-Fly Code Performance Profiling," *ACM Transactions on Software Engineering and Methodology*, p. 3725212, Mar. 2025, doi: 10.1145/3725212.

[2] R. Bernecky, "Profiling, performance, and perfection (tutorial session)," in *Proceedings of the ACM/SIGAPL conference on APL as a tool of thought (session tutorials)* -, New York, New York, United States: ACM Press, 1989, pp. 31–52. doi: 10.1145/328877.328879.

[3] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, Jun. 1982, doi: 10.1145/872726.806987.

[4] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, CA, USA: IEEE, Mar. 2014, pp. 35–44. doi: 10.1109/ISPASS.2014.6844459.

[5] P. Wood, "A Survey of Performance Analysis Tools," [Online]. Available: https://www.cse.wustl.edu/~jain/cse567-06/ftp/perf_tools/index.html

[6] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu, "Can we trust profiling results?: understanding and fixing the inaccuracy in modern profilers," in *Proceedings of the ACM International Conference on Supercomputing*, Phoenix Arizona: ACM, Jun. 2019, pp. 284–295. doi: 10.1145/3330345.3330371.

[7] B. Gregg, "The flame graph," *Communications of the ACM*, vol. 59, no. 6, pp. 48–57, May 2016, doi: 10.1145/2909476.