Only Relative Speed Matters: Virtual Causal Profiling

Behnam Pourghassemi
University of California, Irvine
bpourgha@uci.edu

Ardalan Amiri Sani
University of California, Irvine
ardalan@uci.edu

Aparna Chandramowlishwaran
University of California, Irvine
amowli@uci.edu

ABSTRACT

Causal profiling is a novel and powerful profiling technique that quantifies the potential impact of optimizing a code segment on the program runtime. A key application of causal profiling is to analyze *what-if* scenarios which typically require a large number of experiments. Besides, the execution of a program highly depends on the underlying machine resources, e.g., CPU, network, storage, so profiling results on one device does not translate directly to another. This is a major bottleneck in our ability to perform scalable performance analysis and greatly limits cross-platform software development.

In this paper, we address the above challenges by leveraging a unique property of causal profiling: *only relative performance of different resources affects the result of causal profiling, not their absolute performance.* We first analytically model and prove causal profiling, the missing piece in the seminal paper. Then, we assert the necessary condition to achieve *virtual causal profiling* on a secondary device. Building upon the theory, we design **VCoz**, a virtual causal profiler that enables profiling applications on target devices using measurements on the host device. We implement a prototype of VCoz by tuning multiple hardware components to preserve the relative execution speeds of code segments. Our experiments on benchmarks that stress different system resources demonstrate that VCoz can generate causal profiling reports of Nexus 6P (an ARM-based device) on a host MacBook (x86 architecture) with less than 16% variance.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Causal profiling, performance analysis, virtual infrastructure, mobile devices, ARM and x86 architectures

1. INTRODUCTION

Profiling tools are fundamental to the system design and implementation process. They serve several functions such as pinpointing bottlenecks, guiding optimizations, and pruning design space in the overall goal to improve system performance. *Causal profiling* is one such powerful profiling technique [18]. The key idea behind causal profiling is to measure the effect of speeding up a selected line (e.g., function call) at runtime by slowing down all other concurrent functions. It indicates where the programmer should focus their optimization efforts and quantifies the potential impact of optimizations, enabling a wide range of what-if analysis. Causal profiling has been successful in analyzing the performance bottlenecks in large complex software systems such as the Chrome browser [22] and SQLite [18].

However, profiling can also be time-consuming and pose limits on cross-platform application optimization. Performance analysis on a diverse set of systems and configurations requires access to a broad range of devices. Such a setup is challenging to achieve in an academic research lab which restricts researchers to scope their profiling to a small number of systems.

This problem seems to be fundamental. That is, it seems like the only way to increase the coverage is to purchase and use a larger number of devices. Using Virtual Machines (VMs) in public cloud infrastructures, such as Amazon Elastic Compute Cloud (EC2) does not seem to be a feasible solution as the hardware specification of a VM can be different from that of the actual device, such as smartphones, tablets, desktops, laptops, and Chromebooks. This dissimilarity can affect the result of profiling to an extent where the conclusions cannot be relied upon. Cycle-accurate simulators and full-system emulators such as gem5 [16], QEMU [15], and Android Studio emulator [2] provide more accurate timing and performance characterization of applications. Although promising, instruction set simulators are prohibitively slow for full software stack performance analysis [25, 19, 26] and infeasible for *what-if* analyses under different scenarios that require a large number of experiments [18, 22]. Besides, conventional profiling tools typically do not work on top of such simulators [26].

In this paper, we argue that while this problem is fundamental in the general case, we show that causal profiling provides a *unique opportunity* to address this using virtual performance analysis. More specifically, we make a fundamental observation: *the result of causal profiling only depends on the relative execution speed of the application code segments.* Indeed, this relativity is at the core of causal profiling as it measures the potential speedup contributed by a program segment by slowing down all other code segments. Therefore, we can emulate the hardware configuration of, say, an ARM smartphone on an x86 laptop for causal profiling by controlling the relative performance of various re-

Performance Evaluation Review
Copyright is held by author/owner(s).

sources such as CPU, network, and storage.

To this end, this paper makes the following contributions.

1. We present an analytical model and proof of concept for causal profiling, a notable missing piece in the original paper. Then, we prove a necessary condition for virtual causal profiling on a secondary device.

2. Building on the above theory, we design VCoz, an infrastructure for virtual causal profiling that measures the impact of program speedups on various devices through hardware tunning of the host system. We implement a prototype of VCoz and port the causal profiler (originally designed for x86 architectures) to mobile devices (based on ARM) to validate our theory and prototype.

3. We test VCoz's cross-platform application optimization capabilities on multiple benchmarks with different workloads with MacBook Air as the host device and Nexus 6P as the target device. The experiments demonstrate that VCoz can predict the result of causal profiling with less than 16% variance while the original Coz profiler [6] misses the optimization opportunities.

2. CAUSAL PROFILING

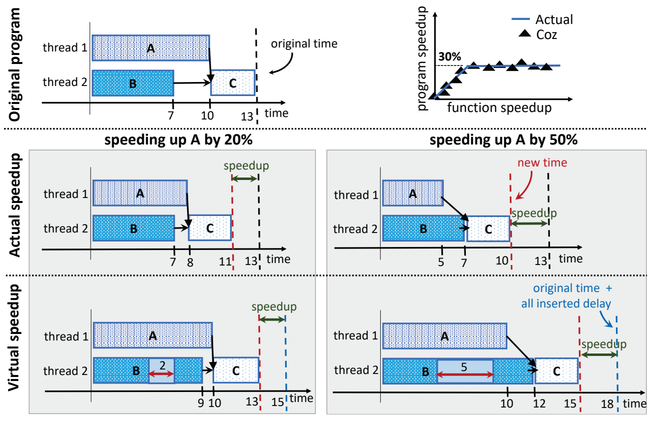


Figure 1: Illustration of the concept of *virtual speedup* and *causal profiling*. The top timeline shows the execution of the original program with 2 threads running functions A, B, and C and the what-if graph for function A. The middle two timelines correspond to actually speeding up function A by 20% and 50%, and the bottom timelines show the effect of virtually speeding up function A by 20% and 50%.

Causal profiling [18] is a novel method for finding performance bottlenecks and determining the impact of optimizations on a program. The Coz profiler [6] is the original implementation of the causal profiler. It is based on the idea of *virtual speedup* to find the impact of an optimization in a line of code (e.g., function call) on the total execution time of the program. Figure 1 illustrates the concept of *virtual speedup* with a concrete example. The top timeline shows the execution of the original program with two threads running functions A, B, and C and the dependency between them. The middle timeline demonstrates the effect of accelerating function A on the total execution time. The range indicated by *speedup* shows the *actual speedup* of the program after accelerating function A by 20% (left) and 50% (right). The bottom timeline presents the effect of *virtually* speeding up A. Whenever A is executing, all other concurrent threads are paused for a certain amount of time depending on how much one intends to accelerate A. For the left timeline, it is 20% of the function A, and for the right, it is 50% of A. The difference between the execution time of the program after virtual speedup and the original time of the program with all inserted delays (indicated by *speedup*) results in the same speedup as actually optimizing A (middle timeline). In general, Coz can generate a *what-if graph* for function A by varying the amount of virtual speedup in A and plotting the corresponding program speedup as shown in the top right corner of the figure.

3. VIRTUAL COZ

Even though Coz works in practice and the concept of causal profiling is comprehensible by examples, the authors do not provide a proof of their method in the original paper [18]. Therefore, in this section, we first prove the concept of virtual speedup and causal profiling with a mathematical paradigm. Then, we extract the critical condition for soundness of causal profiling which is the retention of the relative speed of code segments. Following that, we describe our methodology to translate this theory to practice.

3.1 Theory and Mathematical Formulation

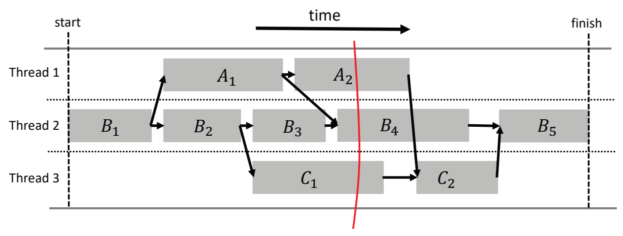


Figure 2: An example timeline for a program running on 3 threads. Edges show dependency between code segments in the program.

Suppose we have a program that runs on N threads. We can then divide the program into smaller code segments such that each segment (e.g., a function or basic block) runs entirely on only one thread. However, there might be dependencies between code segments, creating a critical path that determines the execution time of the program. Figure 2 illustrates an example program with 3 threads and 9 segments and the dependencies between them. Generally, for any code segment, s, we define $S(s)$ and $F(s)$ as the start and end times of s, $E(s) = F(s) - S(s)$ is the execution time of s, and $D(s)$ is the set of segments that s depends on. For any s, $F(s) = E(s) + \max(F(D(s)))$. Therefore, the total execution time, T is given by, $T = E(b_5) + \max(F(b_4), F(c_2))$ for the program in Figure 2. Without loss of generality, we present the proof for the program in the figure for readability. However, it applies to any program with multiple threads and arbitrary dependencies between code segments.

Recursively expanding the above equation and assuming the initial segment starts at time 0 ($S(b_1) = 0$), we have,

$$T = E(b_5) + \max \begin{pmatrix} E(b_1) + E(a_1) + E(a_2) + E(c_2) \\ E(b_1) + E(a_1) + E(b_4) \\ E(b_1) + E(b_2) + E(b_3) + E(b_4) \\ E(b_1) + E(b_2) + E(c_1) + E(c_2) \end{pmatrix} \quad (1)$$

Now, let us speed up an arbitrary segment s by ε seconds. The new execution time, T_{new}, can be calculated from the above equation by updating $E(s)$ (i.e., subtract ε). We state that T_{new} is equal to $T_{virtual} - \varepsilon$ where $T_{virtual}$ is defined as the total time derived by adding ε to all the segments in Equation 1 that a cutset (red line in Figure 2) passes through except the selected segment for speedup. In our example, if we desire to speedup segment a_2, we add ε to $E(b_4)$ and $E(c_1)$ but keep $E(a_2)$ unchanged to calculate $T_{virtual}$.

$$\begin{aligned} T_{virtual} &= E(b_5) + \max \begin{pmatrix} E(b_1) + E(a_1) + E(a_2) + E(c_2) \\ E(b_1) + E(a_1) + E(b_4) + \varepsilon \\ E(b_1) + E(b_2) + E(b_3) + E(b_4) + \varepsilon \\ E(b_1) + E(b_2) + E(c_1) + \varepsilon + E(c_2) \end{pmatrix} \\ &= E(b_5) + \max \begin{pmatrix} E(b_1) + E(a_1) + E(a_2) - \varepsilon + E(c_2) \\ E(b_1) + E(a_1) + E(b_4) \\ E(b_1) + E(b_2) + E(b_3) + E(b_4) \\ E(b_1) + E(b_2) + E(c_1) + E(c_2) \end{pmatrix} + \varepsilon \\ &= T_{new} + \varepsilon \qquad (2) \end{aligned}$$

Using 2, program speedup, S, relates to $T_{virtual}$ by,

$$S = \frac{T - T_{new}}{T} = \frac{T - (T_{virtual} - \varepsilon)}{T} = \frac{(T + \varepsilon) - T_{virtual}}{T} \quad (3)$$

Figure 2 illustrates the above mathematical formulation of speedup that underlines causal profiling [18]. Given this definition of speedup, the theorem that lays the foundation for the proposed virtual infrastructure is as follows.

Theorem. *If the execution time of all the segments is scaled by a constant factor α (i.e., $E(s^*) = \alpha E(s)$) and the speedup in a selected segment is also scaled by the same factor (i.e., $\varepsilon^* = \alpha\varepsilon$), then the new program speedup, S^* is the same as S and given by* $\frac{(T + \varepsilon) - T_{virtual}}{T}$.

Proof. We can prove the above theorem by extending Equation 1. Since the execution time of all the segments is scaled by α, the new execution time of the program, T^*, is now given by,

$$\begin{aligned} T^* &= \alpha E(b_5) + \alpha . \max \begin{pmatrix} E(b_1) + E(a_1) + E(a_2) + E(c_2) \\ E(b_1) + E(a_1) + E(b_4) \\ E(b_1) + E(b_2) + E(b_3) + E(b_4) \\ E(b_1) + E(b_2) + E(c_1) + E(c_2) \end{pmatrix} \\ &= \alpha T \qquad (4) \end{aligned}$$

Similarly, we can derive $T^*_{new} = \alpha T_{new}$ and $T^*_{virtual} = \alpha T_{virtual}$ (omitted due to space constraints). Note that ε is also scaled by α. Finally, combining Equations 3 and 4:

$$S^* = \frac{(T^* + \alpha\varepsilon) - T^*_{virtual}}{T^*} = \frac{(T + \varepsilon) - T_{virtual}}{T} \quad (5)$$

□

3.2 VCoz: Theory to Practice

The proved theorem is important and functional as it asserts that casual profiling of an application on one system will be similar to the causal profiling of that application on another system if all the code segments run $x\%$ faster or slower. In practice, every code segment is built from a series of CPU, memory, and I/O operations. So, we can hypothetically split a code segment into smaller slices of only CPU or memory or I/O operations as demonstrated in figure 3 by different colors. In figure 3(a) two example code segments (A and B) are run on a target device. The relative execution time of these two code segments is $78/56 = 1.4$. Figure 3(b) shows the same two segments executed on the host device with dissimilar hardware components. In this example, the host has a processor that runs 20% slower ($\alpha_{cpu} = 1.2$), a memory with 50% slower bandwidth ($\alpha_{mem} = 1.5$), and I/O systems (including disk, network, etc.) that operate 2× faster ($\alpha_{io} = 0.5$). Therefore, the relative execution time of the code segments is different on the host ($84/71 = 1.2$), leading to incompatible causal profiling of the target device. To achieve a similar relation, the different slices in the code segments have to retain an identical scaling factor from target to host. In figure 3(c), the CPU and I/O on the host are adjusted to match the speed of memory, i.e., the processor runs 25% ($\alpha_{mem}/\alpha_{cpu}$) slower and I/O systems $3\times(\alpha_{mem}/\alpha_{io})$ slower. This adjustment preserves the relative execution time of the two segments on the target device since all the slices scale by $\alpha = \alpha_{mem} = 1.5$. Figure 3(d) demonstrates another adjustment to the host hardware to conserve the relative execution time of code segments but this time, both segments scale by $\alpha = \alpha_{io} = 0.5$. For this case, CPU and memory are adjusted to operate 3× (α_{cpu}/α_{io}) and 2.4× (α_{mem}/α_{io}) faster, respectively.

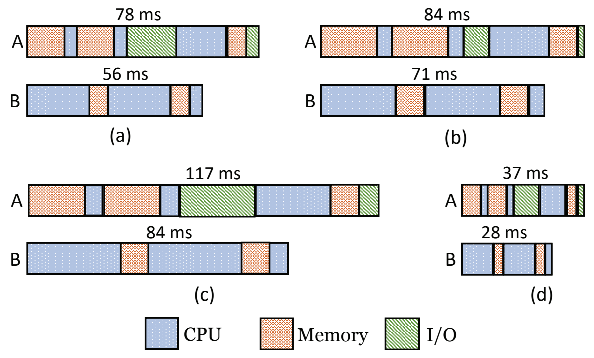


Figure 3: Two code segments A and B (split into CPU, memory, and I/O slices) running on (a) target device; (b) host device with different hardware component speeds; (c) host device with all hardware components executing 50% slower than the target device; d) host device with all the hardware components executing 2× faster.

Based on this idea, we design and implement a prototype of VCoz that configures the hardware components of the host system to simulate the causal profiling of target devices. Figure 4 presents the design of VCoz. First, VCoz estimates the performance scaling factor of each component on the host. To do so, it either compares spec of two hardware components or runs *performance tests* where microbenchmarks stress distinct hardware resources on both systems and compares the results. For any pair of target and

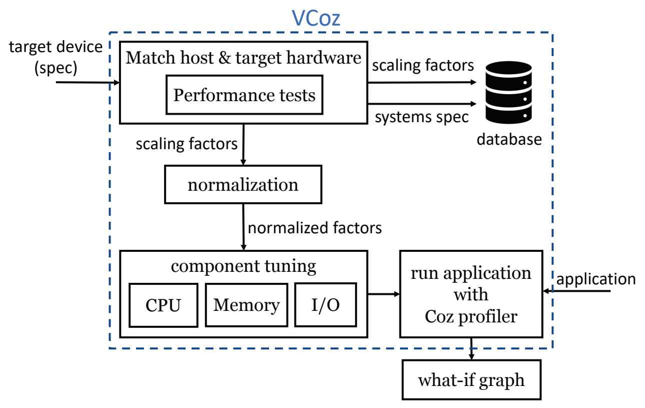


Figure 4: Design overview of VCoz containing inputs, modules, and outputs of each module.

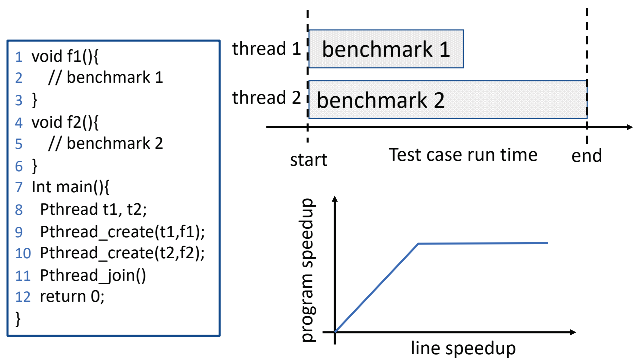
host, the stress tests are performed only once, and the measured scaling factors are stored in a *database* to eliminate the need for future access to the same devices.

After estimating the scaling factor of each hardware component, VCoz optionally normalizes them. *Normalization* handles two scenarios that arise on real devices. (i) In practice, sometimes the host system does not provide a knob for tuning a hardware component (e.g., memory). In this case, all the other resources are normalized by a scaling factor determined by the untunable component. (ii) The scaling factor of a component can be beyond the range of configurable values. For example, consider the CPU, and let's say it has to be scaled by 5×. However, the attainable CPU frequency on the host may limit the scaling to at most 4×. In this case, VCoz divides all the resources' (e.g., CPU, memory, and I/O components) scaling factors by 5.

Component tuning adjusts the hardware component according to the normalized factors. In our prototype, we normalize by memory since most of the systems do not expose a knob to change RAM operating frequency. Otherwise, VCoz changes RAM frequency either in BIOS or OS-level. For CPU, we modify the clock speed of all the cores using the `CPUfreq` governor in the Linux kernel [7]. For I/O, we adjust the interface of each I/O peripheral. Currently, VCoz configures the network interface using Linux traffic control utility (`tc`) [10] to restrict the uplink and downlink bandwidth of the system. It can also be easily extended to support other I/O components such as disk (using `hdparm` [9] in the Linux kernel). Once all the hardware components are tuned, VCoz runs the original Coz profiler on the application to generates a *what-if graph*. According to the theory, the host generated what-if graph will be the same as the one generated by Coz on the target device.

4. EXPERIMENTAL SETUP

To validate the above hypothesis we run Coz [6] directly on the target device and compare it against VCoz for different applications. The details on porting Coz to mobile devices are presented in appendix A. Figure 7 shows our testbed for this validation. The code snippet spawns two threads where each thread invokes a different benchmark. The top diagram in Figure 7 shows the program execution timeline for this testbed. We select the line corresponding to the slowest benchmark (i.e., line 10 which belongs to thread 2) for speedup. Consequently, Coz generates a what-if graph similar to the one shown at the bottom of the figure.



```
1  void f1(){
2     // benchmark 1
3  }
4  void f2(){
5     // benchmark 2
6  }
7  Int main(){
8    Pthread t1, t2;
9    Pthread_create(t1,f1);
10   Pthread_create(t2,f2);
11   Pthread_join()
12   return 0;
}
```

Figure 5: Description of test-case: the baseline code, program execution timeline (top diagram), and expected what-if graph generated by Coz profiler (bottom diagram).

We integrate benchmarks with different types of workloads in our testbed: CPU-intensive (LU and Cholesky decomposition from Splash [24]), memory-intensive (stream benchmark [14]), and I/O-intensive (network client-server benchmark). Note that benchmarks can have a mix of all three types of instructions (CPU, memory, and I/O). For example, Cholesky decomposition contains memory load/store operations to read/write matrices. However, to examine the impact of hardware component tuning in VCoz, we consider benchmarks to primarily stress one component.

The target device is Nexus 6P with quad-core ARM Cortex-A53 + quad-core ARM Cortex-A57 processor, and the host device is MacBook Air 2.2 GHz Intel Core i7 processor. Both systems have Low Power Double Data Rate (LPDDR) dual channel memory operating on 1600MHz, so they are expected to have similar memory performance, i.e., $\alpha_{mem} = 1$. The processors, however, have different architecture, so VCoz needs to find the CPU scaling factor. For this reason, VCoz runs multiple CPU-intensive benchmarks (from Phenoix [23] and Splash [24] test suites) and compares the execution time on the host and target devices as shown in Table 1. We observe that the host x86 processor computes approximately 2.3× faster than the mobile ARM processor (i.e., $\alpha_{cpu} = 2.3$). We consider the network interface as an example of I/O in this paper. For each experiment, we try 20 different speedups from 0 to 100% and 4 to 8 profiling runs for each speedup.

Benchmark	Nexus 6P	MacBook Air	ratio (α_{cpu})
Matrix Multiply	3.1 s	1.5 s	2.1
FFT	56 ms	23 ms	2.4
LU	2.3 s	1.0 s	2.3
Word Count	38 s	16 s	2.3
Cholesky	1.1 s	450 ms	2.4
PCA	690 ms	300 ms	2.3

Table 1: Execution time of 6 CPU-intensive benchmarks on Nexus 6P and MacBook Air.

5. RESULTS

We compare the results of virtual causal profiling (VCoz) with the causal profiler (Coz) that runs directly on the device to assess the functionality of VCoz and evaluate the accuracy of our prototype. Figure 6 shows the corresponding what-if graphs in purple and blue lines, respectively, for various combinations of benchmarks that stress different hardware resources. Additionally, we also compare them against the results generated by Coz on the host x86 system (yellow line) to evaluate the impact of hardware tuning.

5.1 CPU and Memory test-cases

CPU-CPU. We first consider benchmarks with a similar workload in the testbed (section 4). We run two different CPU-intensive benchmarks namely, Cholesky and LU decomposition, concurrently and the results are shown in the leftmost plot in Figure 6. As we can observe, VCoz predicts the result of causal profiling with at most 13.6% variance. Moreover, the graphs generated by Coz on the host (no CPU frequency tuning) and VCoz with the host CPU frequency tuned to 1.5 GHz are comparable to VCoz with accurate CPU tuning (900 MHz), which indicates that CPU tuning is not effective for this test case. This supports our theorem since scaling CPU by any factor would homogeneously scale both execution paths, maintaining the relative speed of code segments and hence producing identical what-if graphs.

CPU-Memory. In this test case, we run the stream benchmark on one thread and LU decomposition on the other to evaluate a combination of two different workloads: memory-intensive and CPU-intensive. According to Figure 6, VCoz predicts the program speedup on the target device with less than 15.8% error on the host system while Coz incurs more than 50% error. Note that CPU and memory have different scaling factors in our experiments. Therefore, this test case highlights the limitation of Coz in profiling devices where CPU code segments execute at different relative speeds compared to memory code segments, thereby violating the necessary condition for causal profiling.

5.2 I/O test-cases

Now we evaluate the behavior of VCoz when the program has I/O operations. Here we run a network-heavy benchmark (server-client data streaming) as an example of an I/O-intensive workload. On the concurrent thread, we run a CPU-intensive benchmark (LU decomposition).

Matched network. First, both systems are connected to the same network (100 Mbps), which allows VCoz to match the I/O speed without adjusting the network module of the host system (i.e., $\alpha_{io} = 1$). So, it only adjusts the CPU frequency. As we can infer from the third plot in Figure 6, VCoz predicts the result of Coz on the device with high precision (less than 9.1% variance). Meanwhile, Coz is unable to uncover the potential impact of optimizations on the target device since it incorrectly predicts marginal program speedup.

Different network. Now, we connect the Nexus 6P to a slower network connection, 17.7 Mbps. This is the global average network connection for mobile devices in 2020 [1, 5]. Since the I/O speed of the target device is modified, VCoz reapplies component matching. The I/O speed is now 5.6× slower, and the CPU computing speed is 2.3× slower. If VCoz normalizes by the I/O scaling factor, the CPU has to run 2.4× faster. Given the operating frequency of the host system is (2.2 GHz) and the maximum available CPU frequency is 3.1 GHz (in turbo boost mode), VCoz cannot fulfill this scaling factor. However, if we normalize by the CPU scaling factor, the network speed has to reduce by 2.4×, and VCoz can satisfy this by adjusting the network interface of the host system to limit the bandwidth to 41 Mbps. Figure 6 shows how remarkably VCoz predicts the actual causal profile with less than 11.4% error. Meanwhile, Coz cannot detect the behavior of the critical path (no changes in the critical path) in the range of speedup because of missing characterization of the underlying hardware of the target device.

6. RELATED WORK

Profilers are an important tool for performance analysis of applications. Tools such as gprof [21] and Oprofile [11] along with the Linux built-in hardware counter profiling tool, perf [12], and the equivalent for Android, simplePerf [13] are among the popular profiling tools for desktop and mobile developers. These tools rank code by its contribution to total execution time, however, code that runs for a long time is not necessarily a good candidate for optimization. Causal profiling captures the dynamic behavior of the critical path of the application because it applies virtual speedups directly into the execution path at runtime. Several recent studies [28, 27, 22] have applied causal profiling for identifying bottlenecks in multithreaded applications and/or what-if performance analysis of large software systems such as web browsers [22] which further underscores its potential impact.

Besides profilers, developers exploit simulators for performance analysis for cross-platform application development and optimization. Cycle-accurate instruction set simulators (ISS) such as gem5 [16] and QEMU [15] and full-system emulators such as Android studio emulator [2] and Appetize [4] for iOS generate relatively accurate timing and power consumption for system-level performance analysis. However, simulation on these platforms is considerably slower which makes it infeasible for comprehensive what-if analysis [26, 25, 19] which typically requires a large number of experiments to explore the hardware and software design space. Multiple prior efforts have attempted to enhance the functionality and speed of ISS for performance and power analysis of mobile platforms [17, 20]. Nonetheless, they are still slow making them infeasible for causal profiling [26]. In this paper, we present the theory and proof-of-concept of virtual causal profiling which enables scalable cross-platform performance analysis on real devices.

7. CONCLUSIONS

We present a theoretical formulation for causal profiling and extract a necessary condition for virtual causal profiling. According to the theory, the result of causal profiling only depends on the relative execution speed of the different code segments. Therefore, we design VCoz that enables virtual causal profiling by emulating the hardware configuration of say, mobile devices on a laptop/desktop for causal profiling by controlling the relative performance of various resources, such as CPU, memory, and network. We implement a prototype of VCoz and evaluate it on multiple benchmarks with a combination of different workloads (CPU-intensive, memory-intensive, and I/O-intensive). Our results show that VCoz can predict the results of causal profiling with significant accuracy by tuning different hardware components. For example, VCoz predicts the outcome of the Coz profiler on an Arm-based Nexus 6P mobile device with less than 16% variance on an x86 laptop while the origi-

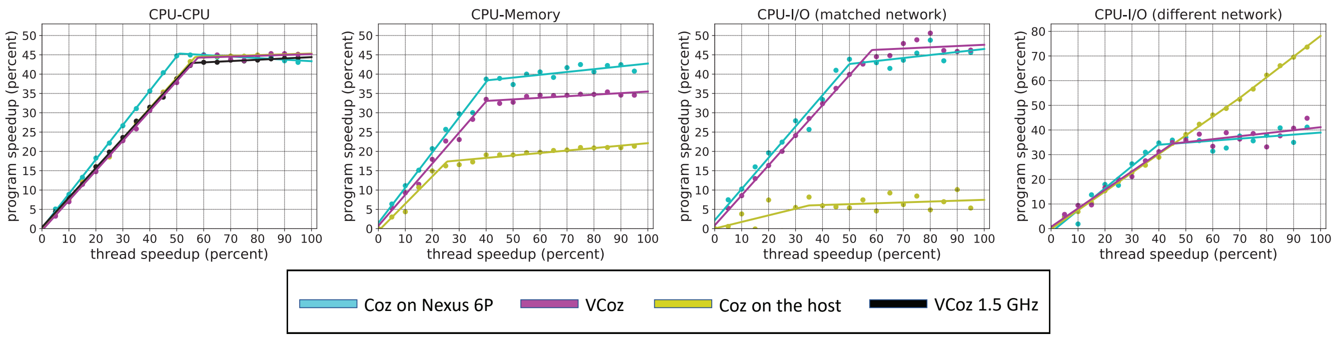


Figure 6: Comparison of VCoz and Coz on host and target devices for different test cases. Plots from left to right: 1) CPU frequency tuning on a program with two streams of compute-heavy code. 2) Memory and CPU heavy code segments. 3) I/O and CPU heavy code segments. Both devices are connected to 100 Mbps network connection. 4) I/O and CPU heavy code segments. Host is connected to 100 Mbps network connection and Nexus 6P is connected to 17.7 Mbps.

nal Coz profiler misses predicting optimization opportunities or generates inaccurate what-if graphs. As a result, VCoz advances the state-of-the-art in designing practical profilers that are much needed for scalable cross-platform application development.

8. ACKNOWLEDGEMENTS

This project is partially supported by a Google Faculty Research Award and the National Science Foundation (NSF) under the award number 1939237.

9. REFERENCES

[1] Cisco annual internet report (2018–2023) white paper, Feb 2017. https://www.statista.com/statistics/371894/average-speed-global-mobile-connection/.
[2] Android emulator, May 2020. https://developer.android.com/studio/run/emulator.
[3] Android NDK, May 2020. https://developer.android.com/ndk.
[4] Appetize, May 2020. https://appetize.io/.
[5] Average global mobile network connection speeds from 2016 to 2021, May 2020. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.
[6] Coz profiler, May 2020. https://github.com/plasma-umass/coz.
[7] CPU performance scaling, May 2020. https://www.kernel.org/doc/html/v4.14/admin-guide/pm/cpufreq.html.
[8] Google android bionic, May 2020. https://android.googlesource.com/platform/bionic.
[9] Linux hdparm, May 2020. http://man7.org/linux/man-pages/man8/hdparm.8.html.
[10] Linux traffic control, May 2020. http://man7.org/linux/man-pages/man8/tc.8.html.
[11] Oprofile - a system profiler for linux, May 2020. https://oprofile.sourceforge.io/.
[12] Performance profiling with linux perf, May 2020. http://man7.org/linux/man-pages/man1/perf.1.html.
[13] simpleperf, May 2020. https://developer.android.com/ndk/guides/simpleperf.
[14] Stream benchmark, May 2020. http://www.cs.virginia.edu/stream/ref.html.
[15] F. Bellard. Qemu, a fast and portable dynamic translator.
[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.
[17] N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. Gemdroid: a framework to evaluate mobile platforms. *ACM SIGMETRICS Performance Evaluation Review*, 42(1):355–366, 2014.
[18] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197. ACM, 2015.
[19] B. Franke. Fast cycle-approximate instruction set simulation. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 69–78, 2008.
[20] M. Ju, H. Kim, and S. Kim. Mofysim: A mobile full-system simulation framework for energy consumption and performance analysis. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–254. IEEE, 2016.
[21] S. L. G. P. B. Kessler and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the Symposium on Compiler Construction*. Citeseer, 1982.
[22] B. Pourghassemi, A. Amiri Sani, and A. Chandramowlishwaran. What-if analysis of page load time in web browsers using causal profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–23, 2019.
[23] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.
[24] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.
[25] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver. A structured approach to the simulation, analysis and characterization of smartphone applications. In *2013 IEEE International Symposium on Workload Characterization*, pages 113–122. IEEE, 2013.
[26] C.-H. Tu, H.-H. Hsu, J.-H. Chen, C.-H. Chen, and S.-H. Hung. Performance and power profiling for emulated android systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 19(2):1–25, 2014.
[27] A. Yoga and S. Nagarakatte. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–501, 2019.
[28] T. Yu and M. Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of*

the 25th International Symposium on Software Testing and Analysis, pages 389–400, 2016.

APPENDIX

A. PORTING COZ TO MOBILE DEVICES

The existing implementation of Coz is merely designed for and tested on the desktop applications (e.g., applications running on x86 systems) that host Linux OS. Therefore, the current version is not compatible with the majority of smartphones and mobile devices in the market. To improve the usability of this profiler among mobile users, we leverage Coz to support profiling of the applications that run on Android devices with ARM architectures. However, this extension is non-trivial and requires substantial re-engineering and troubleshooting of Coz and its dependent libraries. Figure 7 shows an overview of the Coz profiler and its interaction with the dependent libraries. The Coz bootstrapper in the figure interposes the entry point of the target application (`__libc_start_main`) to initiate a profiler instance under the same process. This profiler instance uses third-party libraries (e.g., `libdwarf.so`) to read and process debugging information of the target application and C++ Pthread library to interpose Pthread APIs. It also invokes Linux `perf_event` system calls under the hood to monitor and profile the target application.

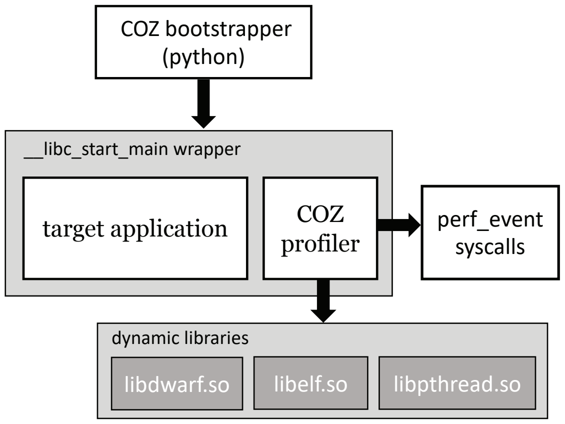


Figure 7: Overview of COZ profiler and dependent libraries.

To safely port Coz to Android devices, we rewrite its bootstrapper for two main reasons. First, the current code is not compatible with native Android apps. For instance, Google Android Bionic [8], the standard C library for the Android operating system, uses different entry points (e.g., `__libc_init`) and startup procedure than Linux standard C library. Second, most of the bootstrapper is written in Python which is not a standard language for Android systems. So, we rewrite the code in C++ and generate a native executable that can be safely launched directly by Android applications or via ADB shell without requiring a Scripting Layer for Android (SL4A).

Google's Android development kit (NDK) [3] is the official toolset for porting C/C++ programs to Android devices, however, not all of the C++ APIs and libraries for GNU/Linux are implemented by Android NDK. Therefore, the majority of our effort is to provide a workaround for missing functionalities in the Coz code and third-party libraries. In some cases, we implement the missing functions (for example, `std::to_string()`, an STL function used in `libelf.so` and `libdwarf.so`) or replace them with compatible implementations. For instance, we use stack tracing APIs implemented in `stdio.h` instead of `execinfo.h`. The most challenging part, however, is to find a workaround for Pthread APIs since Coz interposes several Pthread APIs to handle thread suspension in the target application. Two fundamental issues with Pthread APIs prevent the current implementation of Coz to be ported to Android devices. 1) The entire Pthread API is not supported on Android devices. That being said, we remove all associated modules that belong to unsupported Pthread APIs (such as `sigwaitinfo, sigtimedwait, pthread_tryjoin_np, pthread_sigqueue, pthread_timedjoin_np, pthread_barrier_wait`) from the code. This pruning does not invalidate the functionality of the profiler since valid Android applications do not have calls to the unsupported Pthread APIs. 2) Pthread APIs are directly implemented through C/C++ library on Android rather than a separate Pthread library as in GNU/Linux systems. Thus, the existing implementation for loading and interposing symbols of the Pthread dynamic library, i.e., using `dlopen()` and `dlsym()` to load `libpthread.so`, is not compatible with Android. We fix this issue by providing Android-compatible wrapper functions for loading Pthread symbols.

Android NDK contains the implementation of the majority of the Linux perf APIs for Android devices. Since Perf APIs use kernel syscalls and read hardware counters, we have to validate and inspect their functionality on Android devices. Accordingly, we sample hardware counters used by Perf APIs in Coz on the Nexus 6P mobile with ARM v8 processor for multiple benchmarks in the Phoenix benchmark suite [23]. We then compare our log file with the log file from the same benchmarks on the x86 system hosting Linux OS and confirm the compatibility of the Perf APIs used in Coz as well as the functionality of the entire Coz sampling modules. Finally, we modify the Coz build and configuration files and pass specific switches and configurations for building ARM targets to the Android NDK compilers.