Effective Performance Issue Diagnosis with Value-Assisted Cost Profiling

Lingmei Weng
Columbia University

Yigong Hu
Johns Hopkins University

Peng Huang
University of Michigan

Jason Nieh
Columbia University

Junfeng Yang
Columbia University

Abstract

Diagnosing performance issues is often difficult, especially when they occur only during some program executions. Profilers can help with performance debugging, but are ineffective when the most costly functions are not the root causes of performance issues. To address this problem, we introduce a new profiling methodology, *value-assisted cost profiling*, and a tool vProf. Our insight is that capturing the values of variables can greatly help diagnose performance issues. vProf continuously records values while profiling normal and buggy program executions. It identifies anomalies in the values and the functions where they occur to pinpoint the real root causes of performance issues. Using a set of 15 real-world performance bugs in four widely used applications, we show that vProf is effective at diagnosing all of the issues while other state-of-the-art tools diagnose only a few of them. We further use vProf to diagnose longstanding performance issues in these applications that have been unresolved for over four years.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: Debugging; profilers; program analysis

ACM Reference Format:
Lingmei Weng, Yigong Hu, Peng Huang, Jason Nieh, and Junfeng Yang . 2023. Effective Performance Issue Diagnosis with Value-Assisted Cost Profiling. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3587444

1 Introduction

Performance issues are prevalent in deployed systems and are notoriously difficult to diagnose. To help developers diagnose these issues, numerous profilers [4, 18, 20, 22, 26] have been designed. Unfortunately, in practice, even with mature profilers, it often takes a developer a long time to figure out the root cause of a performance issue. In a real-world performance debugging story [27], the developer "*spent 5 hours debugging, and finally moved a single line of code up 10 lines*", which reduced the CPU usage by 20×. Although the fix was simple, it took the developer many hours to find the bug, because the profiler results suggested the wrong places to investigate.

```
830 void recv_sys_init() {
831   ...
846   recv_n_pool_free_frames = buf_pool_get_n_pages() / 3;
847
3192 bool recv_scan_log_recs(ulint available_mem, ...) {
3203   bool finished  = false;
3348   if (recv_parse_log_recs(checkpoint_lsn,
3349         store_to_hash, available_mem, apply)) {
3355     finished = true;
3356     goto func_exit;
3357   }
3376 }
3388 bool recv_group_scan_log_recs(lsn_t ckpt_lsn, ...) {
3417   ulint available_mem = srv_page_size *
3418         (buf_pool_get_n_pages() -
3419         (recv_n_pool_free_frames * srv_buf_pool_ins));
3424   do {                                    function has more
3431     recv_apply_hashed_log_recs(false);    than 200 LOC
3439     log.read_log_seg(&end_lsn, start_lsn + RSCAN_SIZE);
3440   } while (end_lsn != start_lsn &&
3441         !recv_scan_log_recs(available_mem, ...
```

Figure 1. A real performance issue in MariaDB (MDEV-21826).

Such anecdotal examples widely exist. A key reason is that traditional profilers focus on identifying costly functions. They are effective when the performance bug happens to be in the function that takes the most time. However, tricky performance bugs are often caused by improper code logic. The buggy code itself may be fast and ranked low by profilers, misleading developers to waste effort trying to speed up costly functions that are necessary and already highly optimized.

Figure 1 shows a real performance issue [9] in the widely used MariaDB as an example. Based on user-provided logs, developers suspected that the user's database caused an out-of-memory error. Existing profilers report that the function `recv_apply_hashed_log_recs` consumes most of the execution time, but this is not the root cause. From its call count, developers recognized that this function was called frequently. This could mean that the function is too costly to be executed frequently and needs to be further optimized. Alternatively, it could mean that there is an issue with the calling of the function. Knowing which answer is correct is difficult when the root cause is unknown. In this case, digging into and trying

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3587444

to optimize the `recv_apply_hashed_log_recs` function would waste huge amounts of time since it has hundreds of lines of code and 20 branches. The developers ended up not doing that and instead focused on the loop that calls the function. Nevertheless, they still ended up wasting significant time investigating the loop conditional and the call chains from `recv_scan_log_recs` to `recv_parse_log_recs`. Each function was complex, leading to a wild goose chase.

The real root cause is inside functions `recv_sys_init` and `recv_group_scan_log_recs`. `recv_sys_init` incorrectly sets variable `recv_n_pool_free_frames` to one-third of the buffer pool (line 846). It is used in `recv_group_scan_log_recs` to calculate variable `available_mem` (line 3417), incorrectly setting it to zero. As a result, `recv_scan_log_recs` returns false, causing wasteful computation in the loop (line 3441). The problem was not in the loop where the developers spent significant time, but in code before the loop. Developers missed focusing on the crucial beginning of the function `recv_group_scan_log_recs` before the loop, as profilers provided no indication that this function was costly or important. Eventually, developers took 20 days to find the root cause, with the user being actively involved, even when their initial suspicions of an out-of-memory error turned out to be correct.

Our insight is that existing profilers' gaps are often caused by the lack of program data-flow information in the profiling result. Information such as the length of an array, the value of a variable, and the history of a variable's values during the execution is indispensable in debugging. Indeed, we observe that, in debugging complex performance issues, developers often have to take additional steps including adding *ad-hoc* `printf` statements, re-compiling and re-executing the software, and attaching a debugger like `gdb`, to obtain data-flow information to guide performance debugging.

Based on this insight, we introduce a new profiling methodology, *value-assisted cost profiling*. Its basic idea is to not only measure execution costs during profiling, but also *continuously* record the values of program variables to provide data-flow information. The recorded values are then used to distinguish anomalous costly functions from necessarily costly functions to localize the root cause in the code.

We build a tool *vProf* by modifying the popular `gprof` [20] profiler to realize value-assisted cost profiling, addressing three key challenges. First, vProf needs to decide which variables to record and how to locate them at runtime. Simply recording all variables and the complete program data-flow would incur unacceptable overhead, and invalidate the profiling results. Second, vProf needs to record variables efficiently in a manner that aligns well with other profiling information so it can be useful. Third, vProf needs to use the recorded value information to improve the diagnosis of performance issues.

vProf decides which variables to record by using static analysis to identify the types of program variables that commonly influence performance. vProf uses an LLVM [29] analysis pass to scan the source of the target program to identify these variables, typically generating hundreds to thousands of candidate variables.

vProf not only needs to identify which variables to record, but also reliably locate them at runtime. The runtime location of a variable, especially a local variable, can change during program execution, such as being stored in different registers, pushed onto the stack, or becoming dead or out of scope. Like `gprof`, vProf presumes debugging information is available in the target program executable, which it statically analyzes to obtain variable scope and location information. This is used to record the variable values at runtime.

vProf records variables efficiently at runtime in a manner aligned with other profiling information by leveraging the same mechanism it uses for measuring execution costs. Like existing profilers such as `gprof`, to minimize the overhead, vProf uses program counter (PC) sampling to measure execution costs per function. It sets a periodic alarm such that at each alarm signal, vProf records the current PC to identify which function is executing. The executing cost of a function is determined based on how often PC samples occur in its address range. vProf leverages this same approach to passively record variable values at each alarm signal, which we refer to as *value samples*. vProf not only records value samples for variables accessible at the current PC, but also virtually unwinds the stack to record additional value samples in callers of the current execution context, as well as the PCs at which they are accessed. vProf introduces efficient data structures so that the variables accessible at a given PC can be quickly identified and recorded.

vProf improves the diagnosis of performance issues by introducing a novel post-profiling analysis algorithm that combines value samples with traditional profiling execution costs. Using only value samples is insufficient for performance debugging, as they can be noisy. The value samples themselves also do not carry any information about costs, while costs are central to performance reasoning. Instead, vProf uses value samples to calibrate raw execution costs in two ways.

First, in addition to computing function execution cost based on PC sampling, vProf uses value samples recorded with virtual stack unwinding to calculate a *variable-based function execution cost* based on how often value samples occur in functions. The idea is that a function that has variables of interest that calls other functions should be considered more carefully even if its own execution time may not be that high. This is done by having the caller effectively inherit the execution cost of its callees, thereby making it appear more costly. A function that does not have variables of interest will have no value samples, so its variable-based execution cost will be zero. vProf assigns each function a raw execution cost which is the greater of the execution cost based on PC sampling and the variable-based execution cost.

Second, vProf computes a *discount ratio* for each profiled function based on the degree to which its associated value samples are anomalous. Anomalous values are determined by

comparing value samples between normal and buggy executions of a target program. The more anomalous a function's variable values, the lower the function's discount ratio will be. vProf then weighs a function's raw execution cost by one minus its discount ratio. Discounting demotes necessarily costly functions and promotes suspicious, lower-ranked functions. vProf further identifies the basic blocks in which anomalous values occur to help developers localize the root cause of a performance issue.

We evaluated the effectiveness of vProf against other state-of-the-art tools, including `gprof`, `perf` [18], COZ [12], and statistical debugging [40]. We collected and reproduced 15 real-world performance bugs in large server applications, including Apache, MariaDB, PostgreSQL and Redis. We then used these various tools to attempt to diagnose the bugs. vProf ranks the root cause function first for seven of the bugs and within the top five for all 15 bugs. It significantly outperforms the other tools, which at best ranked the root cause function within the top five for at most six of the bugs.

We show that vProf has low profiling overhead, does not require explicit instrumentation or code changes to target programs, and provides a similar usage model to `gprof`. These properties make vProf a practical tool to assist developers to debug tricky performance issues. In fact, we used vProf to diagnose several previously unresolved performance bugs in MariaDB and Redis, which have been confirmed by their developers, demonstrating its usefulness in practice.

2 Overview of vProf

Figure 2 shows the workflow of vProf, which can be decomposed into four steps. First, a developer runs vProf's schema generator to extract a list of variables in the target program to monitor during profiling. This schema generator uses static analysis on the program source code to automatically identify variables in instructions that likely influence a program's performance, such as global variables, variables in conditional expressions, and call parameters, as discussed in Section 3. It records the definition locations of all identified variables. For example, in Figure 1, vProf identifies the variables `recv_n_pool_free_frames` and `available_mem` for monitoring, the former since it is a global variable and the latter since it appears in a conditional expression as a call parameter of the function recv_scan_log_recs (line 3441).

Second, the developer compiles the target program with the `-pg` flag, the same as using `gprof`, so that the resulting executable contains DWARF debugging information [15]. This is used to translate the generated schema into runtime location information for the variables of interest. For example, in Figure 1, the global variable `recv_n_pool_free_frames` is accessed via its memory address, but the local variable `available_mem` is accessed from a register determined by the compiler. vProf uses the debugging information to determine what register to use to access `available_mem`.

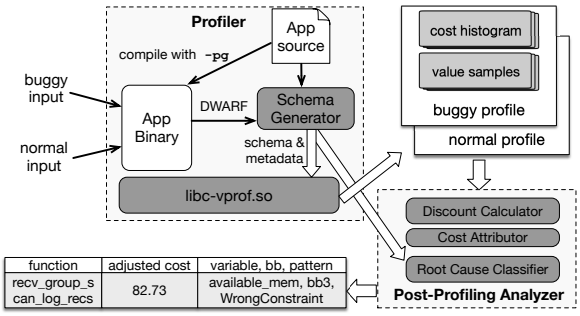


Figure 2. Workflow of vProf.

Third, the developer runs and profiles the program executable. The same `-pg` flag used for compilation alters linking to link the executable with the vProf profiling library. At the start of program execution, the library reads the generated schema into memory and sets periodic alarms, using the `profil` system call. At each alarm signal, vProf collects the PC and value samples, the latter by using the schema to determine which variables are accessible at the current PC and where to read their values. vProf also performs bounded virtual stack unwinding to record value samples in the callers of the current function. The developer is expected to profile the program at least twice using vProf, one to produce a profile of a normal execution and another to produce a profile of a buggy execution. Obtaining a normal execution is usually not difficult, as it often only requires executing the program with a smaller workload or less complex command. For example, in Figure 1, variable `recv_n_pool_free_frames` will have some constant value for each execution of the program, but the value will be different for a normal versus buggy execution. Similarly, variable `available_mem` will have some nonzero value for a normal execution, but be zero for a buggy execution.

Finally, the developer runs the vProf post-analysis tool, using the normal execution profile of the program as a baseline to compare against the buggy execution profile. PC samples are used to determine the execution cost of each function. If the alarm interval is t and the PCs that lie in the address range of function f are sampled n times during the profiling session, then the execution cost of f is calculated as $t \times n$. Value samples are grouped based on the functions where they occur and used to calculate a variable-based execution cost and a discount ratio to adjust the cost of each function. The discount ratio is based on a comparison of the value samples from the normal and buggy profiles, with larger discounts for more similar value distributions between the profiles. vProf automatically classifies bug patterns based on the value samples and identifies where anomalous value samples occur to pinpoint suspicious basic blocks. For example, in Figure 1, function `recv_group_scan_log_recs` will be assigned a variable-based execution cost and have no discount to its execution cost because of the presence of anomalous values for its variables `recv_n_pool_free_frames` and `available_mem`. On the other hand, function `recv_apply_hashed_log_recs` will have a substantial discount to its execution cost. The end result is that

vProf will rank the former ahead of the latter, alerting the developer to the correct root cause of the performance issue.

3 Schema Generator

To enable value-assisted cost profiling, we need to decide what variables to monitor during profiling. If a variable key to a performance issue is not monitored, vProf's effectiveness will become similar to conventional profiling. To address this challenge, we use program analysis to systematically identify the types of variables that commonly influence performance. Then, we make value recording efficient enough to allow vProf to sample many variables.

3.1 Source Code Static Analysis

vProf leverages LLVM to automatically identify the variables to monitor. For C/C++ programs, it uses the widely used Clang compiler frontend to parse the target program source code into LLVM's language-independent intermediate representation (IR). For each program source file, LLVM IR provides a call graph for all functions in the file. vProf introduces a simple LLVM analysis pass to traverse the call graph and identify where the variables of interest are defined. vProf identifies variables that are important to reason about performance bugs, specifically global variables and local variables from loops, branches, and function calls. vProf monitors all global variables in part because most programs contain only a relatively small number of them and they are accessible from any execution context, making them easy to monitor with low overhead. vProf is more selective with local variables, since monitoring all of them would be too costly. For loops, vProf monitors the induction variables, which can indicate not only the number of iterations but also timing information. For example, if an induction variable's sampled values are 3,6,6,6,6,9 in the buggy profile and 3,6,8 in the normal profile, it could indicate a performance issue caused by a missing skipping or breaking condition inside the loop, because the iteration 6 lasts for a much longer time in the buggy profile. For branches, vProf monitors all variables in a conditional expression. For call instructions, vProf monitors all variables used as call parameters.

vProf typically affords the ability to monitor thousands of variables, which can include all relevant variables for small programs. For large programs, to reduce overhead, developers can limit the variables to monitor to specific components of the program related to a performance issue, e.g., the buffer pool component in MariaDB whose source code locates in `storage/innobase/buf`. vProf will then only extract variables in source files of the specific component. If the restricted value recording does not reveal the performance bug, developers can iteratively choose another component to monitor.

The analysis pass returns a schema showing where each variable being monitored is defined in the source code. Each variable is a schema entry in the following format:

```
file_path, function, line, variable, type, tags
```

`file_path` is the file path of the source code file that contains the variable definition. `function` is the name of the function that contains the variable definition if it is a local variable or the keyword `#global` if it is a global variable. `line` is the line number of the source code file where the variable definition is located. `variable` is the name of the variable. `type` is the type of the variable. `tags` is a set of vProf-specific tags that indicate how the variable is used, such as `loop`, `branch`, and `args`. For example, vProf monitors the variables `recv_n_pool_free_frames` and `available_mem` in Figure 1, which are represented in the schema shown in Figure 3. `recv_n_pool_free_frames` has `tags` equal to `none` since it is not used in any loop induction variables, branch conditional expressions, or call parameters. `available_mem` has `tags` equal to `cond|args` since it is used in conditional expressions and call parameters.

3.2 Binary Static Analysis

vProf transforms the schema to automatically identify the runtime locations of variables to monitor, which we refer to as *variable metadata*. Once the developer compiles the target program with the `-pg` flag, the program executable contains DWARF debugging information. vProf simply uses a DWARF parsing library [8] to search the debugging information to retrieve the scope and location information for each variable in the schema. vProf outputs a new schema of variable metadata, where each entry represents a contiguous range of PCs in which the variable can be accessed. Each entry of variable metadata is in following format:

```
pc_start:pc_end:location:offset:size:basic_type_ptr
```

`pc_start` to `pc_end` is the range of PCs for which the entry is valid. `location` indicates the location in which the variable can be accessed, such as a register. `offset` is either the offset at which to access a variable in a register or the address at which to access the variable in memory. `size` is the size of the variable. `basic_type_ptr` is a flag to indicate whether the variable is a pointer to a basic type, such as a `char` or `int`, in which case vProf can dereference the pointer to obtain the actual value that is stored. vProf may generate multiple entries of variable metadata for each variable.

For example, Figure 3 shows some of the metadata entries generated for the variables in Figure 1. The entry for `recv_n_pool_free_frames` indicates it is accessible in memory at address 21316200, 8 bytes in size, and not a basic type pointer. The entries for `available_mem` indicate that it is accessible in register `rbx`, 8 bytes in size, and not a basic type pointer. Its offset is zero as it uses all bits of the 64-bit register.

DWARF debugging information may be incomplete, in that a variable may be accessible at a given PC but the information is not captured in the debugging information. For example, the entries for `available_mem` in Figure 3 cover two separate PC ranges in the function `recv_group_scan_log_recs`. The

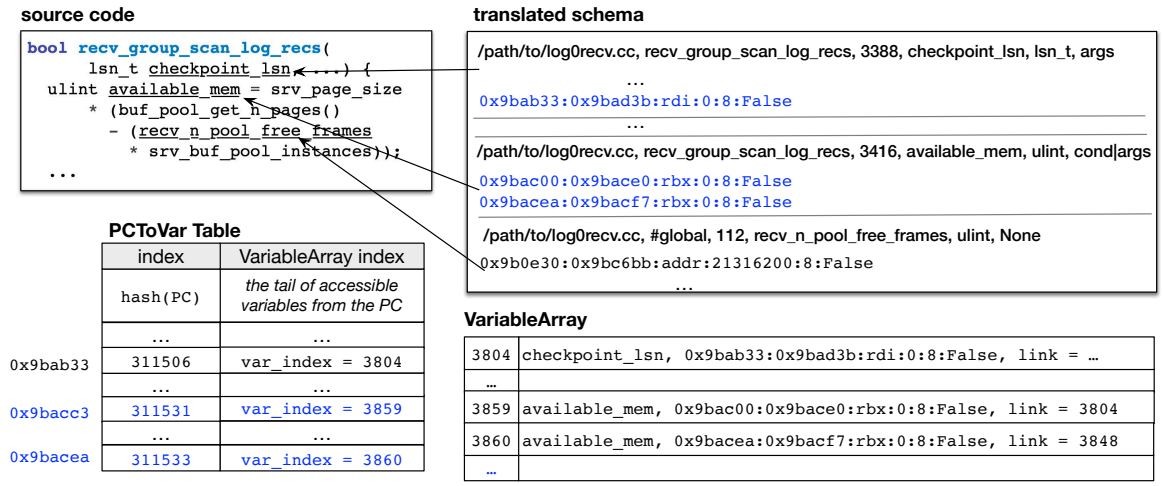


Figure 3. vProf generates variable metadata and initializes profiler data structures from schema for the example in Figure 1. Highlighted entries indicate overlap in PC ranges with other variables.

first entry includes the variable definition and the second entry includes its use in the conditional expression. However, there is a gap between them, likely because `available_mem` is pushed onto the stack due to the call to `recv_parse_log_recs`, and thus no longer accessible in a register. Efficiently determining the exact address on the stack from which to read such variables is a challenge. For simplicity, vProf assumes that a variable is not accessible at a given PC if there is no explicit DWARF debugging information that includes the PC to indicate its runtime location.

3.3 Profiler Intialization

Since profiling is done using PC sampling, we want an efficient mechanism to determine what value samples to record at a given PC. vProf accomplishes this by transforming the variable metadata into a more efficient representation used for profiling. vProf introduces two data structures in the profiler, a PC hash table, `PCToVarTable`, and an array for the variable metadata, `VariableArray`, shown in the example in Figure 3. The data structures are connected via a `var_index` field in each entry of `PCToVarTable` and a `link` field in each entry of `VariableArray`. By default, `PCToVarTable` is allocated to be half the size of the text section of the program being profiled.

Before executing the program to be profiled, vProf reads the variable metadata from a file. For each metadata entry, vProf allocates an entry in `VariableArray` for the metadata and hashes each PC in the range of the metadata to an entry in `PCToVarTable`, which it fills in. For example, Figure 3 shows that the variable `checkpoint_lsn` is accessible starting at PC value 0x9bab33. vProf allocates the `VariableArray` entry at index 3804 to `checkpoint_lsn`, and fills in multiple `PCToVarTable` entries, including 311506 for PC 0x9bab33, whose `var_index` is set to 3804. Collisions from hashing different PCs to the same element of `PCToVarTable` are handled using separate chaining.

Multiple variables may be accessible at a given PC. If vProf finds an entry in `PCToVarTable` already filled in for a given PC, that means that some other variable metadata entry has an overlapping PC range with the one currently being processed. If the entry in `PCToVarTable` is already filled, vProf saves the `var_index` from `PCToVarTable` to the `link` field of the current `VariableArray` entry for the variable metadata currently being processed. It then updates the `PCToVarTable` entry with the index of the current `VariableArray` entry. In this way, multiple `VariableArray` entries are chained together to a related `PCToVarTable` entry.

For example, in Figure 3, the `var_index` of `PCToVarTable` entry 311531 for PC 0x9bacc3 stores the index 3804 for the `checkpoint_lsn` `VariableArray` entry since for PC 0x9bacc3 falls within the PC range for `checkpoint_lsn`. When processing the variable metadata for `available_mem`, PC 0x9bacc3 also falls within the PC range. The `link` field of the `available_mem` `VariableArray` entry is thus set to 3804. The `var_index` of `PCToVarTable` entry 311531 is then updated to the index 3859 for the `available_mem` `VariableArray` entry.

Note that Figure 3 shows the state of `PCToVarTable` and `VariableArray` before processing the variable metadata for `recv_n_pool_free_frames`, a global variable that is accessible at all PCs shown in `PCToVarTable`. For example, after that variable metadata is processed, the `var_index` of `PCToVarTable` entry 311531 will be updated to the index for a `VariableArray` entry for `recv_n_pool_free_frames`, which in turn will have its `link` set to 3859.

After this process, the metadata of all variables is stored in `VariableArray` and accessible by PC from `PCToVarTable`. vProf also stores the mapping from the schema to `VariableArray` in a `Layout Log`, which is used later for post-profiling analysis.

4 Value Sample Recording

vProf's program analysis and data structure design make it straightforward to efficiently record value samples during profiling. vProf uses `PCToVarTable` and `VariableArray` together with a `SampleArray` to store value samples. When the alarm

fires and the PC is sampled, vProf reads all *accessible* variables according to the metadata. It looks up the sampled PC in `PCToVarTable` and follows its `var_index` and subsequent `link` fields in the chain of `VariableArray` entries. For each `VariableArray` entry in the chain, vProf checks that the sampled PC falls within its PC range, in which case it accesses the variable value and stores it, as well as the sampled PC, to a new `SampleArray` entry.

For example, when profiling the program shown in Figure 3, if the alarm fires and the PC sampled is 0x9bacc3, vProf will look up the `PCToVarTable` and follow its `var_index`. We assume for this example that the `PCToVarTable` and `VariableArray` have been updated to include the variable metadata for the global variable `recv_n_pool_free_frames`. Thus, `var_index` will be the index to a `recv_n_pool_free_frames` `VariableArray` entry. vProf will record the `recv_n_pool_free_frames` value in a new `SampleArray` entry. vProf will then follow the `link` to `VariableArray` entry 3859 and record the `available_mem` value in a new `SampleArray` entry. vProf will then follow the `link` to `VariableArray` entry 3804 and record the `checkpoint_lsn` in a new `SampleArray` entry.

Checking that the sampled PC falls within the variable metadata's PC range is necessary as it is possible for this not to be true due to the manner in which `VariableArray` entries are linked together when their PC ranges overlap, especially since the property is not transitive. Since most variables are local with limited PC ranges only accessible within their respective functions, we do not expect to encounter many `VariableArray` entries linked to a `PCToVarTable` entry which are not accessible.

`SampleArray` entries are chained together with their corresponding `VariableArray` entry. Each `SampleArray` entry has a `link` field. Each `VariableArray` entry has a `sample_tail` field, which is used to record the index of the most recently recorded `SampleArray` entry for that variable. When a value is stored to a new `SampleArray` entry, its `link` is set to the `sample_tail` from the respective `VariableArray` entry, and the `sample_tail` is updated to the index of the new `SampleArray` entry.

vProf's passive value recording approach relies on having PC samples occurring within the PC range of the variables being monitored. For functions that do not run much, vProf may not get enough value samples. This can be an issue especially for callers with time consuming callees. For example, in Figure 1, the root cause function `recv_group_scan_log_recs` calls the costly function `recv_parse_log_recs`, so vProf almost always only observes PCs from `recv_parse_log_recs` when it samples the PC. Thus, vProf has few samples for local variables like `end_lsn` and `available_mem` in the root cause function, which are not accessible in the PC range of `recv_parse_log_recs` based on the DWARF debugging information available. A related shortcoming of `gprof`, on which vProf is based, is that when a target program calls into a dynamic library, `gprof` does not record PC samples since they are outside the range of the target program.

To address this issue, vProf introduces virtual stack unwinding. For each sampled PC, it unwinds the call stack by a bounded depth (default 3) and records variables accessible at the caller PC, which is PC before the `call` instruction. Specifically, we restore the registers in each step and begin the value sampling using the caller PC. We also add a field `stack_depth` in the `SampleArray` entry to indicate how many stack layers are unwound before the sample is recorded. The stack frames are restored to their normal state before virtual unwinding at the end of the sampling. Virtual stack unwinding allows vProf to obtain many more value samples to improve the fidelity of profiling. For example, in Figure 1, virtual stack unwinding results in value samples for `recv_n_pool_free_frames` and `available_mem` in `recv_group_scan_log_recs` even when the PC sampled occurs in `recv_apply_hashed_log_recs`. Note that virtual stack unwinding will generate no additional samples if there are no variables of interest accessible at the caller PCs.

vProf dumps the profiling data to disk at program exit. It saves PC samples and variable samples separately. The samples are then processed as part of post-profiling analysis.

5 Post-profiling Analysis

After value sample recording, vProf analyzes the data files from both normal and buggy executions. The data files include the PC samples, which `gprof` refers to as the PC cost histogram, value samples, and layout mapping used to connect value samples to variable information. vProf performs two post-profiling analyses. Cost calibration computes raw execution costs and then adjusts them based on anomalous value samples to promote suspicious functions in a function cost ranking. Bug pattern inference infers potential root cause patterns to help developers narrow down the root cause.

5.1 Cost Calibration

Traditional profilers only rank functions based on their raw cost, where a function may be ranked high due to unavoidably costly operations, while the real culprit of a performance issue is lower in the raw cost rank. vProf calibrates the cost of functions by increasing the cost of functions that contain many variables of interest, and decreasing the cost of functions whose variables are not anomalous.

vProf increases the cost of functions with variables of interest by computing an alternative execution cost based on the frequency of value samples, which we refer to as the variable-based execution cost. The standard approach to determine the execution cost of a function using PC sampling is to count the number of PC samples that lie in the PC range of the function and multiply it by the alarm interval. Instead of counting PC samples, vProf determines the variable-based execution cost by counting the number of value samples with distinct PCs that lie in the PC range of the function and multiplying it by the alarm interval. Multiple value samples at the same PC are

counted as one sample. vProf then uses the maximum of the two costs as the raw execution cost of the function.

The variable-based execution cost will be higher than the standard execution cost if the number of value samples with distinct PCs in a function is higher than the number of PC samples. This can occur especially due to virtual stack unwinding if some variables being monitored are accessible within the function, and the function calls some other function with higher execution cost. The idea is to use the higher variable-based execution cost as the function has variables of interest which could be related to a performance issue. For example, `recv_group_scan_log_recs` has a higher variable-based execution cost than its standard execution cost since variables being monitored such as `available_mem` are accessible within the function and it calls `recv_apply_hashed_log_recs`. This will result in it having many more value samples than its own PC samples because the value samples will occur at the frequency of the PC samples of its more time consuming callee due to virtual stack unwinding.

vProf decreases the cost of functions whose variables are not anomalous by introducing a *variable-discounter*, which is vProf's main cost calibration mechanism. It computes a discount ratio for each sampled variable based on how anomalous are its samples. The less anomalous the samples are, the greater the discount ratio, meaning that the variable is unlikely to be contributing to the performance issue. Discount ratios for variables are aggregated to the functions in which they are accessible to compute a discount ratio for each function. The cost of a function is calculated by multiplying its raw execution cost and one minus the discount ratio, which is between zero and one. As a result, a greater discount ratio (less anomalous samples) results in a greater decrease in the calibrated execution cost, so that the respective function will be less likely to be considered in diagnosing a performance issue.

We first describe how vProf determines how anomalous are a variable's samples and computes a discount ratio. The idea is to compare the value samples collected from the normal execution versus those collected from the buggy execution. vProf defines samples as anomalous based on how different the sample distributions are between the normal and buggy executions. The idea is to consider distributions to be different if they have different shapes. For example, if two distributions with the same normal distribution shape will be considered the same even if their means are different, but a normal and uniform distribution will be considered different.

Specifically, given the null hypothesis that the distributions are identical, vProf applies the k-sample Anderson-Darling test [2] to the distributions to determine if the null hypothesis holds with some probability. By default, vProf uses a probability of 0.05. This means that vProf assumes the distributions are the same by default unless it can determine with high (95%) confidence that they are different. If the null hypothesis holds, vProf sets a discount ratio of DefaultDiscount for the variable, which is 0.8 by default. If the null hypothesis is rejected, vProf calculates the Hellinger distance [34], a measure of how different the distributions are. Its value is between 0 and 1, where a larger value indicates greater difference. The discount ratio for the variable is set to one minus the Hellinger distance, unless it is below a ValidDiscount threshold, in which case the ratio is zero. ValidDiscount is 0.1 by default.

Assuming the variable is a basic type, vProf considers the degree of anomaly in a variable along three dimensions. First, it considers values, as previously described. Second, it considers deltas of values in adjacent samples. This quantifies how much the values change. Third, it considers processing costs of values, specifically how many alarm intervals a variable value stays the same. This quantifies how often the values change. vProf determines the discount ratio for a variable in each of the three dimensions, and uses the lowest of the discount ratios. For pointers to non-basic types, vProf only uses the discount ratio based on processing costs, since the differences in pointer values, meaning differences in addresses, is not generally a useful distinction between normal and buggy executions.

We next describe how we aggregate discount ratios for variables to functions. For local variables, their discount ratios are attributed to the function in which they are defined. For global variables, their discount ratios are attributed to the functions which contain recorded PCs at which the variable was sampled. When a function has multiple associated variables with different discount ratios, vProf uses the lowest discount ratio among them, because the most anomalous variable often suggests the function is worthy of further examination. For each function, if its raw execution cost is x and its discount ratio is r, its calibrated cost is $(1-r)\times x$. By using a DefaultDiscount of 0.8, vProf can significantly demote costly functions without anomalous value samples, but avoid eliminating them entirely. By using a ValidDiscount of 0.1, vProf can preserve the ordering of functions by cost for functions with similarly low discount ratios, as value samples may be noisy. Section 6.4 evaluates how sensitive vProf is to these defaults.

For large programs, the variables being monitored may be limited to functions located in certain program components, resulting in no discount ratio being available for functions outside of those program components. To derive a discount ratio for these functions as well, vProf includes a simple *hist-discounter*, which computes a discount ratio by comparing how the function ranks in terms of raw execution cost between normal and buggy executions. Because of potential variability in the rankings, the hist-discounter is based on profiling the program multiple times. Given n buggy profile(s) and m normal profile(s), we perform a cross-comparison among the two groups for each function. We maintain a counter h for each function to record in how many comparisons this function ranks higher in the normal profile(s) than in the buggy profile(s). We also record c ($c \leq n \times m$) as the number of comparisons for the function. Then we set the discount ratio to $r = \frac{h}{c}$. The ValidDiscount threshold is also used with hist-discounter

```
void ap_mpm_pod_killpg(ap_pod_t *pod, int num) {
  for (i = 0; i < num && rv == APR_SUCCESS; i++) {
+   if (ap_image->servers[i].status != SERVER_READY ||
+     ap_image->servers[i].pid == 0)
+     continue;
    rv = dummy_connection(pod);
  }
}
```

Figure 4. Root cause for performance issue HTTPD-54852. When using the Multi-Processing Module (MPM), the graceful restart of Apache httpd can sometimes take a few minutes. The problem is the `dummy_connection` call becomes much slower due to polling if all the children have already exited. Developers fixed the bug by adding a check in the loop to skip unnecessary `dummy_connection` calls when there are no more children.

to avoid reordering the rankings of functions with similar low discount ratios. The hist-discounter is only used for functions which otherwise would have no discount ratio available.

5.2 Bug Pattern Inference

Since providing a high-level characterization of potential root cause patterns can further ease performance debugging, vProf provides a root cause classifier to infer potential root cause patterns for top-ranked functions based on their calibrated costs. We observe three common performance bug patterns:

1. *Wrong constraint*: These bugs cause the program execution to unnecessarily fall into a costly path. They often happen when a conditional expression or its evaluation is incorrect. For example, Figure 1 shows the while loop condition is evaluated with an incorrect `available_mem`.
2. *Missing constraint*: These bugs occur when the code performs some operations uniformly instead of discriminating based on some constraint, such as a conditional expression. For example, Figure 4 shows such a bug in Apache fixed by adding a conditional expression.
3. *Scalability*: These bugs usually arise when the program processes data larger than the developers expected, such as traversing a large list in a critical section. For example, Figure 5 shows such a bug in MariaDB.

To infer the bug pattern for each function, the classifier queries the variable-discounter for information about which sampled variable was most anomalous. Specifically, for each function, it finds the anomalous sampled variable with the minimum discount ratio and the dimension used in calculating that ratio. Then, it obtains the the variable's abnormal samples from the buggy execution. The variable-discounter provides this by computing a variable's normal range from the normal execution and identifying the value samples in the buggy execution that are out of the normal range. Since each value sample contains the PC at which it was recorded, the classifier uses the DWARF information to map the PC back to the text section to localize the code region for abnormal samples and get the basic block label and control flow structures.

The classifier then checks how an anomalous variable is used in the code region based on its tags, as discussed in Section 3. With the discount ratio, dimension, and tags, the

```
bool buf_LRU_scan_and_free_block(bool scan_all) {
  ulint scanned = 0;
  for (bpage = buf_pool.lru_itr.start(); bpage && scan_all;
      ++scanned, bpage = buf_pool.lru_itr.get())
    ...                    the LRU list search was slow, scanned=134468
}
buf_block_t* buf_LRU_get_free_block() {
loop:
  mutex_enter(&buf_pool.mutex);
  block = buf_LRU_get_free_only();
  ...
  if (n_iterations || buf_pool.try_LRU_scan)
    freed = buf_LRU_scan_and_free_block(n_iterations > 0);
  ...
  mutex_exit(&buf_pool.mutex);
  n_iterations++;                  scan the whole LRU list
  goto loop;                       when n_iterations > 0
}
```

Figure 5. Root cause for performance issue MariaDB-23399. Under I/O-bound TPCC workloads, MariaDB throughput gradually decreases and is worse than a previous version. The problem is when the buffer pool is full, `get_free_block` calls `buf_LRU_scan_and_free_block` to do a linear scan of 1.6 million buffer pool blocks. The thread holds the `buf_pool.mutex`, preventing other threads stopping the scan by releasing pages to the buffer pool.

classifier infers the bug patterns by using the following rules in order:

1. If some loop induction or conditional expression variable stays the same for an abnormally long time, which is identified as a variable with a `loop` or `cond` tag and anomalous samples based on a discount dimension of processing cost, the function is labeled with a *Missing Constraint* bug.
2. If some loop induction variable has abnormal values, which is identified as a variable with a `loop` tag and anomalous samples based on a discount dimension of value or delta of the value, the function is labeled with a *Scalability* bug.
3. If a conditional expression variable is abnormal, which is identified as a variable with a `cond` tag and anomalous samples, the function is labeled with a *Wrong Constraint* bug.
4. If the most costly function is normal and has no variables of basic types being sampled, meaning it has a Default-Discount and discount dimension of processing cost, the function is labeled with a *Scalability* bug. Without values of basic types, vProf does not have enough information to identify other bug patterns in this case.

6 Implementation and Evaluation

We implemented vProf for C/C++ programs, mostly by modifying `gprof`, though vProf is compatible with any profiler based on PC sampling. This involved changes to `glibc`, mainly in `gmon.c` and `profil.c`. We modified `gmon.c` to set up the in-memory profiling schema metadata on initialization, which is called from `__monstartup`. We modified `profil.c` to collect value samples. We extended the profiler signal handler to read values of variables accessible from the current PC. We implemented virtual stack unwinding using the `libunwind` library [31]. We fixed issues in `gprof` to better support multiple-process programs, such as renaming the `gmon.out` file with the process id, setting profiling timers for child processes, and unblocking `SIGPROF` signals. We implemented the schema

generator using an LLVM analysis pass and a Python library. We implemented the post-profiling analysis in Python.

We evaluated vProf in diagnosing performance issues in widely used applications. We performed a comparative study against other state-of-the-art solutions on previously diagnosed performance issues to quantify effectiveness. We further used vProf to diagnose several previously unresolved performance issues in widely used applications. We also quantify vProf's performance overhead. All measurements were done on a desktop computer with a 6-core (12 hyper threads) Intel 2.60 GHz Core i5 CPU and 48 GB DRAM, running Ubuntu-20.04 with Linux kernel 5.11.0.

To collect bugs for evaluation, we considered four large applications: MariaDB [30], Apache HTTPD [3], Redis [37], and PostgreSQL [36]. We queried their official issue trackers using keywords *slow* and *performance*, randomly selected from among the issues, read their reports, and included the issues if they were truly performance-related and the reports had sufficient information for bug reproduction. We then excluded bugs that developers found from just reading source code as such bugs typically do not impact real users. In total, we collected 26 issue tickets. Three of the issues could not be reproduced by following the reports. Five of the issues were database-related and could be resolved by simply comparing the SQL explanations in the normal and buggy cases. Our evaluation focused on the remaining 18 out of the 26 issues, including 15 resolved issues, listed in Table 1, and three unresolved issues, discussed in Section 6.2.

ID	Description	Bug Pattern
b1: MDEV-21826	Server crash recovery loops on the same log sequence number (LSN) forever	Wrong Constraint
b2: MDEV-23399	Performance drops when the size of data set is larger than the size of buffer pool	Scalability
b3: MDEV-13498	Deleting a table with CASCADE constraint is very slow	Missing Constraint
b4: MDEV-15333	Slow start-up even when `.ibd` file validation is off	Wrong Constraint
b5: MDEV-17933	Checking the server status takes >10 seconds with 3M tables	Scalability
b6: HTTPD-62668	Output filter endless loop so server process never terminates	Missing Constraint
b7: HTTPD-54852	Gracefully restart service with MPM workers takes long time	Missing Constraint
b8: HTTPD-62318	Health check is executed more often than configured interval	Wrong Constraint
b9: HTTPD-64066	Slow startup/reload when many `vhosts` are configured	Scalability
b10: HTTPD-52914	Workers eat 60-100% CPU even though no client sent requests	Wrong Constraint
b11: Redis-8145	`cluster nodes` command is costly in a large cluster	Scalability
b12: Redis-8668	`BRPOP` becomes slow when a large number of clients exist	Missing Constraint
b13: Redis-10310	`ZREVRANGE` command 50% slower after upgrade	Missing Constraint
b14:Postgres-17330	EXPLAIN query hangs for some query plans	Scalability
b15:Postgres-14b1	vacuum process fails to prune all heap pages and endlessly retries	Wrong Constraint

Table 1. Reproduced real-world performance issues.

6.1 Comparative Study

We used the bugs in Table 1 to evaluate the effectiveness of vProf versus other widely used and state-of-the-art tools in diagnosing performance issues in widely used applications. The other tools we tried were `gprof`, `perf`, `perf` with an enhancement using Intel Processor Trace (`perf`-PT), COZ [12], and statistical debugging [40] (stat-debug). Table 2 briefly describes each tool and how it was configured; similar configurations were used whenever possible.

Several of the tools, `perf`-PT, statistical debugging, and vProf, required profiling normal execution in addition to the buggy execution. Normal executions were obtained for MariaDB-21826 and Redis-10310 by running the same command on a different version. Normal executions for all other issues were mostly obtained by using smaller inputs on the same software version. Specifically, we reduced the number of tables in the database for MariaDB, the number of virtual hosts in Apache httpd, and the number of nodes in a cluster for Redis. For example, in MDEV-13498, we deployed a database with the test script provided by the user in the bug report. Deleting the first table took 20 minutes, which exposed the symptom. Deleting a second table from the same script took 2 minutes, which we used as the normal execution. We simply reran the same command with the same inputs multiple times if multiple profiling runs were needed.

Because the applications are large, several of the tools require some identification of the component in which the performance issue occurs, to limit overhead. For `perf`-PT, we only performed its control-flow profiling on the top ten most costly functions by using the Intel Processor Trace address filter feature to limit the size and decoding time of the resulting branch traces. For COZ, we identified the top-level function in the source code file that contains the performance issue to limit runtime since it can otherwise take many hours to run as it randomly picks source code lines to virtually speedup to measure potential performance improvement. For statistical debugging and vProf, we identified the source code file that contains the performance issue to limit the predicates and variables sampled, respectively.

For each issue, we ran each tool on a buggy execution that reproduced the issue based on descriptions in the bug reports. We then measured how the tool ranked the root cause function in its output; lower number rank is better. The best result is for a tool to rank the root cause function first, meaning the tool pinpoints the function that causes the performance issue. Table 3 lists the results. vProf outperforms all other tools, ranking the root cause function within the top five (2nd on average) in all 15 cases. In comparison, `gprof`, `perf`, `perf`-PT, COZ, and statistical debugging ranked the root cause function

Name	Description and Configuration
`gprof`	Version 2.34 with glibc-2.31, default options used.
`perf`	Version 5.11.22, default options used.
`perf`-PT	`perf` with top-10 functions re-ranked using control-flow profiling: profile normal and buggy executions, Intel Processor Trace counts branches taken, calculate difference in branches taken per function for normal versus buggy executions, and use ratio of difference over total branches to scale top-10 function cost.
COZ	Determines which basic block if optimized further will improve overall performance the most; user identifies which functions to consider by identifying file that contains root cause function and top-level function in that file that will eventually call root cause function.
stat-debug	Records values of predicates, namely conditional statements and return values of functions, then ranks functions based only on how different the predicate distributions are between normal and buggy executions; user identifies file that contains root cause function and predicates only considered for functions in that file, 5 normal and 5 buggy executions used.
vProf	User identifies file that contains root cause function to limit number of variables sampled to that file, 5 normal and 5 buggy executions used for hist-discounter, but only one of each was used for variable-discounter.

Table 2. Configurations of tools to diagnose performance issues.

ID	vProf			Other Tools					
	rank	bb-dist	class.	gprof	perf	perf-PT	COZ	stat-debug	hist-disc
b1	1st	5, 0	✓	454th	32nd	32nd	NR	4th	447th
b2	1st	7, 0	✓	5th	2nd	2nd	NR	12th	1st
b3	1st	n/a	✓	2nd	3rd	6th	1st	30th	177th
b4	3rd	9, 0	✓	21st	9th	5th	NR	18th	31st
b5	4th	0, 0	✓	13th	4th	9th	NR	566th	22nd
b6	5th	19, 0	✓	36th	13th	13th	NR	NR	15th
b7	3rd	0, 0	✓	182nd	1024th	1024th	crash	7th	181st
b8	1st	0, 0	✓	1st	6th	7th	child	3rd	6th
b9	2nd	21, 0	✓	11th	28th	28th	NR	9th	11th
b10	1st	0, 0	✓	4th	16th	16th	child	161st	4th
b11	1st	0, 0	✓	1st	10th	10th	2nd	NR	59th
b12	1st	7, 5	✓	5th	19th	19th	1st	8th	2nd
b13	2nd	0, 0	NC	16th	13th	13th	9th	NR	33rd
b14	4th	17, 0	✓	NR	163rd	163rd	child	13th	NR
b15	3rd	2, 0	NC	14th	56th	56th	child	18th	8th

Table 3. Diagnosis effectiveness of tools. NR denotes the root cause function was not ranked, crash denotes the tool crashed, and child denotes the tool failed diagnosis because the root cause function was run in a child process. For vProf, *bb-dist* shows the (mean, minimum) distance between the basic block vProf identified and the root cause, and *class* shows whether the bug pattern reported matched the root cause; NC denotes the root cause could not be classified.

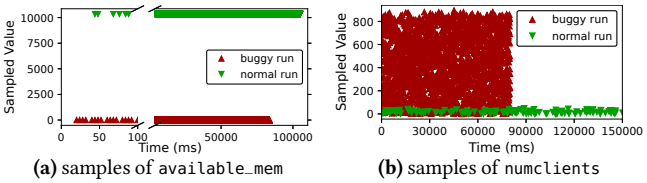
within the top five in only six, three, two, three, and two cases, respectively. In fact, vProf ranked the root cause function first in seven cases, more cases than the less precise top-five results for all of the other tools. In comparison, none of the other tools ranked the root cause function first in any of the cases, with the exception of `gprof` which did so for only two cases. Of all the tools, COZ performed the worst, failing to rank the root cause function in 11 cases, of which one was due to the tool crashing and four were due to its inability to support multiprocess applications.

For comparison purposes, Table 3 also shows the result when using vProf with zero variables monitored and only its hist-discounter (hist-disc), discussed in Section 5.1. hist-discounter alone reports the root cause function within top five for only three cases. This demonstrates the key vProf mechanism is not just comparing normal and buggy profiles, but doing so using variable value information, in conjunction with cost discounting using variable value information. Note that the hist-discounter is still useful for large applications in which variables are only monitored in some components. For example, without hist-discounter, vProf has worse results for four cases, causing the ranking of the root cause function to drop from first to third in one case and dropping it out of the top five in two cases. Even without using hist-discounter for components without any monitored variables, vProf still far outperforms all other tools.

This observation that values are important for profiling is reinforced in comparing the results with vProf versus other tools such as statistical debugging or `perf`-PT. Statistical debugging also compares normal and buggy profiles, but uses only predicates, which may be noisy, without accounting for the actual function execution costs. Furthermore, statistical debugging requires the monitored predicates to be observed many times in both normal and buggy executions. In contrast, vProf uses variable value samples and conventional function execution costs, correlating them together with its analysis. Similarly, `perf`-PT compares normal and buggy profiles, but by monitoring control flow based on branch information as an alternative idea. Modern applications have abundant branches and many sources of non-determinism, so their control flow traces are noisy. In general, a performance issue may not be visible in control flow. For example, a performance bug that causes a loop to iterate many more times likely shows the same control flow as a normal execution. In fact, `perf`-PT, which enhances `perf` with control flow profiling, shows no overall improvement over just `perf`.

Table 3 also shows how effective vProf is in identifying the specific root cause basic block. Since vProf may report multiple basic blocks, we calculate the mean and minimum distance between the basic block reported by vProf and the one in which the developers fixed the bug. Shorter distances generally make diagnosis easier. Table 3 shows that in six cases, the basic block vProf reports in the root cause function is exactly where developers fixed the bug. For MDEV-13498, vProf did not report a basic block because DWARF did not provide sufficient information to map a PC sample of an anomalous value sample to basic blocks.

Furthermore, Table 3 shows how effective vProf is in classifying bugs using its bug patterns. vProf infers correct bug patterns for 13 out of 15 cases. It misses the bug pattern in Redis-10310 because the identified variable invokes a function pointer and has no labels. Similarly, it misses the bug pattern



(a) samples of `available_mem` **(b)** samples of `numclients`

Figure 6. Value samples for a variable for two performance issues.

in Postgres-14b1 because of missing information on a variable that is stored inside a class pointer.

Case Studies. We describe two cases in further detail, focusing on how vProf compares to `gprof`, the tool on which it is based. Other case details are omitted due to space constraints.

MDEV-21826: This is the example in Figure 1. `gprof` ranks `recv_apply_hashed_log_recs` first, while the actual root cause function `recv_group_scan_log_recs` ranks 454th. vProf ranks the root cause function first, promoting it based on its monitored variables `available_mem` and `pool_free_frames` using the variable-based execution cost, and demoting 44 other functions based on its variable-discounter. vProf assigns a zero discount ratio to `recv_group_scan_log_recs` as its value samples are quite different between the normal and buggy executions, as shown in Figure 6a. vProf calculates high discount ratios for many other functions. For example, variables such as `end_lsn` have no significant differences in their distributions between normal and buggy executions, discounting the cost of `recv_apply_hashed_log_recs`. Furthermore, vProf translates the PC of the anomalous variable sample into lines and corresponding basic blocks. One of the line numbers is right before the while loop in `recv_group_scan_log_recs`. The basic block distance is zero.

Redis-8668: `gprof` ranks functions from the `zmalloc_*` family and `dictEncObjKeyCompare` above the root cause function `serveClientsBlockedOnKey` which is ranked fifth. vProf ranks the root cause function first, demoting other functions based on its hist-discounter and keeping the root cause function highly ranked based on its variable-discounter. vProf finds the `zmalloc_*` are inherently costly in both normal and buggy executions, have no variables being monitored, so its hist-discounter assigns a discount ratio of 1.0 to them. For similar reasons, `dictEncObjKeyCompare` is assigned a discount ratio of 0.76. vProf assigns a zero discount ratio to the root cause function as its variable samples for `numclients` are quite different between the normal and buggy executions, especially in terms of processing costs. Specifically, Figure 6b shows that the distribution of the value samples in normal versus buggy executions are different, but this results in a discount ratio of 0.12. Instead, the distributions based on processing costs are even more different, resulting in a discount ratio of zero, which vProf uses since it is the smaller of the two. Furthermore, vProf translates the PC of the anomalous variable sample into lines and corresponding basic blocks. One of the line numbers falls in the invocation of `listRotateHeadToTail`, which makes up the costly part of a while loop in `serveClientsBlockedOnKey`. The basic block distance to the while loop is five.

False Positives. Like all profilers, vProf cannot guarantee that the root cause function is always ranked first. Fortunately, a performance issue often involves multiple functions, which are also helpful for performance diagnosis. For example, in HTTPD-54852, vProf ranks `dummy_connection` above the root cause function `ap_mpm_mod_killpg`. However, `dummy_connection` is called by the root cause function, so revealing that function in addition to the root cause function can help with performance diagnosis since the root cause function is still highly ranked. This connection is less clear with `gprof`, which ranks the root cause function well outside its top 100 ranked functions.

However, if the top ranked functions are unrelated to a performance issue, they can waste developers' investigation time and are considered false positives. For vProf, we computed the false positive ratio for each issue by counting the number of functions unrelated to the performance issue before the developer reaches the root cause function and dividing that by five. The false positive ratio would be 100% if all top five ranked functions are unrelated to the performance issue. Across all 15 cases, the average false positive ratio was only 10.6%. Given that vProf ranked the root cause function first in almost half the cases and in the top five in all cases, this means that when vProf does not rank the root cause function first, on average at most one other function was ranked ahead of the root cause function that was unrelated to the performance issue.

The false positive ratio does not imply that the developers would necessarily waste time investigating unrelated functions, which depends on the sources of false positives. First, an inherent costly function can be top-ranked even though it has a high discount ratio. For example, in MDEV-17933, vProf ranks the function `ut_delay` first but with a high discount ratio. In such cases, the discount ratio indicates the function is inherently costly in normal and buggy cases, so the developer can consider it lower priority to investigate. Second, some functions are costly as a side effect of a buggy execution. For example, in HTTPD-62668, vProf ranks the function `listener_thread` first because it takes a long time in the buggy case waiting for a request timeout, but it returns immediately in the normal case. Such false positives are hard to eliminate but usually help confirm the causes of performance issues. Third, false positives can also be due to the limitations of statistical methods. Developers can exclude such functions by verifying the annotated bug pattern or increasing the accuracy with repeated experiments.

6.2 Diagnosing Unresolved Issues

We further used vProf on three real unresolved performance issues to demonstrate its effectiveness at diagnosing unknown root causes in practice. These issues are listed in Table 4.

Redis-10981: Developers investigated the issue by bisecting their commits but could not draw a definitive conclusion for the performance degradation in version 7.0.3. In both 7.0.3 and

ID	Description	Date
Redis-10981	`lrange` command takes longer to finish when `redis` is upgrade from version 6.2.7 to 7.0.3	07-14-2022
MDEV-16289	Query runs unexpectedly slow; the query selects records created within a given time period in one table, and excludes records whose certain fields are after a given time by checking another table.	05-25-2018
MDEV-17878	Searching for the query execution plan for a SELECT query involving many joins takes forever for larger datasets, using 100% CPU	11-30-2018

Table 4. Unresolved performance issues diagnosed using vProf.

the earlier version, traditional profilers attribute the highest costs to functions `_addReplyToBuffer` and `addReply`. Comparing the ranking of functions in profiling reports from the two versions does not provide useful information either.

We used vProf to diagnose the performance issue, which had remained unresolved for more than six months. We first investigated the component `db.c`. vProf ranks its function `lookupKey` first. It shows that the variable `key` has different processing costs and sampled values in the buggy version. Looking into the code, we found that function `expireIfNeeded` was moved into `lookupKey`. The code refactoring caused a longer execution time and different values samples, leading to a false positive.

We next investigated the component `networking.c`. Although vProf ranks its function `_addReplyToBufferOrList` first, it is new in 7.0.3 due to code refactoring, so we excluded it from further consideration. vProf ranks the function `clientHasPendingReplies` second as the processing cost for its variable `client` differs in the two versions. vProf reports the anomalous variable samples are accessed in a conditional expression. The condition was introduced in 7.0.3. We verified our findings by reverting this condition, which caused the performance degradation to disappear. vProf successfully identified the unresolved issue that was unable to be clarified previously using the commit-bisecting method or traditional profilers. We reported our findings to the developers, who quickly confirmed the diagnosis.

It took about four-person hours per component to generate schemas for a specified program component, run test cases with vProf, and investigate the source code based on the vProf reports. Since we investigated two components, the total time to diagnose the performance issue was eight person-hours.

MDEV-16289: A developer reproduced the issue and reported that different timezone settings caused different processing costs, identifying it as a performance bug because he believed the query results should be independent of the timezone. In trying to diagnose the issue, the developer traced the query execution plans for two different timezone settings, but the results were similar and provided limited hints for further debugging.

We used vProf to diagnose the performance issue, which had remained unresolved for more than four years. We investigated the component `row0sel.cc`, which implements row selection in MariaDB. The function `row_search_mvcc` was ranked first. Although this function is costly whether or not the query runs slow, its discount ratio is zero because the sample distributions for local variable `clust_index` differ between fast and slow queries. No value samples are captured when the query is fast, but over 30 are captured when the query is slow. We also noticed a similar issue for the variable `result_rec`. Both variables appear to be pointers to temporary storage of intermediate query results.

Because references to additional temporary storage only appear when the query runs slow, we suspected the queries might return different numbers of records for different timezone settings. We verified our hypothesis by changing the query's timestamp to refer to the same absolute time in different timezones. For example, instead of querying with 8pm in all timezones, we queried with 8pm EST and 5pm PST. By doing the latter, the difference in query performance disappeared. We further confirmed our hypothesis by checking the number of records returned; many more records were returned for the slow query case. Contrary to the developer's belief, this issue turned out not to be a performance bug, but correct operation with different query times for what are actually different queries. Diagnosing the issue using vProf took roughly five person-hours. We reported the findings to the developer.

MDEV-17878: The user who reported the issue also profiled the issue using `perf`, which ranks function `prev_record_reads` first. In trying to diagnose the issue, developers obtained query execution plans from both MariaDB and a different version of MySQL that finishes the query quickly. The information obtained did not provide enough hints for the developers to diagnose the performance issue.

We used vProf to diagnose the performance issue, which had remained unresolved for more than four years. We identified the program component involved in optimizing the query execution plan and monitored its variables using vProf. We then needed to profile a useful normal execution, which took us three tries. First, because the report indicates that the performance issue does not occur for small datasets, we created a small dataset to profile a normal execution. However, the query finished too fast and resulted in no value samples being collected. Second, we took the original dataset causing the bug and reduced the number of joins so that the performance issue disappeared. vProf ranked the functions `best_access_path` and `best_extension_by_limited_search` first and second, respectively; the latter calls the former. However, vProf set both their discount ratios to DefaultDiscount, indicating a lack of anomalous value samples.

Finally, because the report was specific to a version of the application, we tried a different version with the original dataset that caused the bug and found that the performance issue disappeared. We used this different version with the original dataset as the normal execution. In this case, vProf ranked the

ID	Variables	Init Time	PCToVar Table	Variable Array	Value Samples	Run Time
b1	233	7.4 ms	3862 KB	430 KB	21133 KB	105 s
b2	65	0.9 ms	4143 KB	29 KB	153 KB	1903 s
b3	399	0.4 ms	4005 KB	26 KB	38563 KB	1140 s
b4	852	15.9 ms	3987 KB	67 KB	58 KB	338 s
b5	577	18.2 ms	3575 KB	22 KB	8 KB	1635 s
b6	501	31.1 ms	673 KB	287 KB	2 KB	1448 s
b7	113	0.3 ms	162 KB	6 KB	16 KB	147 s
b8	169	4.5 ms	260 KB	127 KB	43 KB	553 s
b9	374	6.2 ms	194 KB	16 KB	25 KB	36 s
b10	164	1.4 ms	642 KB	186 KB	13 KB	139 s
b11	531	3.4 ms	612 KB	382 KB	1216 KB	885 s
b12	623	5.5 ms	591 KB	44 KB	1755 KB	112 s
b13	564	7.1 ms	641 KB	754 KB	132 KB	10 s
b14	479	5.2 ms	2037 KB	1031 KB	79 KB	68 s
b15	805	6.4 ms	2297 KB	927 KB	3269 KB	29 s

Table 5. Memory overhead and execution time for profiling performance issues.

function `best_extension_by_limited_search` first. vProf labels it a `Missing Constraint` bug because of anomalous value samples for `use_condition_selectivity`, which is used in a conditional expression. This variable value comes from the system variable `optimizer_use_condition_selectivity` in `sys_vars.cc`, which has different default values for different versions of MariaDB. `use_condition_selectivity` decides the heuristics used to estimate the cost of the current partial query plan. The query plan search algorithm stops if the cost is greater than the current best heuristic. However, if the default value of `optimizer_use_condition_selectivity` is one, the search algorithm fails to stop searching through more costly heuristics to find a better plan.

Diagnosing the issue using vProf took roughly 12 person-hours, eight of which were for going through the three approaches to profile a normal execution, and four of which to investigate the source code. In this and the other cases, the process could be faster for actual developers who are familiar with the program source code. This case also shows how using a different program version can be useful to profile a normal execution. We reported the root cause to developers, who confirmed our diagnosis and updated the issue ticket to include our reported root cause.

6.3 Performance Overhead

We measured the memory and runtime overhead when using vProf to profile buggy executions of the performance issues in Table 1. For each case, Table 5 shows how many variables were monitored, the time for initializing the vProf-specific profiler data structures, how much memory was consumed by vProf during profiling to store metadata and value samples, and the time to profile the buggy execution. In almost all cases, vProf monitored hundreds of variables for a program component. In all cases, vProf-specific profiler initialization was fast enough to appear instantaneous to a user, and memory overhead was small for vProf's core data structures except in some cases for storing variable samples, which scales based on the number of samples recorded. We further measured the application memory footprint under profiling with vProf and gprof. The maximum memory footprint with vProf scales as expected based on the measurements in Table 5, but the difference versus `gprof` is modest overall. For example, MDEV-13498 has the largest memory footprint, but vProf's maximum memory footprint is only 8% larger than `gprof`. On average, the maximum memory footprint with vProf is 7% (8 MB) larger than with `gprof`.

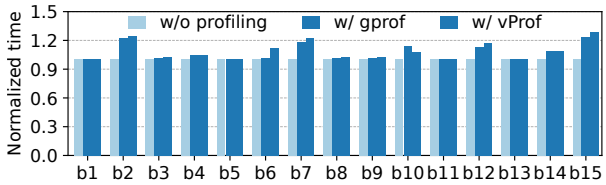


Figure 7. Profiling overhead for performance issues.

Figure 7 shows the runtime overhead of vProf when profiling each performance issue, with performance normalized to execution without using the profiler. For comparison, we also measured the runtime overhead of `gprof` on these issues. vProf runtime overhead is modest in all cases except for when `gprof` overhead is higher, in which case vProf overhead tracks that of `gprof`, on which it is built. We also used sysbench to measure the latency and throughput of MariaDB under a TPCC workload, with and without profiling. Both vProf and `gprof` incurred the same latency and throughput overheads, 32% and 20%, respectively; vProf shows no increased overhead for the features it adds. Overall, these results show that vProf is lightweight and practical for diagnosing performance issues in large applications.

vProf also incurs some cost for its schema generator and post-profiling analysis, which we quantified for the 15 issues in Table 1. vProf's LLVM pass increases compilation time by an average of 5 s. Using DWARF debugging information to obtain variable metadata takes an average of 142 s. Post-profiling analysis takes an average of 117 s. If we monitor variables across the entire program instead of per program component, analysis can take much longer. For example, doing so for Redis-8145 resulted in 17,930 variables being monitored and 7 GB of value samples being recorded, which took post-profiling analysis roughly six hours to process.

6.4 Sensitivity

We evaluated how vProf's effectiveness is affected for the 15 issues in Table 1 for different values of DefaultDiscount and ValidDiscount. We measured effectiveness by how many issues had their root cause function ranked in the top five. We first used the default ValidDiscount of 0.1 and set the DefaultDiscount to different values between 0.1 and 1.0. We then used the default DefaultDiscount of 0.8 and set the ValidDiscount to different values between 0.1 and 1.0. Figure 8 shows that vProf is most effective with a DefaultDiscount of at least 0.8 and a ValidDiscount of less than 0.3.

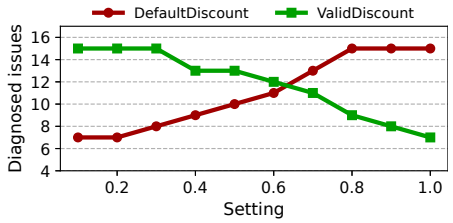


Figure 8. Sensitivity of settings for discount parameters.

7 Limitations

vProf limits the value sampling to variables of primitive types, structure members, and pointers. We plan to extend it to support value sampling for more complex types. The schema generator in vProf runs a call graph analysis. The call graphs can be incomplete due to missing analysis of function pointers.

vProf currently only supports the diagnosis of on-CPU performance issues. Other off-CPU profilers can analyze performance issues to due I/O blocking, paging, locks, *etc.* Our future work will explore applying the idea of value-flow profiling in these off-CPU profilers to support diagnosis of blocking events related performance issues.

vProf's support for multi-threaded applications relies on gprof, which counts the CPU time spent by the whole process and delivers SIGPROF when the timer expires. The method is feasible for multi-threading because the SIGPROF is delivered randomly to one of the running threads. However, vProf could be subject to potential sampling bias. Empirically, since most of our evaluated issues are from multi-threaded applications, vProf is effective despite the potential bias.

8 Related Work

Performance Optimization. Various tools [6, 10–12, 14, 16, 17, 24, 25, 28, 33, 38, 39, 42, 44, 45] help developers find optimization opportunities, including COZ [12], which tells developers the *potential* speed-up *if* a certain function is optimized. While useful, none of these solutions help diagnose specific performance issues and pinpoint their root cause.

Profilers. Many profilers help with performance diagnosis, including Valgrind [32], Oprofile [26], and Gperftools [19]. However, these profilers focus on recording costs and cannot distinguish whether the costs are necessary and why an operation is costly. vProf additionally collects program variable value information along with costs to enable performance reasoning and improve diagnosis effectiveness.

Algorithmic profiling [45] attempts to discover the relationship between an input and the amount of work in a function. Freud [38] extends algorithmic profiling to discover the relationship between input and real performance metrics using regression analyses. These goals are complementary to vProf.

Performance Debugging. Several approaches target debugging performance issues that occur across multiple software components. Stitch [46] reconstructs the execution flow of distributed software using logs. Magpie [7] uses an event schema to correlate events across kernel, middleware and applications for constructing performance models. Argus [43] applies annotated causal tracing on desktop applications to localize the abnormal event sequence. These solutions focus on inferring high-level causality, which is important in distributed systems, but cannot pinpoint the root cause of performance issues in single-component software at precise code locations.

Statistical performance debugging [40] compares predicates in normal and buggy executions to diagnose performance bugs. vProf samples values of program variables while collecting cost information in parallel to provide more effective performance diagnosis, as demonstrated in Section 6.

Some approaches focus on debugging special types of performance issues. X-Ray [5] and GLIMPS [41] diagnose performance issues caused by bad configurations. SyncPerf [1] diagnoses performance problems related to synchronization primitives. vProf is complementary to these solutions.

Performance Bug Detection. Some solutions aim to find performance bugs using static efficiency rules checking [23], static analysis [35], symbolic execution [21], or a combination of static rule checking and dynamic system call analysis [13]. Their effectiveness is limited by the rules, and complex performance bugs are often hard to capture with static rules.

9 Conclusions

Value-assisted cost profiling is a new profiling methodology that provides effective diagnosis of performance issues in real-world applications. It measures execution costs together with program data-flow information to more accurately reason about whether a costly function is necessary and why a function is slow. vProf is a practical tool that implements this methodology. It leverages static analysis to identify variables that commonly influence performance and determine their runtime locations. It builds efficient data structures for profiling to quickly index accessible variables and continuously records value samples with PC sampling. It provides post-profiling analysis to compare value samples across normal and buggy program executions to identify anomalous samples, use them to calibrate function costs, and pinpoint root causes. vProf significantly outperforms other state-of-the-art tools in diagnosing real-world performance bugs in large applications, yet incurs only modest performance overhead. We used vProf to diagnose longstanding unresolved performance issues in real applications, which have been confirmed by developers.

10 Acknowledgments

Michael Stumm provided helpful comments on earlier paper drafts. This work was supported in part by an Amazon Research Award, a Meta Research Award, a Guggenheim Fellowship, a GE/DARPA grant, a CAIT grant, gifts from JP Morgan, DiDi, and Accenture, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, CCF-2124080, CNS-1942794, and CNS-1910133.

References

[1] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proceedings of the 12th European Conference on Computer Systems*, page 298–313, April 2017.

[2] T. W. Anderson and D. A. Darling. Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes. *The Annals of Mathematical Statistics*, 23(2):193 – 212, 1952.

[3] Apache. httpd: Apache Hypertext Transfer Protocol Server. https://httpd.apache.org/.

[4] Apple. macOS Instruments Overview. https://help.apple.com/instruments/mac/current/#/dev7b09c84f5.

[5] Mona Attariyan, Michael Chow, and Jason Flinn. X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, page 307–320, October 2012.

[6] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters. In *Proceedings of the 19th Annual International Conference on Supercomputing*, page 101–110, June 2005.

[7] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 259–272, December 2004.

[8] Eli Bendersky. Parsing ELF and DWARF in Python. https://github.com/eliben/pyelftools.

[9] Damien BRS and Thirunarayanan Balathandayuthapani. Recovery Failure: Loop of Read Redo Log up to LSN. https://jira.mariadb.org/browse/MDEV-21826.

[10] Marc Brünink and David S. Rosenblum. Mining Performance Specifications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 39–49, November 2016.

[11] Milind Chabbi and John Mellor-Crummey. DeadSpy: A Tool to Pinpoint Program Inefficiencies. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*, page 124–134, March 2012.

[12] Charlie Curtsinger and Emery D. Berger. COZ: Finding Code that Counts with Causal Profiling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, pages 184–197, October 2015.

[13] Ting Dai, Daniel Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 641 – 652, September 2017.

[14] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, page 607–622, October 2015.

[15] Michael J. Eager. Introduction to the DWARF Debugging Format. pages 1–11, 2012.

[16] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[17] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 175–184, October 2006.

[18] Thomas Gleixner, Ingo Molnar, et al. perf: Linux Profiling with Performance Counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[19] Google. Gperftools: Google Performance Tools. https://github.com/gperftools/gperftools.

[20] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, page 120–126, June 1982.

[21] Yigong Hu, Gongqi Huang, and Peng Huang. Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 719–734, November 2020.

[22] Intel. Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[23] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 77–88, June 2012.

[24] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, pages 163–181, July 2021.

[25] Chung Hwan Kim, Junghwan Rhee, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. PerfGuard: Binary-Centric Application Performance Monitoring in Production Environments. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 595–606, November 2016.

[26] John Levon. OProfile: A System Profiler for Linux. https://oprofile.sourceforge.io/about.

[27] Sarah Jamie Lewis. A Performance Debugging Story. https://twitter.com/SarahJamieLewis/status/1397313537538592769.

[28] Xu Liu, Kamal Sharma, and John Mellor-Crummey. ArrayTool: A Lightweight Profiler to Guide Array Regrouping. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, page 405–416, August 2014.

[29] LLVM. Writing an LLVM Pass. https://llvm.org/docs/WritingAnLLVMPass.html.

[30] MariaDB. The Open Source Relational Database. https://mariadb.org.

[31] David Mosberger-Tang, Arun Sharma, Dave Watson, et al. The libunwind Project. https://savannah.nongnu.org/projects/libunwind/.

[32] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, June 2007.

[33] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of the 21st ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 268–278, August 2013.

[34] M.S. Nikulin. Hellinger Distance. *Encyclopedia of Mathematics*, 2001.

[35] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 369–378, June 2015.

[36] PostgreSQL. The World's Most Advanced Open Source Relational Database. https://www.postgresql.org.

[37] Redis. A Vibrant, Open Source Database. https://redis.io.

[38] Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. Analyzing System Performance with Probabilistic Performance Annotations. In *Proceedings of the 15th European Conference on Computer Systems*, pages 1–14, April 2020.

[39] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing Performance vs. Accuracy Trade-Offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 124–134, September 2011.

[40] Linhai Song and Shan Lu. Statistical Debugging for Real-World Performance Problems. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, page 561–578, October 2014.

[41] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support. In *Proceedings of the 44th International Conference on Software Engineering*, page 1571–1583, July 2022.
[42] Shasha Wen, Milind Chabbi, and Xu Liu. REDSPY: Exploring Value Locality in Software. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 47–61, March 2017.
[43] Lingmei Weng, Peng Huang, Jason Nieh, and Junfeng Yang. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing. In *2021 USENIX Annual Technical Conference*, pages 193–207, July 2021.
[44] Xin You, Hailong Yang, Zhongzhi Luan, Depei Qian, and Xu Liu. ZeroSpy: Exploring Software Inefficiency with Redundant Zeros. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, November 2020.
[45] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 67–76, June 2012.
[46] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, page 603–618, November 2016.

A Artifact Appendix

This appendix describes the workflow of vProf. The main steps are to build vProf with the clean glibc and a provided patch, prepare workloads to run target applications, collect profiling data, and analyze the data with post-profiling scripts. The source code of vProf is available at: https://github.com/wenglingmei/vprofAE.

A.1 Abstract

Configuring vProf takes four steps:

1. patch glibc to support collected variable values from metadata;
2. run an LLVM pass to generate a schema file and translate the schema into a metadata file;
3. compile a target application with `-pg -O2` to support profiling like gprof;
4. run a target application with `LD_PRELOAD` to link the patched libc-2.31.so for vProf.

Reproducing the diagnosis results in our paper requires profiling both the normal and buggy runs of the application. With the profiling data, our post-profiling analysis generates annotated profiling reports similar to traditional profilers.

All applications evaluated in the paper are publicly available from their official websites, but users need to prepare workloads for reproducing the performance bugs and constructing a normal baseline. We provide scripts for reproducing one of the evaluated issues as an example.

A.2 Description & Requirements

A.2.1 How to access.

- Download the vProf source code from https://github.com/wenglingmei/vprofAE.

A.2.2 Hardware dependencies.

- Architecture: x86-64
- Memory: >= 32 GB

A.2.3 Software dependencies.

- **System**: Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-27-generic x86_64).
- **Development Tools**: Install tools with the command `sudo apt install build-essential`.
- **Python**: Python 3.
- **llvm-project**: Clang 14.0.0 downloaded from LLVM official repository https://github.com/llvm/llvm-project. The specific commit ID used in vProf is `3782624`. Compiling LLVM requires that you have several software packages installed: CMake >= 3.13.4; GCC >= 7.1.0; Python >= 3.6; zlib >= 1.2.3.4; GNU Make >= 3.79. Compilation options: `-DCMAKE_BUILD_TYPE=Release`. and `-DLLVM_ENABLE_PROJECTS="clang;lld"`
- **glibc**: glibc 2.31 is the default version shipped with Ubuntu 20.04.3.
- **libunwind**: download and install the newest version from https://github.com/libunwind/libunwind.
- **pyelftools**: install the *pyelftools* library from https://github.com/eliben/pyelftools.

A.3 Set-up

We provide a one-click script `prepare.sh` under the directory `vprofAE`. It is composed of the following steps:

- Download vprofAE and software dependencies in Section A.2.3.
- Install the development tools and software dependencies.
- Prepare glibc with `glibcForPRELOAD/build_glibc.sh`.
  To include `libunwind` in glibc:
  build – build a clean `glibc` first
  patch – patch the glibc with `glibc-2.31.patch`
  rebuild – rebuild the patched glibc without clearing the built object files
  softlinks – creates soft links to correct the version issues of libraries referenced by libc
- Install LLVM following the official instructions with the options mentioned in Section A.2.3.
- Build the LLVM pass `libProfileVarPass.so` in directory `LLVMPassSchemaGen` for generating schema, and *make sure to run the small code example with the LLVM pass before moving to Section A.4.*

A.4 Evaluation Workflow

A.4.1 Major Claims. The paper has the following major claims in the evaluation part.

1. Diagnosis Effectiveness. The function rankings from vProf are annotated with variables, locations where anomalous values are accessed, and bug patterns. All the annotations allow developers to debug performance issue more effectively.
2. Sensitivity to Parameters. During post-profiling analysis, varying the DefaultDiscount and ValidDiscount parameters is done to assess their impact on the profiling report.
3. Overhead. To determine profiling overhead, the test case can be executed with gprof, vProf, or without profiling. We measured CPU usage, execution time, and maximum memory usage at the end of profiling.

A.4.2 Experiments.

Experiment 1: Diagnosis Effectiveness.

- Set the environment variable `SchemaComponent` with the source code path.
- Create a schema file by compiling the source code with the LLVM pass `libProfileVarPass.so`.
- Build the application with `-pg -O2` to enable profiling.
- Translate the schema into variable metadata using the script `translate_schema_multiprocessing.py`.
- Link the metadata file to /tmp/vprof/info.txt, from which glibc loads the variable metadata.
- Run both normal and buggy executions repeatedly and collect data into directory `norms` and `bugs`. For each run, vProf will produce three files:
  `/tmp/vprof/gmon/gmon.[pid].out`
  `/tmp/vprof/gmon_var/gmon_var.[pid].out`
  `/tmp/vprof/layoutl/layout.[pid].out`.
  Note: The script `redis-8145/test.sh` is provided to reproduce the results for Redis-8145 in the paper. If the test case fails due to limited hardware resources, reduce the cluster nodes in the Redis test script, repeat this step, and collect the profiling data.
- Run the post profiling Python script `vprof_profile.py` to produce the result.

Experiment 2: Sensitivity to Parameters.

- DefaultDiscount and ValidDiscount can be directly set via the Python script `vprof_profile.py`.
- Check the result in the vProf profiles.

Experiment 3: Overhead.

- Run the test case with gprof, vProf, or without profiling, and measure the CPU usage and memory usage of the process using the `ps` command.
- The runtime overhead can be measured with the `time` command.
- The memory overhead for storing data structures and value samples is printed by running the python script `var_sample_multiprocessing.py`, which takes the file `/tmp/vprof/gmon_var/gmon_var.[pid].out` as input.