# Performance Comprehension at WiredTiger

Alexandra Fedorova
University of British Columbia
Canada

Craig Mustard
University of British Columbia
Canada

Ivan Beschastnikh
University of British Columbia
Canada

Julia Rubin
University of British Columbia
Canada

Augustine Wong
University of British Columbia
Canada

Svetozar Miucin
University of British Columbia
Canada

Louis Ye
University of British Columbia
Canada

## ABSTRACT

Software debugging is a time-consuming and challenging process. Supporting debugging has been a focus of the software engineering field since its inception with numerous empirical studies, theories, and tools to support developers in this task. *Performance* bugs and performance debugging is a sub-genre of debugging that has received less attention.

In this paper we contribute an empirical case study of performance bug diagnosis in the WiredTiger project, the default database engine behind MongoDB. We perform an in-depth analysis of 44 Jira tickets documenting WiredTiger performance-related issues. We investigate how developers diagnose performance bugs: what information they collect, what tools they use, and what processes they follow. Our findings show that developers spend the majority of their performance debugging time chasing outlier events, such as latency spikes and throughput drops. Yet, they are not properly supported by existing performance debugging tools in this task. We also observe that developers often use tools without knowing in advance whether the obtained information will be relevant to debugging the problem. Therefore, we believe developers can benefit from tools that can be used for unstructured exploration of performance data, rather than for answering specific questions.

## CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software development methods**; *Programming teams*;

## KEYWORDS

Software performance comprehension, performance debugging

**ACM Reference Format:**
Alexandra Fedorova, Craig Mustard, Ivan Beschastnikh, Julia Rubin, Augustine Wong, Svetozar Miucin, and Louis Ye. 2018. Performance Comprehension at WiredTiger. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236081


*ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA*

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236081

## 1 INTRODUCTION

Software is becoming increasingly complex and spans mobile devices, data centers, diverse architectures (multi-core, GPUs), etc. In this world performance is an important software engineering consideration [35]. In particular, many software products are routinely compared, rated, and bought or not bought based on their performance. This includes databases, key-value stores, distributed computing systems, machine learning frameworks, and more.

Achieving high performance requires performance debugging: identifying bottlenecks and mitigating them. However, without understanding the underlying cause, without proper information and experiments to reproduce the bug, developers are at a loss of how to fix a performance bug. This process, which we term *performance comprehension*, has not been previously detailed.

Existing work on performance debugging develops new tools [12, 13, 25, 34, 37] to complement mainstream commercial and open source performance debugging tools such as *perf*. Instead, this paper asks: what information do developers collect? Do they formulate hypothesis, or are they driven by the tool at hand? And, how do they collaborate to understand system performance? In other words, we are interested in answering the broad question of: how do developers approach performance comprehension?

We explore how developers diagnose performance issues with a case study of WiredTiger (WT)[1], an open-source high-performance storage engine written in C. As of 2014, WT is part of MongoDB[2], a popular NoSQL database. WT developers collaborate in a decentralized manner and use Jira [43] for tracking and for thoroughly discussing and resolving all performance issues.

In our study we analyzed 44 performance-related tickets in WT's Jira repository. We focused on the information that developers do and do not collect, tools they use and do not use, and processes they follow. We also considered the relationship between information and tools: do information needs drive tool usage, or do developers prefer and use certain tools first, and then sift through the resulting information? Here are a few of our findings from the study:

**Finding 1.** WT developers spend most of their performance debugging time chasing latency spikes and throughput drops. Despite being outliers, these events represent worst-case system behaviour and are thus of critical importance for performance-oriented groups such as WT.

**Finding 2.** When looking for root causes of latency spikes, developers consider correlated behavior between threads and events that happen over time. However, developers are not properly supported by existing performance debugging/comprehension tools.

[1] http://www.wiredtiger.com/
[2] https://www.mongodb.com/

Most program profilers aggregate samples and cannot identify occasional latency spikes. The WT developers thus rely on manually correlated log messages to diagnose tail latencies.

**Finding 3.** Developers typically do not formulate initial hypotheses about the root cause of a performance issue. We found that developers did not know in advance if the performance data they plan to collect from running a tool will be relevant to debugging the problem. They often used the tools in an exploratory fashion.

## 2 STUDY DESIGN

For our case study, we extracted performance-related tickets from WT's Jira repository and analyzed them to answer three research questions:

**RQ1:** What information do developers collect during performance comprehension?

**RQ2:** What tools do developers use during performance comprehension?

**RQ3:** What processes do developers follow during performance comprehension?

Next, we give the necessary background on the WiredTiger project and its software practices. We then describe our data collection, analysis and validation techniques, and outline threats to the validity of our study.

### 2.1 WiredTiger

WiredTiger (WT) is a multi-threaded open-source storage engine, designed for high performance and scalability. Started in 2008, it has a team of around 24 developers[3], with the core six developers contributing over 200 commits each. At the time of writing, the most active developer contributed over 9,000 commits, and the most active six developers contributed over 13,000 commits in total. Three of these developers have 30+ years of experience, one has around 15 years of experience, and the remaining two developers have around 5 years of experience.

WT is comprised of over 84,000 lines of C code[4] and runs on POSIX-compatible systems, including Linux and FreeBSD, as well as Windows. WT clients include several popular web service providers, such as Amazon Web Services. In 2014, WT was acquired by MongoDB [31] — the provider of one of the most popular open-source document-oriented (NoSQL) databases. WT achieves a 7-10x performance increase over MongoDB's previous storage engine [30].

**Why study WT.** We use WT to study performance comprehension because the main goal of WT is to develop a high performance storage engine [30]. WT is carefully designed to efficiently use all available CPU cores, large amounts of RAM, and advanced synchronization techniques to enable more efficient parallelism. Developers thus spend a significant amount of effort understanding and resolving performance-related issues. Moreover, WT presents a unique opportunity for our study because its open development makes it possible to investigate performance comprehension practices of a successful performance-oriented software team.

[3]Reported by GitHub's contributors list: https://github.com/wiredtiger/wiredtiger/graphs/contributors

[4]Calculated using cloc on .c, .cc, .h, and .i files in the src and api directories of the WT repository.

**Development process.** WT developers work remotely from around the world and communicate with each other online. At the time of writing, there is a significant number of commits from seven distinct time zones. Developers communicate primarily through Jira tickets for issue tracking (over 3,700), saying that they used Jira "as a notebook for debugging sessions". Other, less significant communication channels are Github [42] pull requests for code reviews and merges (over 19,000 commits and over 2,900 pull requests), and mailing lists. Yet, the project conventions are to maintain all conversations about issues in Jira as opposed to other means, such as email. The WT team does not use IRC or Slack.

WT developers use the Jenkins [4] continuous integration (CI) system to monitor the health and performance of the latest product version. For each new commit, Jenkins automatically runs tests to evaluate the current quality and performance of the system. Jira performance issues are created following failures listed in Jenkins' reports, directly by customers, or by developers who encountered performance-related issues while working on the project.

### 2.2 Data Collection

For WT, Jira is a sufficiently complete source of performance debugging activity, which motivated us to perform our study with this data source. To collect performance-related tickets for our analysis, we navigated to the MongoDB Jira issue tracker repository[5] and selected project *WiredTiger*. We searched for tickets that satisfy the following criteria:

- Tickets of any standard type, i.e., bug, documentation, improvement, new feature, task, technical debt, or workload.
- Tickets in any state, i.e., open, in progress, in code review, resolved, closed, waiting for user input, needs reviewer, or needs merge.
- Tickets containing the keyword "performance" in either the title or the body of the ticket.

We sampled the repository twice: first in August 2017 and then in February 2018. Both times, we analyzed the tickets one by one, using the method described in Section 2.3. We stopped when we reached *conceptual saturation* – no new information was obtained from analyzing further data. As the result of this process, we analyzed 44 tickets updated between April 2016 and December 2017 (34 in the first round and 8 in the second).

The analyzed tickets contained between 1 and 39 comments made by developers (median 6.5, mean 8.7), with 1 to 7 distinct developers contributing to each ticket (median 3, mean 3). Initial issue descriptions were between 13 and 394 words (median 86.5, mean 111.6). Developer comments were between 3 and 915 words (median 60.5, mean 93.3). In total, 16 developers contributed to the majority of tickets, with each of the top four developers contributing to more than 10 issues.

### 2.3 Data Analysis

We used *open coding* – a technique from grounded theory for deriving theoretical constructs from qualitative analysis [15, 20]. Two authors of this paper independently read each selected Jira ticket and coded ideas contained in the data. We focused on information,

[5]https://jira.mongodb.org

tools, and processes that developers use to diagnose performance issues. We did not code the description of the problem or its resolution.

On a weekly basis, all authors of the paper met to discuss quotes identified by the two coders, review their grounding in Jira tickets, merge and refine codes and concepts as necessary, identify emerging categories, and also refine coding rules and processes. Following Strauss's version of grounded theory, where data sampling and data analysis are seen as interleaved steps that are repeated until one can describe the researched phenomenon, we continued coding the tickets until all authors agreed that new data does not change our understanding of performance diagnosis process. At this stage, we coded 36 tickets originally sampled in August 2017. To further validate our results and ensure a stable coding process and set of concepts, one author of the paper sampled the repository one more time in February 2018 and coded another eight tickets. The complete set of codes is available online[6].

As a result of this process, we analyzed 44 tickets. We marked between 1 and 106 quotes per ticket (median 14, mean 25.5), identifying 36 codes related to information needs, 22 codes related to tools, and 26 codes related to process (9 for information and tools inter-relationships, 7 for reproducibility, and 12 for experiments). The four most senior developers were responsible for comments corresponding to 71.6% of the codes. These are discussed in detail in Sections 3 – 5.

## 2.4 Member Check

To validate our findings, we shared an earlier version of this paper with the WiredTiger developers whose comments we studied. Three developers responded; together, the respondents account for 60% of all comments written across 37 (84%) of the tickets we studied. Two of the three developers said they read the entire paper.

**Comments on findings.** We asked if the developers agreed with our findings. The respondents found our findings accurate; they did not dispute the findings in any of the sections, offering feedback on typos and grammatical fixes.

> I'm frankly surprised how accurately you managed to capture the [performance issues] in the paper - it did an excellent job of defining the categories.

**Completeness of studying Jira tickets.** We asked developers whether their comments on Jira tickets accurately reflect the challenges they faced and the tasks they performed in diagnosing and fixing performance issues. Two of the three developers responded in the affirmative (the other developer did not comment on this question).

> Personally, I comment in tickets for a few purposes. Some of it is so that we have a history and can revisit "why did we do that?" at a future time after we've (or I've) forgotten the details. Some is so that others can reproduce what I'm doing and see (or not) the same results. Some is to validate or simply communicate how I'm attacking a problem.

## 2.5 Threats to Validity

**Internal validity.** We might have misinterpreted developers' intentions or misidentified concepts, introducing researcher bias into our analysis. We attempted to mitigate this threat by performing independent coding and cross-validating each code identified in our analysis by at least two authors. We also discussed and validated the results during weekly meetings with all authors of this paper.

Moreover, one member of our team worked as a consultant for WT and is thoroughly familiar with the WT development processes. (S)he confirmed that our understanding of the debugging traces in the tickets we studied are representative of performance debugging sessions. In addition to the person who consulted for WT, our team had other members who developed database software for other companies, so we were well positioned to understand the technical details in the tickets. We thus believe our analysis is reliable.

In organizations where developers communicate through a variety of different channels, especially in-person, bug reports or issue trackers provide incomplete views of a bug's history [9]. However, for WT, the Jira database is the main communication channel; we are thus confident that Jira is a sufficiently complete source of performance-related debugging information for our study.

**External validity.** As in other software engineering case studies, our research might not generalize beyond our subject of study: the WT system and the sample of tickets we consider. It is possible that the practices used by other companies and teams may differ from those used at WT. Our tickets sample includes patterns and practices followed by 16 experienced software developers, giving us confidence that our study captures at least some of the best practices for performance diagnosis.

# 3 RQ1: INFORMATION USED

The information developers collect during performance comprehension (**RQ1**) is captured by 36 codes extracted from Jira tickets. Table 1 shows codes that appeared in at least 10% of the tickets, together with their descriptions, the number of tickets containing each code, and the total number of occurrences of each code across all documents. We categorized the codes along three dimensions: temporal view, execution view, and system stack location, as shown in Figure 1. The next section describes these categories, followed by a detailed description of findings derived from the categorization.

## 3.1 Categorizing Information Codes

**Temporal view** tells us whether the information about program execution (typically metrics like throughput, latency, CPU time breakdown by function) were used as aggregates, e.g., averaged or summed up across an execution, or in the form of time series. An example of time-series presentation is where a metric of interest, e.g., the latency of a function, is presented as a set of measures, so that the developer has a chance to see how the metric changes during the execution and correlates with other metrics or events.

**Execution view** distinguishes between information aggregated across all execution contexts or threads, vs. when it is presented for individual threads. For example, latency of a critical function could be averaged across all threads or could be examined separately for each thread (or thread type), allowing the developer to learn whether some threads ran more slowly than others.

**System stack location** describes the origin of the information: is the metric related to the behavior of the hardware, such as CPU utilization or disk throughput? Does the information originate

[6] https://goo.gl/DdwU7u

| Temporal view | | | |
|---|---|---|---|
| Aggregated across execution | | Time series | |
| **Execution view** | | | |
| Aggregated across all threads/processes | | Per-thread | |
| **System stack location** | | | |
| Hardware | Operating system | Compiler/ libraries | Application |

**Figure 1: Dimensions of classification for information codes.**

from measuring events in the operating system? Does it come from measurements related to the compiler or libraries? Or, does it originate solely from measuring the events in the application, such as the duration of functions or properties of data structures?

Codes can be labeled along each dimension; a sequence of labels produces a category for a given code. For example, a code that describes information in the form of time series that is presented for each individual thread would fall into *Time-series / Per-thread* category. Table 1 organizes our codes into categories using the first two dimensions (temporal and execution views). In the third dimension (system stack location), the *Application* label dominates, so we do not further split the categories along this dimension (codes in the table correspond to *Application*, unless noted otherwise). Only codes that appeared in at least 10% of the tickets are listed in the table.

Most of the codes are deterministically labeled along each dimension, with two exceptions at the bottom of Table 1 where we did not have sufficient data to uniquely label the codes. For example, the code info-code-behavior, which indicates that the developer referred to the application source code in order to reason about a performance anomaly, was difficult to label as either *Per-thread* or *All threads* along the execution view dimension: source code is static and it is not always clear if a particular code sequence would be executed by all running threads or by a certain subset.

## 3.2 Time Series / Per-thread Category

**Developers need to understand thread activity over time.** *Time Series / Per-thread* is the most prominent category, including 8 out of 17 most frequently occurring codes. That is, developers mostly looked for information across time and across threads, as opposed to the aggregated form. This is despite the fact that aggregated information is much easier to obtain (via tools like CPU profilers); as we will see in the next section, getting time-series per thread information required rudimentary and manual methods, such as manual code instrumentation or stopping the debugger at just the right time to observe specific events.

We believe that the need for *Time Series / Per-thread* information is not unique to developers at WiredTiger. Google's and IBM's in-house performance tools display the execution trace as time series broken down by execution context [1, 17].

The most frequently occurring code is info-functions-or-activity-over-time-by-threads. It represents a situation where a developer learned what threads were doing during different periods in the execution, e.g., what functions or groups of functions they were executing, when, and for how long. In one example the developer wrote:

> There are two checkpoints, the first starts at about (40 total secs) and it finishes quickly, before the next 10 second stat report. The next checkpoint starts at about (80 total secs) and continues for the rest of the test.

In this case, a checkpoint is identified by a group of functions and is executed by a specially designated checkpoint thread. The developer observed when the checkpoint started and ended. In another case, a developer wanted to learn which WT functions the thread is executing, if any, and when:

> Running pmp, I almost never catch any thread in WT during the 1-thread run. With 8 threads, I always see 7 out of 8 yielding in `__wt_page_in_func`.

Another group of codes dominating the *Time Series / Per-thread* category relates to threads being blocked, delayed, active or inactive. Information about blocked or delayed threads was especially difficult to piece together and required developers to analyze series of callstacks obtained via a debugger or by sifting through log files. Below are the codes in this category and the example quotes describing them:

info-threads-blocked-delayed-due-to-certain-functions-or-activity:

> ...the vast majority of the time is spent in application threads in `cond_wait` waiting for `cache->evict_current` to have something. This means app threads are stuck in the loop in `wt_cache_eviction_worker`.

info-threads-waiting-over-time:

> There are often periods where most (91/100) threads are waiting on cache for a full second.

info-threads-being-active-or-inactive-affects-performance:

> This drop is due to the async threads that are created for the compact. The rest of the test is not async, only the compact operation is done using async. Therefore, the async threads spin there looking for work.

**Latency spikes and throughput variations are a key concern.** Much of performance debugging is centered around latency spikes and intermittent drops of throughput. These are rare events that occur when the duration of certain functions or operations or the rate of their completion falls far below average.

info-latency-spikes:

> I got over 500 instances of slow operations on [branch X] and [commit Y], with maximum reported latencies over 1s.
>
> In running the `evict-btree-stress-multi.wtperf` workload, I see a number of multi-second latencies.

info-intermittent-throughput-drop:

> The load versus read phases are very distinct and the load phase shows a 23% drop in performance, taking longer to populate.

This observation is, again, not endemic to WiredTiger. Google's Dean and Barroso wrote about how these rare but important events can wreak havoc in large systems [18]. Latency spikes and drops of throughput can only be observed when the performance metrics are presented as time-series; any aggregated metric, such as overall

| Category and code | Code description | Tickets | Total |
|---|---|---|---|
| **Time series, Per-thread** | | | |
| ▹ Info-functions-or-activity-over-time-by-threads | Developer obtains the functions or groups of functions (i.e., activity) that were executed over time by different threads | 21 | 56 |
| ▹ Info-threads-blocked-delayed-due-to-certain-functions-or-activity | Developer finds when and for how long threads are blocked or delayed when executing certain functions or performing certain activity | 14 | 22 |
| ▹ Info-latency-spikes | Developer observes unusually long latencies seen in the program. | 10 | 47 |
| ▹ Info-intermittent-throughput-drop | Developer observes that throughput occasionally drops during execution. | 9 | 18 |
| ▹ Info-threads-waiting-over-time | Developer finds that threads are waiting on something (e.g., a lock or another condition) during particular portions of the execution. | 8 | 12 |
| ▹ Info-intermittent-throughput-drop-tied-to-certain-functions-or-operations | Developer finds that throughput drops occur when a certain operation (e.g., checkpoint, cache eviction) is happening or when a certain function is executing. | 6 | 15 |
| ▹ Info-latency-spikes-tied-to-certain-functions-or-operations | Developer observed that a latency spike was observed at the same time as a certain function or group of functions was executing. | 6 | 11 |
| ▹ Info-threads-being-active-or-inactive-affects-performance | Developer observes that performance varies depending on which threads were running (or not) during a particular time in the execution. | 5 | 18 |
| **Time series, All threads** | | | |
| ▹ Info-correlation-of-data-structure-state-with-execution-progress | Developer observes that the state of a data structure correlates with how quickly the code runs. | 16 | 39 |
| ▹ Info-data-structure-state-over-time | Developer observes the data structure state (e.g., its size or the presence of specific elements) during various periods in the execution. | 10 | 24 |
| **Aggregated, All threads** | | | |
| ▹ Info-where-time-is-spent-within-a-function | Developer observed where time is spent in a certain function. | 12 | 16 |
| ▹ Info-use-of-synchronization-primitives-affects-performance | Developer observes that using synchronization primitives (e.g. fine vs. coarse) or using one synchronization primitive over the other, or using synchronization primitives too often affects performance. | 9 | 18 |
| ▹ Info-function-cpu-hotspot (OS,Lib,App) | Developer obtained a CPU profile of the running program. | 6 | 14 |
| **Aggregated, Per-thread** | | | |
| ▹ Info-work-accomplished-by-different-threads | Developer compared different threads or different thread types in terms of how much work they accomplished. Work is a loosely defined as anything that had to be done in the program. | 6 | 15 |
| **Aggregated, (Per-thread \| All threads)** | | | |
| ▹ Info-code-behavior | Developer relied on their knowledge of the application code or specifically examined the code. | 21 | 44 |
| **(Time series \| Aggregated), (Per-thread \| All threads)** | | | |
| ▹ Info-performance-tied-to-kernel-activity-or-state (OS) | Developer learned whether the performance correlates with some activity in the kernel (e.g., paging) or a particular kernel state. | 7 | 12 |
| ▹ Info-correlation-of-performance-and-IO (OS) | Developer learned whether performance, either aggregate or during specific times in the execution, correlates with reading or writing files. | 6 | 9 |

**Table 1: Information codes that occurred in at least 10% of the issues. Codes relate to the application level of the stack, unless indicated otherwise. Issues refers to number of issues with code, while Total records the total number of code occurrences.**

throughput or average latency, will conceal them. Profilers do not sample at high enough resolution to detect rare latency spikes, so developers at WiredTiger, Google [1] and IBM [17] alike built their own tools to observe latency spikes in per-thread or per-process timelines. We were surprised that there was no off-the shelf tool that the WT developers found suitable for their needs.

The remaining codes in this category describe developers trying to understand what caused those latency spikes or throughput drops to occur.

info-intermittent-throughput-drop-tied-to-certain-functions-or-operations:

> . . . what I see in a typical run is that performance drops 95% when a checkpoint runs.

info-latency-spikes-tied-to-certain-functions-or-operations:

> . . . when I investigated it for $benchmark at the beginning of this issue, the high latencies always were at the times of merges.

We observed that the WT process of correlating performance anomalies with various program events was entirely manual.

### 3.3 Time Series / All threads Category

**Developers need to observe data structure state over time.** Both codes in this category deal with data structures; specifically, how the state of the data structure changes over time and how it can explain sudden changes in performance:

info-correlation-of-data-structure-state-with-execution-progress:

> When cache utilization hits 95% performance falls off a cliff.

info-data-structure-state-over-time:

> The cache is regularly 100% empty (as per the empty score).

This information was obtained entirely via manually-inserted log messages. While there are tools that automatically track function calls [2, 7, 11], tracking data structures and especially understanding the semantics of what is being tracked is more difficult [36, 38].

### 3.4 Aggregated / All threads Category

This category contains three codes; for two of them, the information developers need can be trivially delivered by the profiler: info-function-cpu-hotspot, where developers obtain the CPU profile of the running program, and info-where-time-is-spent-within-a-function, where developers observe time spent in a certain function. We noticed that some developers always ran the profiler as the first step in debugging a performance issue.

Understanding the use of synchronization primitives was more important than obtaining the CPU profile, but this information was typically collected from multiple sources and often required understanding of the synchronization protocol. The following quote for the code info-use-of-synchronization-primitives-affects-performance demonstrates that the developer had to understand a complex synchronization primitive and what it was used for:

> Using a read/write lock (to account for direct writes) is the dog [cause of the performance issue].

### 3.5 Remaining Information Categories

**Application behavior, and not OS or hardware, is the focus.** It is notable that the remaining categories, just like the first three, are dominated by codes where the information was obtained at the application level. Only the codes in the *(Time series | Agg.), (Per-thread | All threads)* refer to the information obtained at the OS level. This finding is surprising as WiredTiger is a key-value store, so we expected the I/O and virtual memory management sub-systems to be important for performance.

OS-level information was mostly used to explain latency spikes or throughput drops (info-correlation-of-performance-and-IO):

> More runs of this and I've confirmed that there isn't a 100% relationship between read or write operations and the delays. I have instrumented every read and write operation over 250ms and I can see that there are periods when we report operational latency over 700ms, with no reported slow read and write operations.

**Understanding the code base is important, but is no substitute for high-quality performance data.** In several cases, information was obtained by analyzing the source code or by relying on existing knowledge of the code, rather than by using a performance tool (info-code-behavior):

> . . . with the way the code is currently structured, `__wt_verbose` is always called even if it is an empty function.
>
> The reason is that the [branch X] is much more cautious at creating additional eviction workers, and as a result we have less contention and a higher throughput.
>
> The higher latencies for app threads tend to come when eviction is needed but there is no work available.

Anecdotally, we noticed that some developers tended to be "rationalists" in that they more often tried to explain performance anomalies using their innate knowledge of the code, while others were "empiricists": they always gathered experimental data before forming hypotheses. Perhaps the key observation is that even very experienced developers could not debug tough performance issues relying on their knowledge of code alone; having high-quality performance information was crucial.

## 4 RQ2: TOOLS USED

To understand which tools developers use during performance comprehension (**RQ2**), we identified 22 codes related to tools. We categorized the tools according to the same dimensions as the information codes in Section 3. Table 2 shows the codes that appeared in at least 10% of the tickets, along with their descriptions and categories. The remainder of this section summarizes our main findings regarding how WT developers use tools.

**A manually-generated application log is the most often used performance debugging tool.** The top category, *Time series / (Per thread | All threads)*, includes the codes related to using the application log. Tool-manual-log-existing was used when the previously instrumented log messages were sufficient to get insight into the problem. Tool-manual-log-new was used when the developer had to manually add new log messages to get to the bottom of the issue. The relative frequency of this code (it occurred in 23% of all tickets) indicates that obtaining the needed information is largely a manual process; even in software with established logging, developers often had to manually add new instrumentation. This is consistent with prior work that noted the high frequency with which logging code is modified [45].

| Category and code | Code description | Tickets | Total |
|---|---|---|---|
| **Time series, (Per-thread \| All threads)** | | | |
| ▹ Tool-manual-log-existing | The software was previously instrumented to output log messages. The developer turned logging on to collect information for performance debugging. The log included messages generated over the course of the execution (hence the time-series temporal view). Depending on the context, each message identifies the thread that logged it (per-thread execution view), or can reflect information aggregated over all threads. | 20 | 45 |
| ▹ Tool-manual-log-new | This code was used when the developer relied on the same kind of application-level logging as covered by the tool-manual-log-existing code, but the existing instrumentation was not sufficient, and the developer had to add new log messages to the code to obtain the data they wanted. | 10 | 18 |
| ▹ Tool-show-statistics-over-time | This code was used for two in-house tools that visualized statistics, collected via an application log, as time series. | 6 | 12 |
| **Time series, Per-thread** | | | |
| ▹ Tool-gdb-callstacks (Lib,App) | Developer used a debugger, gdb in our case, not for program errors/crashes, but to understand where and when threads spend their time. | 9 | 12 |
| ▹ Tool-pmp (Lib,App) | Pmp stands for poor man's profiler [5]. This is a script that periodically collects gdb callstacks across all threads, and pre-processes them with awk. | 8 | 10 |
| **Aggregated, All Threads** | | | |
| ▹ Tool-profiler (OS,Lib,App) | This is a typical CPU profiler: a tool that automatically provides per-function breakdown of CPU cycles and other information. In our case, profilers included perf, Zoom, and instruments. Developers used profilers to only look at information aggregated across threads. | 8 | 17 |

**Table 2: Tool codes in our study. Codes relate to the application level of the stack, unless indicated otherwise. Tickets is the number of tickets with code; Total is the total number of code occurrences.**

**Developers have yet to find the right tool for visualizing logs.** To analyze the log files, developers either manually sifted through them or used visualizations. In our case study, developers used two different tools, both covered by the code tool-show-statistics-over-time. Over time one tool was deprecated and replaced. Recently, WiredTiger has built another in-house tool that specifically tracks long-running operations [8]. This churn in tools suggests that developers remain on the lookout for the right log visualization tool.

**When the log does not provide per-thread information, developers seek it by other means.** It was surprising to see that even though developers usually required per-thread information (see Section 3), the application log provided it in the aggregated form. In some cases it was possible to distinguish between thread types, such as application threads vs. internal housekeeping threads, based on the contents of the log messages. But when this level of detail was insufficient, the developers resorted to debugger-based tools to fill in the gaps.

Two codes in the *Time series / Per-thread* category illustrate these situations: Tool-gdb-callstacks was used when the developer manually stopped the debugger and examined the callstacks to infer where threads spent time, and in particular where they blocked. Tool-pmp refers to Poor Man's Profiler [5], which is a set of scripts around gdb that obtain callstacks periodically and parse them with awk. Here are examples illustrating how these tools were used to obtain per-thread detail:

> Looking at the PMP traces we could see that there were 7 threads sleeping at all times and one thread working.
>
> Running pmp, I almost never catch any thread in WT during the 1-thread run. With 8 threads, I always see 7 out of 8 yielding in `__wt_page_in_func`.
>
> When running pmp on mongod with WT and running a single thread, any time I saw a thread actually in the WT library, it was in `pwrite`, via the logging subsystem.

Since these tools were not built specifically for obtaining per-thread performance information they appeared awkward to use and often required orchestration, so that the stack traces could be collected at just the right time. For example:

> I do have pmp scheduled to run around the time of the very large dropoff.

**Profilers are not as prevalent as one might expect.** The category *Aggregated / Aggregated* contains the code generated whenever a profiler was used. We observed three profilers in our data: Linux's perf, zoom [6], and MacOS's instruments. Profilers are easy to use and impose low overhead, because they use sampling. Unfortunately, sampling is unable to capture rare latency spikes, so profilers are used to obtain time-aggregated information. Although profilers can provide per-thread breakdown, we did not see them being used in this way.

We were at first surprised that profilers were not used more often. As we realized that developers spent the majority of their time

chasing latency spikes and throughput drops, it became clear why profilers were not helpful. The following quote from a developer captures the sentiment:

> I would only expect perf to be of limited use here. It is useful in giving an indication of where time is spent over the entire run which is good for general performance tuning but I don't know how to use the results to find what is causing outlier operations[7].

## 5 RQ3: PROCESS

Besides collecting information and using tools, performance comprehension involves many processes. For example, developers share and discuss information, formulate hypothesis, perform and report on experiments, etc. We seek to understand the process that WT developers follow during performance comprehension (**RQ3**). The three most dominant types of processes that we observed are: (1) the relationship between tools used and information gathered, (2) the process of reproducing a performance issue, and (3) experimentation to gain new insights into an issue.

### 5.1 Tools and Information Inter-relationships

Tools generate information, but information may also lead to tool usage. We found substantial evidence that the two are closely linked.

We created a mapping between tools and information using process codes. A process code was created whenever the text of the document made it clear that a developer used a specific tool to obtain a specific piece of information. This produced a large number of codes (63) because every such process code reflects a relationship between some tool and some information.

We created process codes by concatenating the tool code with the code for the information obtained using the tool. If the developer indicated that (s)he sought a specific type of information and then used the tool to obtain it, the info- string appears first in the process code and the tool- string appears second, as in:
process-**info**-functions-or-activity-over-time-by-threads-**tool**-pmp.
On the other hand, if the developer appeared to use the tool in an exploratory fashion, that is, without deciding in advance which information produced by the tool might end up being useful in performance debugging, the tool string appears first and the information string appears second:
process-**tool**-pmp-**info**-functions-or-activity-over-time-by-threads.
The following quote is an example of the information-first process:

> I added new stats and turned on statistics logging [...] to see how many times *wt_lsm_manager_pop_entry is called and how many times that call results in a manager operation getting executed.

And, this quote exemplifies the tool-first process:

> I collected WT statistics for all runs too. Maybe there is some trend we can see that can help skew the conditional. Here are the %-ages of [...] how often the server thread helped evict pages.

In Section 3 we described the categories we used to group information and tool codes. The category for a process code is a concatenation of the categories corresponding to the tool and to the information. We further grouped these categories into *concepts*, reflecting whether the tool or the information was time-series, aggregate, per-thread or across threads. Since both the tool and the information could correspond to these four concepts, there was a total of 4x4=16 possible concepts for process codes. Of these, only five concepts listed in Table 3 appeared in at least 10% of all tickets: (1) a time-series per-thread tool was used to obtain the time-series per-thread information, (2) a time-series per-thread tool was used to obtain time-series information aggregated across all threads, (3) a time-series tool that aggregates data across threads was used to obtain time-series information aggregated across threads, (4) a time-series tool that aggregates data across threads was shoehorned into obtaining time-series per-thread information, and (5) a tool that produced data aggregated across time and across threads was used to obtain the same kind of information.

| Tool category | Information category | Tickets |
|---|---|---|
| Time series / Per-thread | Time series / Per-thread | 30 (68%) |
| Time series / Per-thread | Time series / All threads | 5 (11%) |
| Time series / All threads | Time Series / All threads | 14 (31%) |
| Time series / All threads | Time Series / Per-thread | 5 (11%) |
| Aggregated / All threads | Aggregated / All threads | 9 (20%) |

**Table 3: Process concepts accounting for at least 10% of all tickets. In each concept the developers used a tool first, without looking for specific information.**

Using these concepts we identified three key patterns in how developers relate tools and information:

**Developers use tools first, without looking for specific information.** All 5 concepts reflected developers using a tool first. Out of 63 process codes that we created only seven codes (in four tickets) matched the "information-first" scenario, indicating that the process of performance debugging is for the most part exploratory. We found that even experienced developers often do not know where to begin in resolving a performance problem.

**Developers use tools to observe time series information.** Four of these process categories were related to the way developers acquired time series information, either per-thread or across all threads. Developers mainly used manual logs and gdb-based tools to collect time series information. Manual logging was used to collect time series behaviour for specific threads in 27 tickets (61% of all tickets), and used to collect time series behaviour of all threads in 12 tickets (27% of all tickets). GDB-based tools (gdb, gdbmon, pmp) were used to collect per-thread time series behaviour in 12 tickets (27% of all tickets).

**Classic CPU profilers are not frequently used.** Only one of our process categories reflected developers using standard CPU profilers like perf. These tools were used to observe aggregate program behaviour in 5 tickets (11% of all tickets).

Overall, this suggests that the process of performance comprehension is mostly about obtaining time series per-thread information using simple tools, such as application logging or gdb, in an exploratory fashion, without knowing first what to look for.

[7] This refers to unusually long-running function invocations

## 5.2 Reproducibility

Being able to reproduce a performance bug on a machine that has tools to diagnose and trace the issue is critical to root cause analysis and debugging. As we studied performance debugging processes when answering RQ3, we learned that reproducability was a concern. Overall, we created 7 codes relating to reproducibility, with at least one code appearing in 27 tickets (61%). Of these reproducibility tickets, 10 (37%) had instances where a developer *tried* to reproduce an issue but failed. This resulted in 13 (48%) of tickets coded with reproducibility containing directions or guidance on how to generally reproduce the performance problem. A common solution to reproducibility problems is to share a configuration file with other developers. However, only 7 (25%) tickets coded with reproducibility problems required sharing configuration files. Instead most reproducibility discussion centered around the right hardware/software versions to reproduce the issue. The importance and difficulty of reproducing performance issues in heterogeneous environments has been also noted in prior work [19, 46].

**Difficulty reproducing the software version.** In 18 (66%) of the tickets that had reproducibility challenges developers struggled to find the code exhibiting the problem. Developers often had several concurrent versions of their own or supporting software (i.e., MongoDB), and ensuring a developer had the right versions of all software that exhibited the performance problem was not trivial. For example:

> Could you confirm whether the version of MongoDB you are using includes that change?

> The version of master+develop was WT [commit X] on top of mongo [commit Y].

**Difficulty reproducing the hardware.** 10 tickets (37%) with reproducibility issues dealt with hardware conditions. WT developers primarily use cloud-based instances for development. Some hardware challenges were related to cloud use. For example:

> I would appreciate any tips on how to reproduce that behaviour. Perhaps I should run on the same AWS instance?

> What workloads and under what circumstances (like hardware) have you seen performance increases? My AWS box and the Jenkins have 8 cores and 32Gb.

Developers seemed reticent to re-use instances unless absolutely necessary. The reasons behind this is not clear to us, though the long time to setup a new environment might be a cause. Often, a developer 'lives in' a single development environment for a long time because they have added a number of customizations and it would be time consuming to transfer to a new instance. Instead, developers asked each other to run tests on instances they could not access, for example:

> Could you please retest on your instance again?

**Difficulty reproducing the configuration.** In 7 tickets (25%) developers manually shared configuration settings with each other. This involved developers copy/pasting details of their configurations and can be an error prone process because developers may not understand the necessary configuration settings required to reproduce an issue. However, we hesitate to say that this is a problem, as discovering that a little-known configuration detail has an effect on performance can be a significant step in diagnosing the problem. Another form of manual sharing was developers sharing code between each other by including it as a comment on a ticket.

**Dependency on cloud-based tools, such as CI systems.** One complex reproducibility ticket (in WT-2898) stood out to us. The issue was a hang during a performance benchmark run by Jenkins on a cloud instance. Reproduction was challenging in two ways: (1) The Jenkins server itself was not set up to easily debug benchmarks *in situ*; developers instead tried to set it up on their own machines. (2) The benchmark was challenging to configure and set up on the developer's machines, which caused spurious conclusions partway through debugging. This ticket was eventually solved through substantial effort, by reproducing the issue on the Jenkins cloud instance, where it was discovered to be a race condition.

## 5.3 Experiments

Developers run experiments to help identify the cause and potential solutions to a performance issue. As we were analyzing performance debugging processes, we found that much of them revolved around experimental details. We created 12 experiment-related codes with at least one appearing in 36 tickets (81%).

**Experimental comparisons.** Comparing code, workloads, hardware instances, or configurations to identify a performance issue was a common activity. Of tickets with some experiment, 29 (80%) contained a comparison. We classified experiments as comparisons when we saw at least two different versions of some artifact being compared, in contrast to e.g., experiments in which a program was tested with a given input (see workloads).

The most common experiment was a comparison between two different versions of code that already exists, such as a comparison between two branches or commits in git. This appeared in 20 tickets (55% of all tickets where an experiment was performed). This type of experiment helps with performance regressions in which the developer wants to narrow down what affected performance.

> When I was looking at differences between versions I found a good starting place.

> I compared changeset [commit X] (good performance) to [commit Y] (post-merge, bad performance).

The next most common experiment was a comparison between existing code, and a modified version of this code that the developer wrote. This appeared in 18 tickets (50% of all tickets that contained experiments). Developers used these experiment to determine if the bug was present after the change. The code change was sometimes a solution attempt, but it was also used to disable an (often necessary) component to help isolate the cause. For example:

> I made a small experimental change to the develop code to start all eviction threads in wt_evict_create.

> I ran [workload] with sweeps turned off comparing it to an identical run yesterday with sweeps on.

Comparisons between different software configurations (including changing the number of threads) appeared in 16 tickets (44% of tickets with experiments). Changing the number of threads was a common experiment, appearing in 8 tickets, while changing all other kinds of configurations, except threads, appeared in 11 tickets (some tickets used both kinds of experiments). Other kinds of

comparisons occurred less frequently. Comparisons between different workloads (test cases) appeared in 11 tickets (30% of all tickets with experiments), comparisons between hardware configurations appeared in 7 tickets (19%).

**Experiments with workloads.** Experiments involving running workloads to evaluate programs appeared in 22 tickets (61% of all tickets with experiments). Previously established workloads were the most popular, appearing in 19 tickets (52% of all tickets with experiments). A wide collection of workloads are often kept to help test the program in different ways, such as correctness or performance. These workloads are often automatically run by the CI system but were also run by WT developers manually to investigate an issue. These were used as well understood input and helped developers to validate theories about less understood program behaviour. For example:

> I've run the medium-btree.wtperf test on several changesets, both diagnostic and production builds and do not see any change in performance
>
> In case it is 'the other side of the same coin', the multi-btree-zipfian workload read throughput shows a similar 36% performance increase with those changes.
>
> I have started a [populate] and [workload] run on the SSD with mmap turned back on but no block compressor specified. We'll see what that looks like tonight.

Developers only developed new workloads in 4 tickets, drawing on the YCSB[14] benchmarks in 3 of these, and re-used workloads introduce by another ticket in 2 tickets. For example:

> [I am] constructing a workload that: — Inserts 16MB values from 10 threads in parallel — Configures a memory_page_max of 5MB — Does not do checkpoints — Does not use explicit transactions
>
> I am going to run this branch with MongoDB with some of the recent workloads that were having eviction issues. In particular the standalone version of the insert test from SERVER-23622

The final experiment that we noted was a measurement of how performance of a given benchmark varies. This appeared in 4 tickets (11% of all tickets with experiments). Developers used this to determine how consistently the program behaves across repeated runs on the same machine. Frequently this was used to dismiss or accept as valid an observed performance drop (or increase) in the context of the program's historical behaviour (often this information was not known to all developers). For example:

> Developer A: Is fruit_pop usually stable?
> Developer B: I would say fruit_pop is generally stable. I'd say 1% variation may be normal (which on a run of 4000 seconds is 40 seconds on either side), but over 3% feels outside that.
>
> When I started testing I realized that this workload has a lot of variance.

## 6 DISCUSSION AND IMPLICATIONS

**Profilers do not do the job.** Our case study uncovered that WT developers spent most effort on understanding how latency spikes and intermittent throughput drops relate to other events in the program. Off-the-shelf performance tools, namely profilers, were not helpful because developers relied on time series per-thread view of the execution, which profilers do not provide. Although the sample data collected by profilers could, in theory, be displayed across time/threads, the profiler sampling resolution is not fine enough to capture rare latency spikes. To obtain the needed information, developers used rudimentary tools, such as a debugger outfitted with shell scripts, but mostly application-level logging: a battery of log messages manually inserted in the code.

**Log analysis is often manual and exploratory.** For the most part developers manually sifted through raw log files. Manual log analysis works as long as the logs are not too big or if the developer knows what to search for. Unfortunately, as our analysis revealed, developers seldom know what to look for. Instead of starting with a concrete hypothesis and looking for a specific piece of information, developers usually just "throw" a tool at the problem to see what information turns up.

**Tools should support free exploration of execution traces.** Future performance tools must support the "exploratory" process of performance debugging. Requiring a developer to decide *a priori* which information to log is not productive, because developers do not know in advance what information they need. That is why, almost a quarter of issues in our case study necessitated new log messages, instead of using existing ones. Ideally, a performance tool would log, in a non-discriminating fashion, a wealth of information about functions, data structures and their timing, so that developers can freely explore the execution trace and discover new phenomena.

Designing such exploratory tools poses two challenges: large runtime overhead and an overwhelming amount of information to process. Overhead can be addressed with hardware support, such as Intel Processor Trace (PT) [3], or software techniques, such as Log20 [47] or Google's efficient tracing tools [1].

Examining massive amounts of information that tracing can produce is a challenge in and of itself. From conversations with Google engineers we learned that their performance traces are petabytes in size! There are many visual analysis tools for large execution traces, such as ExtraVis [16], SyncTrace [26], Revel [22] and Trümper's *thread overview / sequence view* [40], TraceDiff [41] and Lvis [44]. IBM's Zinsight [17] and Google's in-house tools [1] are performance debugging tools used in practice that explicitly track outlier events. However, they do not support completely free exploration, requiring the developer to specify an execution window to look at. This concern was related to us by one WT developer:

> Identifying interesting time periods is a definite issue with the *ABC* tool[8]. We generally rely on users to indicate a relatively small time window to facilitate investigation. I dream of being able to do some automated pattern matching on the data in order to isolate interesting time periods in the future.

**Developers could take better advantage of cloud resources.** Despite running a cloud-based CI system and working primarily on cloud instances, WT developers still only used a few personal instances for development and testing. When an issue appeared on a specific hardware configuration, it was often left up to a particular user to re-run the experiment after others have pushed changes. Essentially, developers treated their cloud instances as personal development machines. Even the WiredTiger CI system in this case was apparently only able to to test one branch of code at a time.

[8] The developer referred to a specific in-house tool that shows statistics over time.

We think developers could take better advantage of the immediate availability of cloud resources, assuming they have a sufficient budget. For example, to dissuade developers from local and personal customizations, developers could maintain environment configuration as part of version control. To reduce hardware-related reproducibility challenges: developers should be able to easily start new instances with the standardized environment. And, to enable easier instance sharing, developers can maintain common authentication between all of their instances.

**Supporting performance experimentation.** Experiments yield valuable information, but only when the workloads are well understood. In WT's case a common catalog of workloads with well-documented properties and behaviors seemed to improve reproducibility and accuracy of interpretation. But, developers often manually re-ran and cataloged experimental output. We believe that existing CI systems can provide better support, potentially running experiments pre-emptively and along different dimensions to establish baselines for interpreting performance degradations (similar to what was described by Nguyen et al. [32]).

We hope that our findings on *how* developers work to address performance issues will spur new ideas for tools in this area.

## 7 RELATED WORK

Several studies characterize performance bugs. For example, Zaman et al. [46] compare the qualitative difference between performance bugs and non-performance bugs in Mozilla and Google Chrome along four dimensions: impact, context, fix, and fix validation. The study finds that developers and users face problems in reproducing performance bugs and have to spend more time discussing performance bugs than other kinds of bugs.

Likewise, Nistor et al. [33] study how performance bugs are discovered, reported to and fixed by developers, and compare the results with those for non-performance bugs. The authors consider bugs from Eclipse JDT, Eclipse SWT, and Mozilla. They find that fixing performance bugs is more difficult than fixing non-performance bugs and that, compared with non-performance bugs, a larger percentage of performance bugs are discovered through code reasoning than by users observing the negative effects of the bugs.

Song et al. [39] follow up on these findings and find that user-reported performance problems are observed through comparison, including comparisons within one code base with different inputs and comparisons across multiple code bases. Performance problems are also often reported as comparisons, including both good and bad inputs. In addition, like Nistor et al. [33], this study also finds that performance problems take a long time to diagnose.

To better understand how developers locate performance bugs, navigate through the code, understand the program, and communicate the detected issues, Baltes et al. [10] perform an observational study with twelve developers who are trying to fix performance bugs in two open source projects. The developers worked with a specific profiling and analysis tool that visually depicts runtime information in a list representation and is embedded into the source code view. The study concludes that fixing a performance bug is a code comprehension and navigation problem and that flexible navigation features are needed. Program comprehension implicates several of our information codes (Table 1). However, our study considers performance information more broadly (e.g., relation of program activity and OS activity) and also reports on tools and processes that are implicated in performance comprehension.

Jin et al. [23] study 109 performance bugs that are randomly sampled from five large software projects. They discover that more than one quarter of the bugs are caused by developers misunderstanding the workload or performance-related characteristics of certain functions and almost half of the bugs require large-scale input to be reproduced. We also observed the importance of reproducibility in our dataset, and we characterized several corresponding difficulties (Section 5.2). They also found that developers employed one of three bug-tracing strategies: changing a function-call sequence, conditionally skipping code units which do not always perform useful work, and changing program parameters.

Similarly, Liu et al. [29] study 70 performance bugs from eight popular Android apps, focusing on bug characteristics, such as GUI lagging and memory bloat. They found that over a third of these bugs manifest themselves only after special user interactions and that performance bug detection requires manual effort. This is consistent with our findings (Section 4): we found that WT developers expended substantial manual effort on performance comprehension, frequently using custom and manual tools, such as logging.

Our work complements these prior studies: instead of contrasting performance and non-performance debugging, how performance bugs are identified and reported, and what are the code patterns that can help identify performance bugs, we investigate what kind of information developers need to locate performance bugs, what tools they use in practice, and what processes they follow.

An extensive body of prior work considers developers' information needs [27, 28], adoption of certain tools [24], and software processes [21] more generally. By contrast, our focus is specifically on *performance* comprehension. To the best of our knowledge, this is the first study of this kind.

## 8 CONCLUSION

Performance debugging is a task with its own processes, tools, and information requirements [35]. We presented an empirical case study of performance-related issues in the WiredTiger project, a high-performance database engine. We studied developer discussions threads in 44 issues and used grounded theory to characterize the concepts related to information collected, tools used, and the process the developers followed in their investigations.

Overall, we found that these performance-focused developers spend a lot of their time chasing latency spikes and throughput drops, but are not supported by existing tools. Instead, they examine manual logs or the raw output of rudimentary call stack samplers. Developers keep a number of workloads to help monitor the performance of their system, but often have difficulty reproducing performance issues in a cloud environment. To diagnose performance issues, developers perform exploratory differential experiments to understand what recent changes to code could have caused the problem. They also frequently use tools without knowing what information might be relevant to the problem.

# REFERENCES


[1] Datacenter Computers: Modern Challenges in CPU Design. http://www.pdl.cmu.edu/SDI/2015/slides/DatacenterComputers.pdf.

[2] DTrace. http://dtrace.org/blogs/about.

[3] Intel Processor Trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[4] Jenkins. https://jenkins.io/.

[5] Poor Man's Profiler. https://poormansprofiler.org.

[6] RotateRight Zoom. https://en.wikipedia.org/wiki/RotateRight_Zoom.

[7] The DINAMITE Toolkit. https://dinamite-toolkit.github.io.

[8] WiredTiger operation tracking tutorial. https://github.com/wiredtiger/wiredtiger/wiki/WiredTiger-operation-tracking.

[9] J. Aranda and G. Venolia. The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2009.

[10] S. Baltes, O. Moseler, F. Beck, and S. Diehl. Navigate, Understand, Communicate: How Developers Locate Performance Bugs. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ESEM, 2015.

[11] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang. XRay: A Function Call Tracing System. Technical report, 2016. A white paper on XRay, a function call tracing system developed at Google.

[12] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated Test Generation for Worst-case Complexity. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2009.

[13] B. Chen, Y. Liu, and W. Le. Generating Performance Distributions via Probabilistic Symbolic Execution. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2016.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC, 2010.

[15] J. Corbin and A. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage Publications, Inc., 3rd edition, 2008.

[16] B. Cornelissen, D. Holten, A. Zaidman, J. J. Van Wijk, and A. Van Deursen. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *Proceedings of the International Conference on Program Comprehension*, ICPC, 2007.

[17] W. De Pauw and S. Heisig. Zinsight: a Visual and Analytic Environment for Exploring Large Event Traces. In *Proceedings of the International Symposium on Software Visualization*, SOFTVIS, 2010.

[18] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[19] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2015.

[20] B. Glaser and A. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Observations (Chicago, Ill.). Aldine de Gruyter, 1967.

[21] G. Gousios, M. Pinzger, and A. v. Deursen. An Exploratory Study of the Pull-based Software Development Model. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2014.

[22] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann. Combing the Communication Hairball: Visualizing Parallel Execution Traces Using Logical Time. In *IEEE Transactions on Visualization and Computer Graphics*, 2014.

[23] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, 2012.

[24] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the International Conference on Software Engineering*, ICSE, 2013.

[25] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, 2011.

[26] B. Karran, J. Trümper, and J. Döllner. SYNCTRACE: Visual Thread-Interplay Analysis. In *Proceedings of the Working Conference on Software Visualization*, VISSOFT, 2013.

[27] A. J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2007.

[28] T. D. LaToza and B. A. Myers. Developers Ask Reachability Questions. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2010.

[29] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2014.

[30] Michael Cahill. A Technical Introduction to WiredTiger. https://www.slideshare.net/mongodb/mongo-db-wiredtigerwebinar.

[31] MongoDB. MongoDB Acquires WiredTiger Inc. https://www.mongodb.com/press/wired-tiger.

[32] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. An Industrial Case Study of Automatically Identifying Performance Regression-causes. In *Proceedings of the Working Conference on Mining Software Repositories*, MSR, 2014.

[33] A. Nistor, T. Jiang, and L. Tan. Discovering, Reporting, and Fixing Performance Bugs. In *Proceedings of the Working Conference on Mining Software Repositories*, MSR, 2013.

[34] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2013.

[35] J. Ousterhout. Always Measure One Level Deeper. *Commun. ACM*, 61(7):74–83, June 2018.

[36] R. Padhye and K. Sen. Travioli: A Dynamic Analysis for Detecting Data-structure Traversals. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2017.

[37] M. Pradel, M. Huggler, and T. R. Gross. Performance Regression Testing of Concurrent Classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, 2014.

[38] V. Singh, R. Gupta, and I. Neamtiu. MG++: Memory graphs for analyzing dynamic data structures. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, SANER, 2015.

[39] L. Song and S. Lu. Statistical Debugging for Real-world Performance Problems. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA, pages 561–578, 2014.

[40] J. Trümper, J. Bohnet, and J. Döllner. Understanding Complex Multithreaded Software Systems by using Trace Visualization. In *Proceedings of the International Symposium on Software Visualization*, SOFTVIS, 2010.

[41] J. Trümper, J. Döllner, and A. Telea. Multiscale Visual Comparison of Execution Traces. In *Proceedings of the International Conference on Program Comprehension*, ICPC, 2013.

[42] WiredTiger. WiredTiger GitHub. https://github.com/wiredtiger/wiredtiger.

[43] WiredTiger. WiredTiger Jira. https://jira.mongodb.org/projects/WT/issues.

[44] Y. Wu, R. H. Yap, and F. Halim. Visualizing Windows System Traces. In *Proceedings of the International Symposium on Software Visualization*, SOFTVIS, 2010.

[45] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2012.

[46] S. Zaman, B. Adams, and A. E. Hassan. A Qualitative Study on Performance Bugs. In *Proceedings of the IEEE Working Conference on Mining Software Repositories*, MSR, 2012.

[47] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements Under Specified Overhead Threshold. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, 2017.