Making Sense of Multi-threaded Application Performance at Scale with NonSequitur

AUGUSTINE WONG, University of British Columbia, Canada
PAUL BUCCI, University of British Columbia, Canada
IVAN BESCHASTNIKH, University of British Columbia, Canada
ALEXANDRA FEDOROVA, University of British Columbia, Canada

Modern multi-threaded systems are highly complex. This makes their behavior difficult to understand. Developers frequently capture behavior in the form of program traces and then manually inspect these traces. Existing tools, however, fail to scale to traces larger than a million events.

In this paper we present an approach to compress multi-threaded traces in order to allow developers to visually explore these traces at scale. Our approach is able to compress traces that contain millions of events down to a few hundred events. We use this approach to design and implement a tool called NonSequitur.

We present three case studies which demonstrate how we used NonSequitur to analyze real-world performance issues with Meta's storage engine RocksDB and MongoDB's storage engine WiredTiger, two complex database backends. We also evaluate NonSequitur with 42 participants on traces from RocksDB and WiredTiger. We demonstrate that, in some cases, participants on average scored 11 times higher when performing performance analysis tasks on large execution traces. Additionally, for some performance analysis tasks, the participants spent on average three times longer with other tools than with NonSequitur.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*; **Software performance**; • **Human-centered computing** → *Visualization systems and tools.*

Additional Key Words and Phrases: Performance debugging, Multi-threaded Applications, Outlier events, Runtime trace visualization

ACM Reference Format:
Augustine Wong, Paul Bucci, Ivan Beschastnikh, and Alexandra Fedorova. 2024. Making Sense of Multi-threaded Application Performance at Scale with NonSequitur. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 353 (October 2024), 30 pages. https://doi.org/10.1145/3689793

1 Introduction

Debugging performance issues in multi-threaded applications is a frequent and challenging task for software developers. For example, a case study on Firefox found that performance bugs take a longer time to fix, are fixed by more experienced developers, and require changes to larger parts of the code than non-performance bugs[48]. Another study also found that performance bugs were more difficult to fix than non-performance bugs, suggesting that developers need better tool support for performance debugging [35].

One of the challenges that developers encounter when debugging performance is that they often need to analyze large execution traces to understand performance issues [19]. Execution traces

Authors' Contact Information: Augustine Wong, University of British Columbia, Vancouver, Canada, augustinew@ece.ubc.ca; Paul Bucci, University of British Columbia, Vancouver, Canada, pbucci@cs.ubc.ca; Ivan Beschastnikh, University of British Columbia, Vancouver, Canada, bestchai@cs.ubc.ca; Alexandra Fedorova, University of British Columbia, Vancouver, Canada, sasha@ece.ubc.ca.

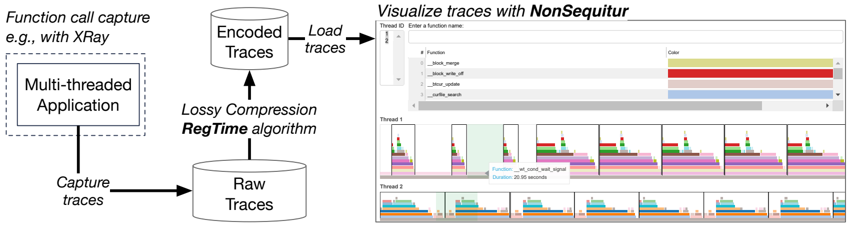
ACM 2475-1421/2024/10-ART353
https://doi.org/10.1145/3689793

are time-stamped operation or function logs, which are generated at runtime and capture each event that a thread executed over time. A typical execution trace of a multi-threaded application would capture when a particular thread entered a function and when a particular thread exited the function. There are several reasons why developers frequently need to analyze large execution traces when debugging performance:

(1) Execution traces contain the information that developers need to debug two types of performance problems: those caused by "outlier events" and those caused by threads being blocked or delayed. We define an *outlier event* as: *a function call invocation or time between function calls that lasts an unusually long time.* An example of an outlier event is a piece of code which runs infrequently, but each execution of this code takes a long time [27]. Although occurring infrequently, performance issues caused by outlier events are a critical type of performance problem because they represent the worst-case system behavior. Performance problems caused by threads being blocked or delayed are also a critical type of performance issue because they constitute a majority of software performance bugs [24]. To resolve these types of issues, developers need to analyze execution traces to find outlier events, understand what threads are doing over time, and identify correlations in behavior across threads.
(2) Developers do not always know a priori what information is useful in diagnosing performance issues [19]. As a result, they collect large execution traces in the hope of capturing key pieces of data that will help them understand the performance degradation [19].
(3) When developers try to understand the performance of their applications, they can use profilers to collect performance data during program execution. Popular profilers include OProfile, Perf, HPCToolkit, and Intel VTune [47]. But profilers do not sample at a high enough rate to catch outlier events. As stated by one developer of WiredTiger, a multi-threaded open-source storage engine, *“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”*[19].

Large execution traces which developers analyze when debugging performance could contain millions of recorded function invocations across multiple threads. One option that developers have to analyze large execution traces is to employ algorithms to extract important information from the execution traces. Such algorithms can remove redundant information from execution traces [23]; detect “phases of execution” [10, 20, 38, 45]; and find interesting patterns of behavior [11, 33]. However, these algorithms are not designed to assist software developers with finding and investigating the causes of outlier events. Furthermore, to the best of our knowledge, most of these algorithms have not been implemented into trace analysis tools which software developers can use. Another option is to use trace visualization tools. However, a survey from 2014 found that the majority of trace visualization tools unable to visualize very large execution traces containing millions of function invocations [25]. Consequently, even visualization tools that are designed to assist developers with performance debugging [30, 43] cannot be used with large execution traces. On the other hand, other trace visualization tools that can handle large execution traces were not designed to assist developers with analyzing performance bugs caused by outlier events [16, 17].

We present RegTime, an algorithm to compress execution traces by removing information not important for performance comprehension [19]. RegTime takes as its input a large execution trace and outputs a compressed version of the trace, with enough information for developers to find outlier events, understand what threads are doing over time, and identify correlations in behavior across threads. We also describe NonSequitur, an interactive visualization tool for developers to study traces compressed with RegTime (see Figure 1). Our current implementation of NonSequitur compresses multi-threaded execution traces that contain the call stacks of executed functions. But



#	Function	Color
0	__block_merge	🟨
1	__block_write_off	🟥
2	__btcur_update	🟫
3	__curfile_search	🟦

Function: __wt_cond_wait_signal
Duration: 20.95 seconds

Fig. 1. NonSequitur workflow.

NonSequitur can also be adapted to compress logs containing any type of event, as long as those events have start times and durations. For example, NonSequitur can be adapted to compress a log of network operations executed by different nodes in a distributed system.

In summary, we make three contributions:

- ★ We present the RegTime algorithm to assist developers with analyzing large execution traces to debug performance issues caused by outlier events and by blocked threads.
- ★ We introduce NonSequitur, a tool that visualizes the compressed execution traces generated by RegTime. Consequently, NonSequitur is able to visualize large execution traces within the width of a computer screen[1].
- ★ We present three case studies in which we used NonSequitur to analyze real-world performance issues with WiredTiger and RocksDB, two complex database backends. We also conduct a user study with 42 software developers to evaluate how effective NonSequitur is at helping developers diagnose performance bugs as compared with existing tools.

2 Background and Related Work

Our goal with designing RegTime and NonSequitur was to assist software developers with analyzing performance bugs captured in large execution traces. To understand how to design these tools, we examined empirical studies that characterised performance debugging. Prior studies focused on comparing the processes of fixing performance and non-performance bugs [35, 48] or characterizing performance bugs [32, 40]. But the findings of these studies were not useful in helping us to design RegTime or NonSequitur because they did not analyze what developers do to *diagnose* those bugs.

One study, however, examined *performance comprehension*, defined as the tools, information, and processes developers use to investigate performance issues. This study used a JIRA database of tickets for WiredTiger, the MongoDB storage engine, focusing on performance issues [19]. The study concluded that WiredTiger developers spent most of their performance debugging time investigating latency spikes and throughput drops. Developers also spend time investigating threads that were blocked or delayed. To that end, developers analyze log files with thread behavior over time, often focusing on correlations between latency spikes, blockages, and thread activity. WiredTiger developers frequently manually look through logs, which is impractical for large log files. This study motivated us to design RegTime and NonSequitur.

The above study emphasized the importance of timing information: to understand what threads do over time, developers need to understand when events in an execution begin and end. To identify outlier events, developers need to know when events occur and the duration of these events. To correlate outliers and blockages with the activities of threads, developers must be able to find the interval during which these occurred and observe what threads were doing during this interval.

[1]NonSequitur is open source: https://github.com/auggywonger/nonsequitur_vis

Table 1. Summary of prior work

Tool	Technique	Large traces	Considers timing	Finds outliers
Compress. Framework [23]	Trace Compression Alg	✓	✗	✗
Phase Finder [38]	Phase Detection Alg	✓	✗	✗
Sabalan [11]	Pattern Detection Alg	✓	✗	✗
Zinsight [17]	Trace Vis	✓	✓	✗
TraceViz [16]	Trace Vis	✓	✓	✗
SyncTrace [30]	Trace Vis	✗	✓	✓
ExtraVis [15]	Trace Vis	✗	✓	✗
TraceDiff [44]	Trace Vis	✗	✓	✗
Sites' Vis Tools [2, 2]	Trace Vis	✗	✓	✓
NonSequitur (this paper)	Trace Compression and Vis	✓	✓	✓

What developers do when diagnosing performance bugs is different from the activities of *program comprehension* [37]. Many existing tools are designed for program comprehension, despite the fact that performance bugs are generally harder to resolve than non-performance bugs [35, 48]. Table 1 summarizes the features of the existing tools.

Prior work has developed algorithms to help developers analyze execution traces. These algorithms summarize the activities captured within the execution traces by removing repeated calls due to loops [23] or by dividing the content of an execution trace into fragments that correspond to execution phases [38]. These algorithms can help explain the functionality of the system. However, they do not consider the timing information captured in the execution traces, treating unusually long function calls the same as those of expected duration. Therefore, these algorithms cannot locate outlier events or show what happened when the outlier events appeared. Other algorithms were designed to find patterns of behavior in execution traces. Sabalan is a trace visualization tool uses a bioinformatics-inspired algorithm to display execution behavior that appears repeatedly in multiple traces [11]. The algorithm treats execution traces like DNA sequences, where recurring patterns in DNA serve biological functions. Sabalan is effective at helping developers with program comprehension tasks such as feature location or understanding how a system works. But Sabalan's algorithm ignores timing information and has limited use in performance comprehension.

In contrast, many trace visualization tools do display the behavior of execution traces over time. However, these trace visualization tools are not suited for analyzing large execution traces to diagnose performance bugs caused by outlier events or blockages. Trace visualization tools typically fall into one of three categories: (1) The tool can visualize large execution traces, but does not support performance comprehension tasks. (2) The tool can support performance comprehension tasks, but cannot visualize large execution traces. (3) The tool neither supports performance comprehension tasks nor is capable of visualizing large execution traces.

Zinsight is a tool to visualize traces from the IBM System Z [17]. TraceViz is a visualization framework based on observations made in real situations at STMicroelectronics, a semi-conductor company [16]. Both tools provide a timeline showing what is happening in an execution over time. However, the main purpose of these tools is to help developers to identify patterns of behavior in execution traces. Zinsight has a Sequence Context view that shows the paths of execution that lead to specific events. TraceViz's timeline view of the execution was designed to show periodic temporal behavioral patterns. These tools are not designed for investigating performance bugs caused by outlier events or blockages.

SyncTrace visualizes thread behavior over time in sufficient detail that it could theoretically be used for performance comprehension tasks [30]. SyncTrace has a panel residing at the top for visualizing the timeline of a thread. Users can interact with the panel to see correlations in behavior between the thread displayed in the panel and other threads. But SyncTrace was only demonstrated to be capable of visualizing threads executing hundreds of thousands of events. Large execution traces, by contrast, could capture *millions* of events per thread.

ExtraVis [15] is a trace visualization tool designed explicitly to support program comprehension. It displays a vertical timeline of events with the purpose of helping developers with identifying the major phases of an execution. ExtraVis was not shown to be capable of visualizing large execution traces. TraceDiff [44] allows developers to visually compare two execution traces. TraceDiff is only capable of visualizing execution traces containing hundreds of thousands of function calls.

Trace compression by RegTime is a form of *performance summarization*, as introduced by Attariyan et al. [12]. Unlike this prior work, however, we do not rely on record and replay, which is an expensive technique. And our goal with compression is to present a coherent and scalable visualization for developers to explore.

Dick Sites described methods for finding software bugs during his career at Google [42]. His visualization tools display [2, 3] various events across time, such as RPC, user/kernel transitions, and acquired locks. However, these visualizations are constrained by their inability to present a large number of events on a single screen, necessitating either small visual features or requiring paging across multiple screens.

Tail latency and stragglers are extensively studied in the distributed systems community [13, 18]. Research has considered these issues in specific sub-domains, including microservice architectures [22, 49], service meshes [50] and analytics systems [36]. A variety of ways to deal with tail latency have also been proposed, such as new schedulers and operating systems [28, 39], as well as increasingly advanced forms of distributed tracing to collect and organize latency and dependency information across systems [21, 29, 34, 41]. Some of these systems include anomaly detection algorithms [26, 31], support aggregation of traces, and provide visualization tools to help developers understand the captured latency information. However, we are not aware of trace compression and scalable call-stack-style trace visualization efforts, which are the core contributions of our work.

3 NonSequitur Design

In this section we present the design of NonSequitur. Two requirements drove the design:

(1) Turn a multi-threaded execution trace into a representation that can be practically visualized. To support traces with millions of events, this logically leads us to the next requirement:
(2) Compress a multi-threaded execution trace in a way that retains information required by software developers to debug performance.

We satisfy both requirements via the new execution trace compression algorithm, RegTime, which we present in Sections 3.1 and 3.2. We build NonSequitur, a new tool that visualizes traces compressed by RegTime. We present NonSequitur in Sections 3.3 and 3.4.

3.1 Trace Compression

We focus on traces containing time-stamped records of function calls, though traces with other kinds of log records could also be potentially processed. We are not the first to propose a technique for compressing execution traces, so as to make them more understandable. For instance, Hamou-Lhadj et al. [23] created a framework for lossless trace compression. Lossless compression techniques achieve compression such that the exact original dataset can be recreated from its compressed version. Lossy compression techniques produce better compression ratios by discarding information, but the original dataset cannot be recreated. Consequently, there is a trade-off between better compression ratios and retaining information from the original dataset.

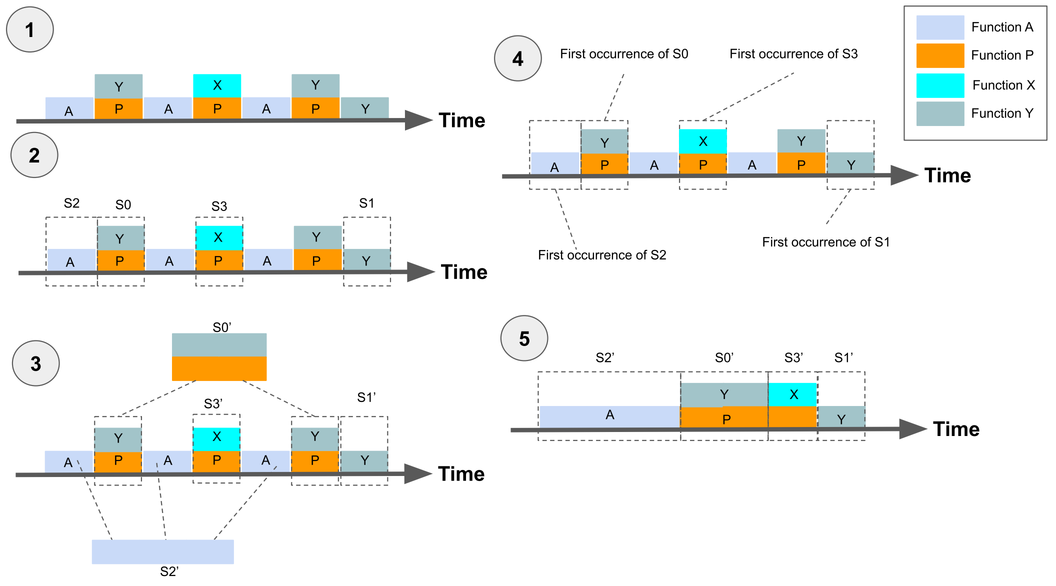


Fig. 2. (1) A compressible function sequence. (2) The callstacks which make up the compressible function sequence. (3) The same callstacks in the compressible function sequence are merged together. (4) The first occurrences of each unique callstack in the compressible function sequence. (5) The compressed callstacks.

To compress traces into a representation that can be visualized in a small amount of screen real estate, RegTime must reduce a trace with millions of events down to a few thousand. Given a compression ratio $n_{initial}/n_{after}$, where $n_{initial}$ is the initial number of function records in the trace and n_{after} is the number of function records in the trace after compression, RegTime must to achieve a compression ratio at least 1,000. However, the lossless compression proposed by Hamou-Lhadj et al., the best technique known to us, demonstrated a maximum compression ratio of around 41. Therefore, we chose to design RegTime as a lossy compression algorithm.

3.2 The RegTime Algorithm

We use Figure 2 and Algorithms 1 and 2 to explain how RegTime works. We use the term *RegTime expression* to refer to a function sequence which has been compressed by RegTime.

Figure 2 visualizes the key steps of the algorithm on an example. Figure 2(1) shows a compressible function sequence, and Figure 2(2) shows that this compressible function sequence has four unique callstacks: S0, S1, S2, and S3. To create the compressed callstacks (*compressed_seq* in Algorithm 2) RegTime first merges identical callstacks. In Figure 2(3), RegTime creates four groups of merged callstacks: S0', S1', S2', and S3'. For example, RegTime formed S2' by merging the three instances of S2. Next, RegTime arranges the groups according to the order in which the first occurrences of each unique callstack appears. Figure 2(4) shows the first occurrence of each unique callstack, and Figure 2(5) lists the final ordering. We now present the full algorithm in detail.

RegTime iterates through the timeline of each thread captured in a multi-threaded execution, deciding which function sequences invoked by a thread can be compressed into a RegTime expression (Algorithm 1, lines 5 - 6). There may be cases when RegTime decides none of the function

Algorithm 1 CompressExecution

```
procedure CompressExecution($T$)
    RegTimes = [][]  ▷ [t][[expr,s,e] is a RegTime expr from positions s to e of thread t's exec
    for $thread\_id$ in $T$ do  ▷ Iterate through all threads in trace T
        $fncall\_index \leftarrow 0$
        while $fncall\_index < T[thread\_id].length$ do  ▷ Iterate through thread's timeline
            $(expr, start\_pos, end\_pos) \leftarrow$ FindRegTimeExpr($fncall\_index, T[thread\_id]$)
            if $expr$ is nil then
                ▷ No RegTime expr found starting at $fncall\_index$
                $RegTimes[thread\_id].append([nil, fncall\_index, T[thread\_id].length - 1])$
                break
            end if
            if $start\_pos > fncall\_index$ then
                ▷ Found RegTime expr interval does not capture $fncall\_index$
                $RegTimes[thread\_id].append([nil, fncall\_index, start\_pos - 1])$
            end if
            ▷ Record found RegTime $expr$
            $RegTimes[thread\_id].append([expr, start\_pos, end\_pos])$
            $fncall\_index \leftarrow end\_pos + 1$
        end while
    end for
    return RegTimes
end procedure
```

sequences executed by a thread need to be compressed (Algorithm 1, line 9). Alternatively, RegTime can compress some function sequences executed by a thread while leaving other sequences uncompressed (Algorithm 1, lines 12 - 15).

For a given thread, RegTime iterates through the function call records corresponding to that thread to find the start of a compressible function sequence (Algorithm 2, lines 6 - 10). To simplify our explanation of RegTime, we treat each function call record as including a start and end time of the function call, stack depth of the call, and call duration.

RegTime considers a function sequence to be compressible if it satisfies three conditions:

- Condition 1: The function sequence is comprised of function calls that have duration less than *CallDurationThresh*.
- Condition 2: The time between two consecutive function calls is less than *CallGapThresh*.
- Condition 3: The time interval occupied by the function sequence does not take up more than a *TotalTimeFractionThresh* of a thread's total execution time.

Condition 1 ensures that RegTime compresses those portions of a thread's behavior that have a high density of function calls. These regions are ideal for compression. Condition 1 also ensures that RegTime does not lose outlier calls with long duration. Additionally, because an outlier event can be an unexpectedly long period of time between two consecutive calls, condition 2 ensures that these periods are not lost. Finally, Condition 3 guarantees that RegTime will use multiple RegTime expressions to represent the behavior of a thread, even if all the calls by that thread satisfy Conditions 1 and 2.

Users can tune *CallDurationThresh*, *CallGapThresh*, and *TotalTimeFractionThresh* to create a custom definition of what an unusually long function or interval looks like for their particular system and performance issue. Once the algorithm has found the beginning of a compressible

Algorithm 2 FindRegTimeExpr Function

```
1:  procedure FindRegTimeExpr($fncall\_index, TTrace$)
2:      $state \leftarrow$ Skip                     ▷ Stay in Skip state until we find a compressible fn call
3:      $compressed\_seq = []$
4:      for $i = fncall\_index$ to $TTrace.length$ do          ▷ Iterate through the thread's fn calls
5:          $is\_longfn \leftarrow TTrace[i].duration > CallDurationThresh$      ▷ Whether call at $i$ has an outlier duration
6:          if $state$ is Skip and not $is\_longfn$ then
7:              $state \leftarrow$ Compress                                   ▷ Start compressing
8:              $expr\_start\_pos \leftarrow i$
9:              $expr.starttime \leftarrow TTrace[expr\_start\_pos].starttime$
10:             $compressed\_seq.append(TTrace[i])$
11:         else if $state$ is Compress then
12:             $expr\_end\_pos \leftarrow i - 1$
13:             $expr.endtime \leftarrow TTrace[expr\_end\_pos].endtime$
14:             $is\_long\_idle \leftarrow (TTrace[i].starttime - TTrace[i-1].endtime) > CallGapThresh$
15:             $is\_long\_regt \leftarrow$ expr longer than $TotalTimeFractionThresh$ of thread exec time
16:             if $is\_longfn$ or $is\_long\_idle$ or $is\_long\_regt$ then
17:                 ▷ Reached end of compressable sequence. Compress sequence so far and return.
18:                 $merged\_callstacks = merge\_same\_callstacks(compressed\_seq)$
19:                 $sorted\_merged\_callstacks = sort\_by\_first\_occurrence(merged\_callstacks)$
20:                 $expr.compressed\_callstacks = sorted\_merged\_callstacks$
21:                 return $expr, expr\_start\_pos, expr\_end\_pos$              ▷ Return RegTime expr
22:             end if
23:             $compressed\_seq.append(TTrace[i])$    ▷ Add TTrace[i] and continue compressing
24:         end if
25:     end for
26:     return $nil, nil, nil$                                      ▷ No RegTime expr found
27: end procedure
```

function sequence, it continues to iterate through the function call records until it determines that it has reached the end of a compressible function sequence. Then, RegTime generates a RegTime expression to represent the compressible function sequence (Algorithm 2, lines 17 - 22). Each RegTime expression would be visualized separately on a timeline by NonSequitur (as explained next). If a thread with consistent and repetitive sequence of function calls is compressed with only a single RegTime expression, then the visualization would include only a single visual element. We reasoned that software developers would better understand how a thread's behavior changes over time if we showed it with multiple visual elements. Hence, we use multiple RegTime expressions.

A RegTime expression has three attributes allowing it to represent a function sequence: start and end times for the function sequence and *compressed_callstacks*: a sequence of sorted groups of callstacks captured in the function sequence period (Algorithm 2, line 20). Figure 2 illustrates how RegTime generates these *compressed_callstacks*.

3.3 RegTime Visual Encoding of RegTime Expressions

We use the term *RegTime Visual Encoding* to refer to the visualization technique NonSequitur uses to display RegTime expressions. Figure 3 compares the function sequence from Figure 2(1) to the RegTime Visual Encoding of RegTime expression for this sequence from Figure 2(5). NonSequitur uses a black box to denote the start and end times of the expression. It uses stacked rectangular glyphs to encode the *compressed_callstacks* in the RegTime expression.

If we consider the RegTime Visual Encoding in Figure 3(2), we can discern the following information about the function sequence it represents:

✓ **Caller-callee relationships.** RegTime preserves the caller-callee relationships between functions. The RegTime Visual Encoding shows function X is always called by function P.

✓ **Order and functions.** The function sequence consists of the functions A, P, X, and Y. RegTime preserves all the functions in the trace, unless their cumulative duration over the entire trace is negligible. RegTime does *not* preserve the total order of the callstacks because it sorts them by the first occurrence within the encoded period. E.g., callstack A is shown first in the encoded sequence, callstack P, Y is shown second, and callstack P, X is shown third. If the program also happened to invoke P, Y *after* P, X (in addition to invoking it after A), the visual encoding would not change. Every callstack is visualized only once in the encoding period regardless of how many times it appeared, meaning only the order of the first occurrence is preserved.

✓ **Encoded interval.** RegTime preserves the start and end time of the encoded function sequence. In this example, the function sequence started at 0 ns and ended at 7 ns.

✓ **Cumulative duration of functions.** RegTime uses the cumulative duration of all callstacks in the encoded sequence to determine the relative width of the corresponding glyphs. For example, given that the cumulative duration of function A in the sequence is 3 ns, function A's glyph takes about 3/7 of the coresponding glyph.

✓ **Loose causal relationship.** Since the order in which the groups of merged callstacks appear in a RegTime expression corresponds to the order that the first occurrences of the callstacks appear in the function sequence, we have some sense of the causal relationships between functions executed by the thread.

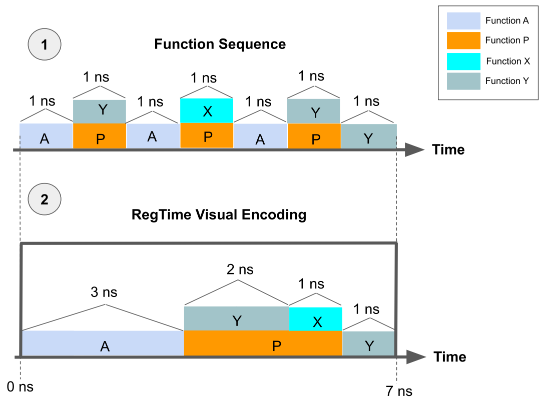


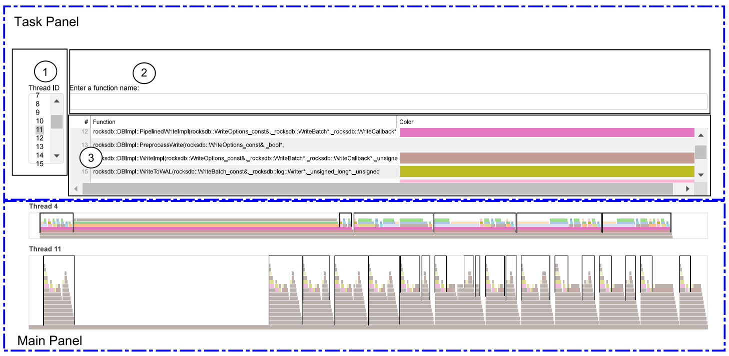
Fig. 3. (1) The function sequence from Figure 2(1). (2) The RegTime Visual Encoding of the corresponding RegTime expression.

For example, the RegTime Visual Encoding shows that function P appears after function A. Strictly speaking, this indicates that an occurrence of function P appeared after an occurrence of function A *at least once* in the function sequence. However, because the behavior of threads is highly repetitive [14], function P likely always appeared after function A. Since a function's name typically captures the function's purpose [9], users can further ascertain causal relationships from the function names.

However, due to the lossy compression, we are unable to discern the following from a RegTime Visual Encodings:

✗ **Individual function calls.** Because invocations of the same function are merged together, we cannot tell when individual function calls were executed or what their durations were.

✗ **Precise causal relationship.** Two merged functions appearing next to each other within a RegTime Visual Encoding does not always mean there is a causal relationships between these functions. Figure 3(2) hows function X appearing after function Y, even though Figure 3(1) shows that there is no causal relationship between these functions.



#	Function	Color
12	rocksdb::DBImpl::PipelinedWriteImpl(rocksdb::WriteOptions_const&,_rocksdb::WriteBatch*,_rocksdb::WriteCallback*	🟪
13	rocksdb::DBImpl::PreprocessWrite(rocksdb::WriteOptions_const&,_bool*,	
	ksdb::DBImpl::WriteImpl(rocksdb::WriteOptions_const&,_rocksdb::WriteBatch*,_rocksdb::WriteCallback*,_unsigne	🟫
15	rocksdb::DBImpl::WriteToWAL(rocksdb::WriteBatch_const&,_rocksdb::log::Writer*,_unsigned_long*,_unsigned	🟨

Fig. 4. NonSequitur is divided into main panel and the task panel. The main panel is where NonSequitur shows what threads are doing over time. The task panel provides users with three options for exploring the visualization: (1) The multi-select widget in NonSequitur which allows users to select thread ID numbers. After users select one or more thread IDs, NonSequitur displays only those threads which were assigned those thread IDs. (2) The search bar in NonSequitur which allows users to enter the name of a function. NonSequitur displays only those threads which have called that function. (3) A legend which shows users the mapping between colours and functions.

3.4 NonSequitur Tool

We developed NonSequitur to visualize the encodings generated by RegTime. The tool is open source[2]. NonSequitur ingests traces that are collected by tracing tools like XRay[3]. NonSequitur works over traces that include function call entry and exit timestamps with the corresponding thread IDs. XRay produces such traces automatically by instrumenting the application at compile time. Most traces we study in evaluation were produced using XRay. But other tools or manual instrumentation could be used as well.

Given traces compressed with RegTime, NonSequitur outputs an .html file that contains the visualization. This file can be opened with any browser. Figure 4 shows a NonSequitur view for two threads from RocksDB. NonSequitur consists of two parts: the main panel and the task panel.
Main panel. The *main panel* shows what the threads are doing over time. As discussed in Section 3.2, the summary of an execution trace produced by RegTime can contain both compressed (encoded) and uncompressed segments. NonSequitur visualizes the activities within the uncompressed segments as callstacks over time, while it uses RegTime Visual Encodings to visualize the compressed segments. The main panel is divided into rows, one for each thread. The X axis represents time, while the Y axis represents callstack depth.

Figure 4 shows the activity of two threads, Thread 4 and Thread 11. The original traces of these threads captured approximately 2.5 million and two hundred thousand function calls respectively. However, RegTime summarized the behavior of these threads using a few hundred function calls. The summaries produced by RegTime are short enough that the horizontal space occupied by NonSequitur visualizations do not exceed 1,300 pixels. Thus, software developers using NonSequitur do not need to use horizontal scrolling within the main panel to see different parts of the visualization.
Task panel. The *task panel* sits at the top of NonSequitur. Using the task panel, users can select one or more thread timelines to show, while hiding the others (Figure 4(1)). We also let users

[2]https://github.com/auggywonger/nonsequitur_vis.git
[3]https://llvm.org/docs/XRay.html

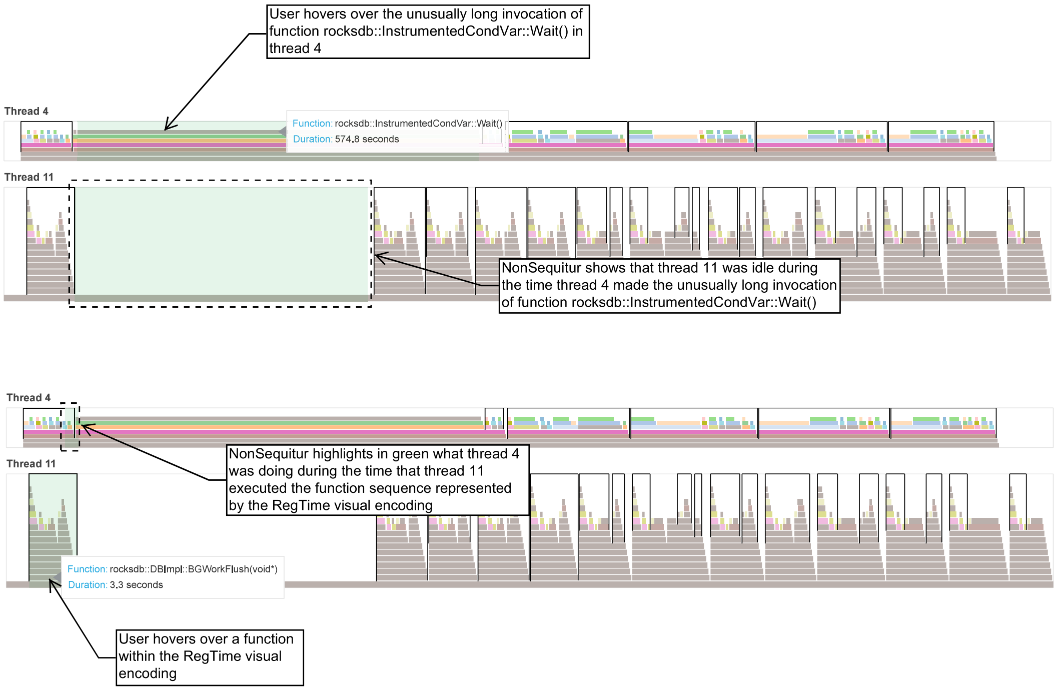

User hovers over the unusually long invocation of function rocksdb::InstrumentedCondVar::Wait() in thread 4
Thread 4
Function: rocksdb::InstrumentedCondVar::Wait()
Duration: 574.8 seconds
Thread 11
NonSequitur shows that thread 11 was idle during the time thread 4 made the unusually long invocation of function rocksdb::InstrumentedCondVar::Wait()
Thread 4
NonSequitur highlights in green what thread 4 was doing during the time that thread 11 executed the function sequence represented by the RegTime visual encoding
Thread 11
Function: rocksdb::DBImpl::BGWorkFlush(void*)
Duration: 3.3 seconds
User hovers over a function within the RegTime visual encoding

Fig. 5. (1) If users hover over a function call in the visualization which is not within a RegTime Visual Encoding, NonSequitur highlights in green the activities of all the threads during the time this function call was invoked. Additionally, a tooltip displays the name of the function and the duration of the function call. (2) If users hover over a function call belonging within a RegTime Visual Encoding, NonSequitur highlights in green the activities of all the threads during the time that the function sequence represented by the RegTime Visual Encoding was invoked. Additionally, a tooltip displays the name of the function and the cumulative duration of the function call within the function sequence represented by the RegTime Visual Encoding.

search the execution for function names (Figure 4(2)). After users enter the name of a function, NonSequitur displays only those threads which have called that function. NonSequitur relies on colour to distinguish which function calls are made by the threads. Below the search bar is a legend showing users the mapping between colours and functions (Figure 4(3)).

Figure 4(3) shows that many of the functions in the legend have been assigned the colour grey. The reason is because only a limited number of colours should be used to represent categories when visualizing data. NonSequitur assigns distinctive colours to only a subset of functions, while the other functions are shown in grey. To decide which functions should be represented with distinctive colours, NonSequitur computes a *prominence score* for each function: the number of times the function appeared in the execution trace summary produced by RegTime multiplied by the number of threads that invoked the function. Users can click on a function in the legend; occurrences of those functions are then highlighted in the main panel.

To improve readability in NonSequitur visualizations, each rectangular glyph is designed to have a minimum width in pixels. This design choice ensures that even the smallest glyphs are visible to users. However, a consequence of this approach is that the X axis in NonSequitur visualizations is nonlinear. For example, a linear X axis may cause some function calls of extremely short duration

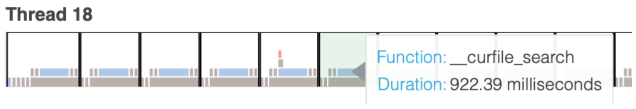


Fig. 6. Case study 1: a portion of the timeline for Thread 18 during a good run.

to occupy less than one pixel of horizontal space, while enforcing a minimum width would allow these function calls to occupy a visible amount of horizontal space.

Linked highlighting. A nonlinear X axis can make it challenging for users to discern correlations between activities across different threads. With a nonlinear X axis, function calls in multiple threads may appear to have occurred simultaneously when, in fact, they happened at different times. To mitigate this issue, NonSequitur uses *linked highlighting* to help users find correlations in activities across threads. If a user hovers over a function call in the visualization which is not within a RegTime Visual Encoding, NonSequitur highlights in green the activities of all the threads during the time this function call was invoked.

For example, Figure 5(1) shows the green highlighting which appears in the main panel when we hover over an unusually long invocation of the function rocksdb::InstrumentedCondVar::Wait(). The green highlighting indicates that during the time interval when thread 4 made an unusually long invocation of the function rocksdb::InstrumentedCondVar::Wait(), thread 11 experienced a period of inactivity. Similarly, if a user hovers over a function call belonging to the RegTime Visual Encoding in Figure 5(2), NonSequitur highlights in green the activities of all the threads during the time that the function sequence represented by the RegTime Visual Encoding was invoked.

Note that hovering over a function call in the main panel also reveals a tooltip, which provides more details about that function call. If users hover over a function call outside of a RegTime Visual Encoding, the tooltip shows the name of the function and the duration of the function call. If a user hovers over a function call within a RegTime Visual Encoding, the tooltip shows the name of the function and the cumulative duration of the function call within the function sequence represented by the RegTime Visual Encoding.

Next, we discuss how we used NonSequitur to investigate real-world performance issues.

4 NonSequitur Case Studies

In this section we present three case studies with using NonSequitur on actual bugs in complex software. The first two are with the MongoDB storage engine WiredTiger (WT). The third is with Meta's key-value store RocksDB.

4.1 Case Study 1: Slow LSM Performance in WiredTiger

In this case study we reproduce a MongoDB performance bug described in their JIRA ticket [6]. To trigger this bug, three workloads execute in sequence on a log-structured merge (LSM) tree. After populating the initial database, the application performs a read-only workload and achieves excellent performance. We call this the *good run.* Then, an application runs an update workload, which increases the size of the database, and then it runs another read workload. That second read workload achieves performance 99% slower than the good run and we refer to it as the *bad run.*

WT developers explored two hypotheses as to why the bad run is slower than the good run:

(1) Cache eviction is struggling to keep up. There are too many cache misses and cache eviction is not able to evict pages quickly enough to provide clean space for missed pages. *"The cache fill ratio is usually above 94.9% for the bad run, so application threads are evicting. And we often struggle to find candidate pages to evict." – a quote from discussion on the JIRA ticket.*

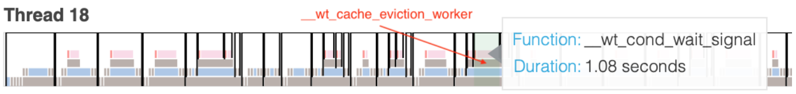


Fig. 7. Case study 1: a portion of the timeline for Thread 18 during a bad run.

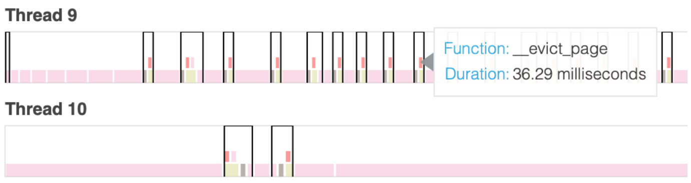


Fig. 8. Case study 1: timelines for two background threads, Threads 9 and 10, during a good run.

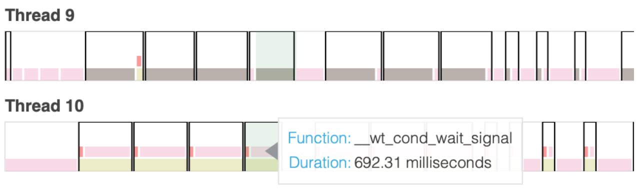

Thread 9
Thread 10
Function: __wt_cond_wait_signal
Duration: 692.31 milliseconds

Fig. 9. Case study 1: timelines for two background threads, Threads 9 and 10, during a bad run. In the green region Thread 9 is executing an eviction-related function, while Thread 10 is blocked on a condition variable.

(2) Internal threads performing the LSM tree consolidation interfere with application threads during the bad run. *"It's also worth noting that it's likely that there is quite a lot of LSM tree consolidation going on in the background during the read phase, which will alter the achievable throughput." – a quote from discussion on the JIRA ticket.*

To decide which of these hypotheses presents the true underlying causes of poor performance developers used (1) manual analysis of the internal statistics collected by their storage engine, (2) in-house visualization tool for these statistics and (3) lots of additional experiments with varying configuration parameters. In the end, the developers discard the second hypothesis and conclude that struggling eviction is the root cause: *"The cache size (*3 GB*) is too small for the size of the tree (*114 GB*). We are spending lots of time on eviction..." – a quote from discussion on the JIRA ticket.*

Although developers did find the underlying root cause and closed the ticket, it took multiple days and three different techniques or information sources. We now show how NonSequitur visualizations quickly guided us to the correct root cause with a single visualization of each run.

Figures 6 and 7 show a portion of the NonSequitur execution timeline for Thread 18, the application thread during the good run and the bad run, respectively. The good run is dominated by __curfile_search, a function that searches the BTree. The bad run is dominated by __wt_cond_wait_signal – a synchronization function that causes the thread to wait. The synchronization function is invoked by the __wt_cache_eviction_worker function. This suggests that in the bad run, Thread 18 is blocking because of eviction. We next look at the NonSequitur activity for the background threads that differ between the good run (Figure 8) and the bad run (Figure 9).

During the good run, Thread 9 makes calls to __evict_page which take a relatively small amount of time, and Thread 10 performs occasional eviction. During the bad run, however, Thread 10

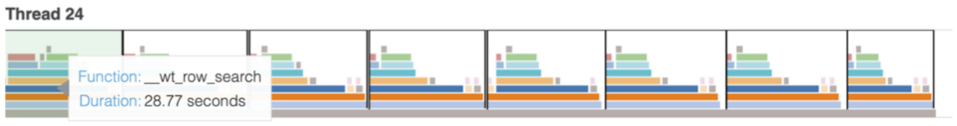


Fig. 10. Case study 2: an example thread that has a function called *worker* at the bottom of the call stack (bottom-most line in grey). The call to __wt_row_search indicates that this thread is likely searching the tree.

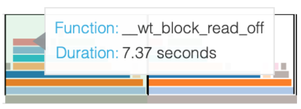
spends most of its time in __wt_cond_wait_signal, suggesting that it is blocked on a condition variable. Meanwhile, Thread 9 is busy with __evict_lru_walk, suggesting that it is working on LRU algorithm eviction-related activity. In summary the good run's application thread is mostly only searching the tree, while the one in the bad run performs blocking eviction. That coincides with inefficient eviction in the bad run's internal threads, which are delayed by waiting and performing the LRU walk activity.

The NonSequitur visualizations show that slow eviction is the main distinguishing feature of the bad run. Moreover, NonSequitur does not lead us down the wrong path of suspecting the internal tree consolidation activity (the alternative hypothesis). Although it seems like other tools showing call stacks over time could lead to the same conclusion as NonSequitur, it is not clear if they can be applied on the large traces used in this case study (~9 GB). We elaborate on this aspect in the next section.

4.2 Case Study 2: Slow Performance With the mmap Option in WiredTiger

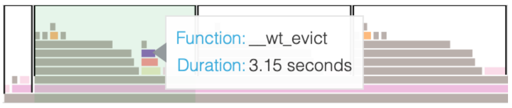
For this second case study there is no associated performance ticket. This issue was presented to us directly by a MongoDB engineer and at that time the engineer did not know the root cause for this performance bug. An update-heavy WiredTiger workload is executed with two different configuration options: with mmap turned on (mmap_update) and with mmap turned off (regular_update). The mmap feature performs I/O by mapping the file into memory and accessing its data via a memcopy, which transfers the requested data into the application buffer. Without mmap, I/O is done via read/write system calls, which also use memcopy internally. The difference is that with mmap_update, I/O operations can use the user-level memcopy, which is faster than the kernel memcopy because the user-level version can use SIMD registers [4]. The developer expected mmap_update to be faster than regular_update. But, in this scenario, they observed the opposite. To help the developer diagnose the problem, one of the authors generated NonSequitur visualizations for the mmap_update and regular_update executions and as the first step identified all application threads by observing the executed functions. We observe six threads having a function called "worker" at the bottom of their call stack and generally call functions such as __wt_row_search, suggesting that they are searching the tree (Figure 10). Examining the time taken by __wt_row_search we observe it is usually longer in the mmap_update than in the regular_update configuration, notably in the first part of the execution highlighted in green in Figure 10.

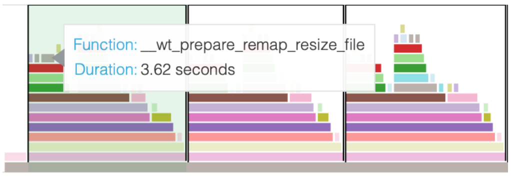
Hovering over the thread's callstack, we notice that a long duration of __wt_row_search is accompanied by a long duration of __wt_block_read_off as compared to later periods in the same threads. This function takes 7.37 seconds in the first period (nearly 25% of the total) and just over a second in subsequent periods (see figure to the right).



We proceed to examine activities of other threads to see if there is correlation between their activities and an unusually long execution of __wt_block_read_off. We observe two correlations:

Correlation 1: During the same period, Thread 23, a background thread, spends over 3 seconds in the function __wt_evict, while in the remainder of the period that function is so short it does not even register in the visualization (see figure to the right).





Correlation 2: During the same period Thread 25, another background thread, spent 3.6 seconds in __wt_prepare_remap_resize_file, whereas in the remaining periods that function was not present (see figure to the left).

When we presented this information to the WiredTiger developer, they quickly realized that __wt_prepare_remap_resize_file was indeed the root cause for the performance issue. With mmap on, a growing mapped file causes the storage engine to unmap it, resize the mapped area and remap the file. This involves synchronization with the other threads performing I/O, which could be regular threads, such as Thread 24 or background threads, such as Thread 23 (eviction thread). The need to resize the file in the beginning of the execution slowed down these threads and caused the overall degradation in performance.

The traces sizes used in this case study were ~3 GB, and as we elaborate in the next section other tools showing call stacks over time are not equipped to effectively present such large traces.

4.3 Case Study 3: RocksDB memtable Concurency

RocksDB is a key-value store used in Meta based on a log-structured merge (LSM) tree. It places new inserts and updates into an in-memory table, called memtable, and then rewrites them into an on-disk table (SSTable). The memtable in RocksDB is lock-free for readers and protected by a mutex for writers. A writer that gets the mutex will apply changes for all threads concurrently waiting; other writers link their updates into a data structure to be consumed by the winning writer. Several years ago we were in contact with a RocksDB developer, who introduced improvements to concurrency for the memtable. In the improved version, there is still a mutex but some of the work done by writer threads can be done concurrently.

The developer told us that while this feature generally improved performance, in certain cases it made performance worse. We reproduced the issue and visualized the executions in two configurations: the one with concurrent memtable off (concurrency_off) and the one with concurrent memtable on (concurrency_on).

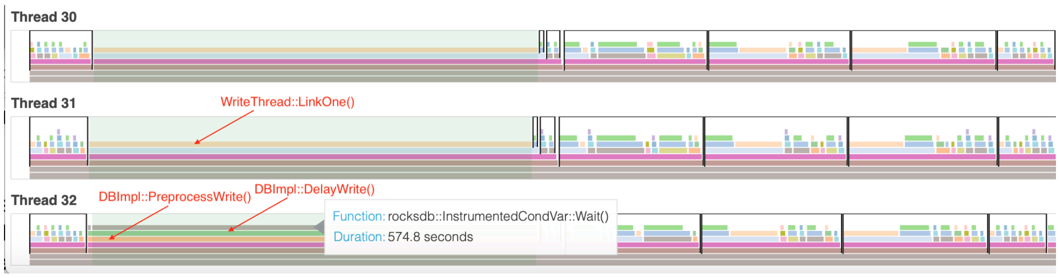


Fig. 11. Case study 3: Thread timelines with concurrency_on. Notice the long delay in the first half.

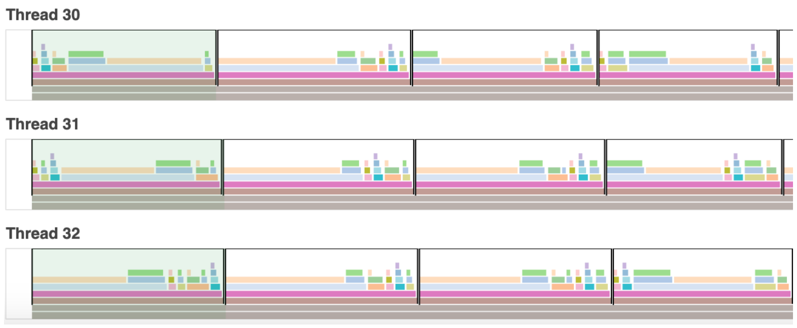


Fig. 12. Case study 3: Thread timelines with concurrency_off.

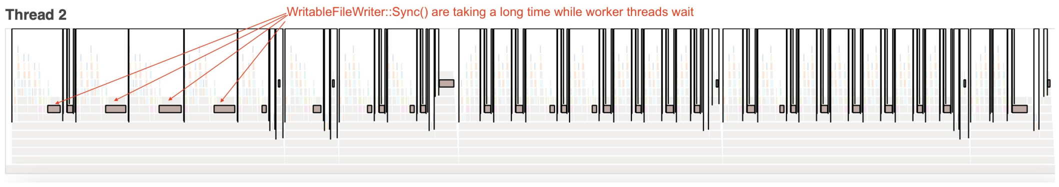
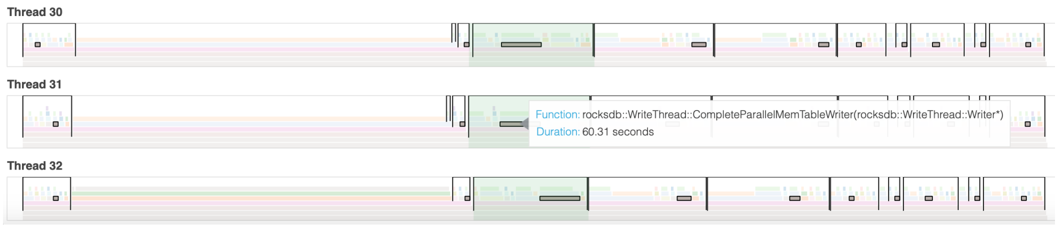

Thread 2
WritableFileWriter::Sync() are taking a long time while worker threads wait

Fig. 13. Case study 3: during a concurrency_on run, while there is delay in Figure 11, a background Thread 2 is busy synchronizing the memtable.



Function: rocksdb::WriteThread::CompleteParallelMemTableWriter(rocksdb::WriteThread::Writer*)
Duration: 60.31 seconds

Fig. 14. Case study 3: the parallel memtable writing function does not appear during the period of delay in Figure 11, eliminating it as the root-cause.

Examining the NonSequitur visualizations, we see clear differences. In the concurrency_on configuration the threads executing the workload (Threads 21-32; we show only Threads 30-32 for brevity, but the omitted threads look the same as Threads 30-32) experience a huge delay in the first half of the execution (Figure 11). Thread 32 is waiting on a condition variable as it is trying to apply writes, while threads 21-31 are trying to link their writes. This does not occur in the concurrency_off configuration (Figure 12).

We observe that this delay in the concurrency_on configuration coincides with a period where one background thread takes a longer time to sync the memtable than during the subsequent periods (Figure 13).

In contrast, in the concurrency_off configuration: Sync() is relatively short in each period throughout the execution, and looks the same as in the second half of the execution in Figure 13.

We also confirmed CompleteParallelMemTableWriter, the function implementing the concurrency feature, does not occur during the period of delay (Figure 14), and does *not* appear to be the culprit.

Table 2. Summary of the Execution Traces Used in Our User Study

Trace Name	Description	Trace Size	Total Exec. Time
WT normal	A trace of WiredTiger executing queries over a log-structured merge-tree (LSM tree).	24 threads making 0.8M calls to 20 different functions	0.5 minutes
WT large	A large trace of WiredTiger running an update heavy benchmark test.	28 threads making 50M calls to 320 different functions	5 minutes (includes the benchmark test setup)
RDB large	A trace of RocksDB running the dbbench benchmark test with parallel writes enabled.	32 threads making 39M calls to 747 different functions	20 minutes (includes the benchmark test setup)

We conclude that the root cause of performance degradation is the delay in applying writes in the beginning of the execution, which coincides with a delayed write sync in a background thread.

The size of the traces used in this visualization was ~10 GB; as we show next, other tools showing call stacks over time are either challenging to use with such large traces or crash when presented with this volume of information.

5 Methodology for NonSequitur Controlled Evaluation

We empirically evaluate how effective NonSequitur is at helping software developers diagnose performance issues by answering the following research questions:

RQ1: Does NonSequitur help analyze large execution traces to understand what threads are doing over time?

RQ2: Does NonSequitur help analyze large execution traces to find when unusually long function latencies occur?

RQ3: Does NonSequitur help analyze large execution traces to learn what one thread was doing during the time when another thread was blocked or delayed?

RQ4: For small traces that can be visualized with other tools, does NonSequitur lead to false conclusions in questions regarding thread activity over time as compared to other tools that do not drop information from the trace?

To address these questions, we conducted a within-subjects user study with 42 participants. The rest of this section presents our methodology and results.

5.1 Multithreaded Traces

Each participant was asked to carry out performance analysis tasks on three different execution traces shown in Table 2. The first two traces, which we call the *normal WiredTiger (WT) trace* and the *large WiredTiger trace* were taken from WiredTiger, MongoDB's storage engine. The third trace, which we call the *large RocksDB (RDB) trace* was taken from RocksDB. We derived these traces from two software systems where we knew developers were struggling with diagnosis of performance bugs: from the original study for WiredTiger [19] or from our own interaction with developers for RocksDB.

Although the normal WiredTiger and large WiredTiger traces came from the same system, they capture WiredTiger performing two different tasks. Additionally, the number of function invocations captured in the large WiredTiger trace is 60 times larger than the number of function

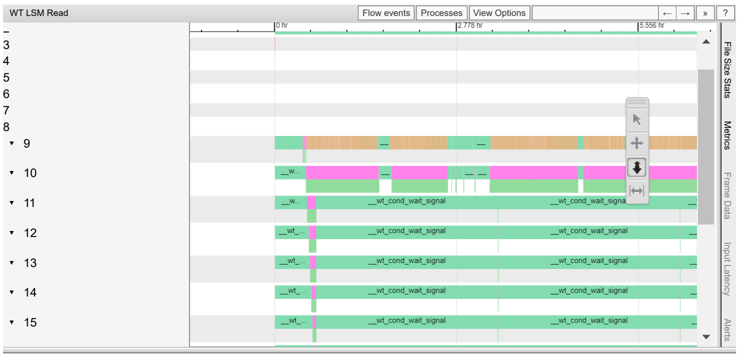


Fig. 15. Chrome TraceViewer, part of the Chrome Developer Tools Suite. It visualizes multi-threaded execution traces as rows of icicle plots.

invocations in the normal trace. Therefore, participants would not be able to apply their knowledge of the normal WiredTiger trace to analyze the large one.

We recruited 42 participants for our user study. The participants included computer engineering students and professional software developers:

- We classified 7 of the participants as "WiredTiger Developers." These participants are either former or current WiredTiger developers.
- We classified 20 of the participants as "Experienced Performance Debuggers." These participants indicated they had three or more years of experience debugging the performance of software systems.
- We classified 10 of the participants as "Performance Debuggers." These participants stated they had 1-2 years of experience debugging the performance of software systems.
- We classified 5 of the participants as "Inexperienced Performance Debuggers." These participants stated they had no experience debugging the performance of software systems.

Because none of the participants indicated that they had experience developing RocksDB, we did not categorize any participants as RocksDB developers.

5.2 Variable Selection

Our study involved one independent variable: the type of trace visualization tool participants used for performance analysis. Because our study followed a within-subjects design, participants were required to use both “NonSequitur” and other tools, which we refer to as *other*. We describe the other tools in the next section. The dependent variables were (1) how accurately the participants were able to do the performance analysis tasks, and (2) how long participants took to perform the tasks.

5.3 Tools Used for Comparison: Chrome TraceViewer, and OpTrack

We used two tools in the *other* category: Chrome TraceViewer [1] and OpTrack [5]. These were the only publicly available tools known to us that, like NonSequitur, display the per-thread callstacks over time. Unlike NonSequitur, they do not drop any information from the trace, so they were ideal for answering RQ4. Because our study followed a within-subjects design, each participant was exposed to both the treatment condition and control condition; consequently, all participants were required to use NonSequitur, Chrome TraceViewer, and OpTrack.

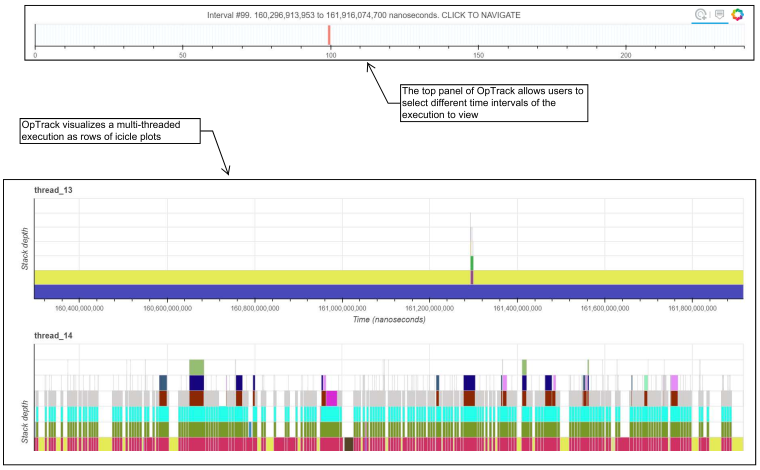

Interval #99. 160,296,913,953 to 161,916,074,700 nanoseconds. CLICK TO NAVIGATE
The top panel of OpTrack allows users to select different time intervals of the execution to view
OpTrack visualizes a multi-threaded execution as rows of icicle plots
thread_13
Stack depth
Time (nanoseconds)
thread_14

Fig. 16. OpTrack is part of WiredTiger. It visualizes per-thread call stacks, breaking up the timeline across 240 individual pages. A developer selects the page to view from the top panel.

Chrome TraceViewer, which is part of the Chrome Developer Tools suite, is representative of a typical trace visualization tool. It visualizes multi-threaded execution traces as rows of icicle plots, with each icicle plot depicting callstacks that occurred over time within one thread (see Figure 15). Chrome TraceViewer offers users a number of navigation features, including zooming into the visualization, highlighting functions of interest with prominent colours, and scrolling horizontally across the X axis.

Since Chrome TraceViewer visualizes *all* events in a trace and uses the width of a single screen to do this, it is unable to visualize execution traces containing millions of events: the UI crashes when presented with very large traces. Therefore, for the large traces we used OpTrack, which is part of WiredTiger distribution [5]. Similarly to Chrome TraceViewer, OpTrack visualizes multi-threaded execution call stacks over time (see Figure 16). To handle large traces, OpTrack divides traces into 240 equal time intervals and shows one interval at a time. Users select from a navigation panel the interval to view. Consequently, OpTrack cannot show the entire execution all at once. Other than the navigation panel, OpTrack does not have any notable navigation features such as zooming or highlighting functions of interest.

5.4 Experiment Treatments

Our study followed a within-subjects design. Each participant had to analyze each trace twice: once with NonSequitur (the *treatment* condition), and once with another tool (the *control* condition).

We randomly placed participants into one of two groups. In the first group, participants were always exposed to the treatment condition first for all traces. In the second group, participants were always exposed to the control condition first for all traces. That is, a participant would either always analyze a trace with NonSequitur first or always analyze a trace with NonSequitur last. In the next section, we describe the tasks we asked participants to perform when analyzing a trace with NonSequitur and the other tools.

Table 3. Performance analysis tasks used in the study

Task	Task Description	Example Question
T1	Understand what a thread is doing over time.	Which thread executes __evict_lru_walk throughout its lifetime but only executes __evict_page at the beginning and at the end of its lifetime?
T2	Find when unusually long latencies or long periods between function calls occur.	Which threads contain invocations of rocksdb::WriteThread::LinkOne exceeding 500 seconds? When do these invocations occur in the execution?
T3	Understand what one thread was doing during the time when another thread was blocked or delayed.	During the time intervals when thread 24 is not executing any function, which thread spends most of its lifetime in __wt_cond_wait_signal?

Table 4. The number of tasks that participants needed to complete for each trace when exposed to either the treatment or the control condition.

Trace	Condition	# of T1 tasks	# of T2 tasks	# of T3 tasks
WT Normal	Treatment	1	1	2
WT Normal	Control	1	1	2
WT Large	Treatment	1	1	1
WT Large	Control	1	1	1
RDB Large	Treatment	1	1	1
RDB Large	Control	1	1	1

5.5 Experiment Tasks

We asked participants to perform the key performance analysis tasks identified in a case study on how developers debug performance [19]. To quantify how well a participant performed on a performance analysis task, each participant was required to answer a question that assessed how accurately they performed that task. Table 3 lists the types of tasks and example questions. Table 4 shows the number of performance analysis tasks that each participant performed on a given trace when exposed to either the treatment or the control condition.

Asking exactly the same question for both the treatment and the control condition would create undesirable *learning effects*. That is, a participant could learn the answer for a question using the treatment tool and then would trivially know the answer for the control tool without having to actually apply it. To that end, we present users with very similar, but non-identical questions for each task.

For a given trace, we asked participants to perform T1 tasks first, followed by T2 tasks, and finally T3 tasks. T1 tasks were the simplest, while T3 tasks were the most complex. By presenting the tasks in this order, we gradually increased the complexity of the performance analysis tasks, thereby reducing user fatigue.

5.6 Experimental Procedure

The procedure for the user study consisted of three phases:

- **Prestudy**: We asked potential participants to complete a pre-questionnaire to gather demographic information about them. Students who indicated that they completed at least five computer engineering courses or possessed at least one school term of software development

work experience were allowed to participate in the study, as were non-student respondents indicating that they were working as software developers.
- **Training**: All participants watched tutorial videos explaining how to use NonSequitur, Chrome TraceViewer, and OpTrack. After that, the participants completed quizzes testing their knowledge of the tools. After completing the quizzes, the participants were given the correct answers, so that they can become comfortable with the tools. Participants were also given opportunities to re-watch the tutorial videos. On average, it took them approximately one hour to complete the training.
- **Tasks**: During this phase, the participants were exposed to the experiment treatments. The order in which the traces were presented to the participants was consistent: starting with the normal WiredTiger trace, followed by the large WiredTiger trace, and ending with the large RocksDB trace. The normal WiredTiger trace was the simplest trace to analyze and was therefore presented to participants first. The large WiredTiger trace and the large RocksDB trace were more complicated, so we presented these trace to the participants after the normal WiredTiger trace. Introducing the traces in this order allowed the participants to gradually gain familiarity with the tools and reduce user fatigue. However, the order in which the participants received questions about the trace was not fixed because, as stated earlier, we randomly exposed participants to either the treatment condition first or the control condition first for all traces.
  Participants took between one and three hours to finish the tasks with both NonSequitur and the other tools. Because of the large number of participants involved in our study and the length of the study, we did not observe the participants in person. Instead, we used Zoom to record the participants' screens, so we could analyze later how they used the tools. Participants were permitted to provide us with feedback on their experience with the user study if they wished.
  We gave participants access to the training material throughout this phase and allowed them to take as much time as they liked to answer questions, only requiring participants to finish within 6 hours, a generous time limit. Participants were also allowed to take breaks to reduce the risk of them answering questions incorrectly due to user fatigue. We had initially considered imposing strict time limits on how long participants had to answer questions to prevent breaks but decided against it because pilot participants preferred having time to familiarize themselves with the tools. We used the Qualtrics survey platform to automatically present tools to participants, display the questions to participants, record the participants' responses, and time how long it took them to answer the questions. The measurement obtained from Qualtrics on how long it took a participant to answer a question included any breaks the participant took.

5.7 Threats to Validity

The external threats of conducting our user study are the representativeness of the performance analysis tasks, participants, and traces selected for the experiment. To mitigate the threat of task representativeness, we designed our three tasks based on a case study on how developers debug performance [19]. We initially designed the study so that participants perform a wider variety of tasks, but the pilot participants felt the study was unreasonably long. We mitigated the threat of participant representativeness by recruiting only professional computer engineers, or graduate students with at least five computer engineering courses or at least one school term of software development work experience. We addressed the threat of the representativeness of the traces by using traces taken from WiredTiger and RocksDB, two software systems where we knew developers tended to spent a lot of time fixing performance issues. We considered having participants analyze traces from other systems as well, but this would have made the study duration unreasonably long.

Because the order questions presented to the participants was not fully randomized, one internal threat is that findings from our user study might be due to other factors that are not directly related to the specific tasks. The order of execution traces presented to participants was consistent, but there was some randomization in the order of questions. Each participant analyzed each trace twice: once with Non-Sequitur (treatment) and once with another tool (control). Participants were randomly assigned to start with either the treatment or control condition for all traces. The questions differed slightly between conditions, resulting in different sequences based on the starting condition. Nevertheless, the order in which participants receive the questions was not fully randomized because, within a trace, each participant received the questions in the same order. We deliberately avoided randomizing the order for two reasons: to ensure observed effects were due to the questions themselves, not their order, and to reduce cognitive load by presenting easier questions first, followed by more difficult ones. This approach helped participants gradually familiarize themselves with the tools and minimized user fatigue.

A second internal threat is the participants experiencing user fatigue from spending up to three hours analyzing three execution traces using three different tools. We mitigated this threat by allowing our participants to complete the performance analysis tasks at their own pace, provided that they could complete the study within 6 hours. Additionally, we gradually increased the complexity of the performance analysis tasks that the participants performed.

A third internal threat of our user study is bias towards exposing participants to the treatment condition first or the control condition first for each execution trace. To mitigate this threat, we relied on the Qualtrics survey platform to randomly assign participants to either always be exposed to the treatment condition first or to the control condition first. We also relied on the Qualtrics survey platform to measure how long it took participants to complete the performance analysis tasks, thus mitigating the threat of bias in measuring time.

Another internal threat is potential bias towards evaluating how well participants performed on the performance analysis tasks. For this, we created a rubric *prior* to conducting our user study, and we strictly used this rubric for grading the participant responses.

The tools we selected for the control condition also pose an internal threat. We provided the participants with Chrome TraceViewer for the small WiredTiger trace because it is representative of a typical trace visualization tool. We provided the participants with OpTrack to analyze the larger traces because it was designed to visualize large execution traces. We mitigate the threat that participants perform worse over time due to fatigue by letting participants complete the performance analysis tasks at their own pace.

A final internal threat is that participants used NonSequitur three times within the same session, but they used Chrome TraceViewer once and OpTrack twice. There is the threat of underlying carryover effect from participants growing accustomed to using NonSequitur over the duration of the experiment and performing better with this tool during later trace analyzes. We believe the threat of the underlying carryover effect is not a significant concern because we provided all participants with training on how to use Chrome TraceViewer and OpTrack. Furthermore, participants had access to these training materials when performing the performance analysis tasks.

6 Results for Controlled NonSequitur Evaluation

To determine how well the participants did on the tasks, we graded their responses to the questions using a rubric that we created *before* conducting our user study. Questions with multiple correct answers were worth between two and three points. For example, a two-point question might

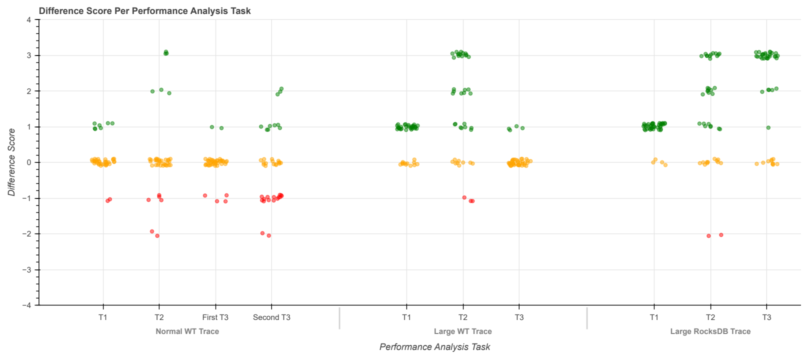


Fig. 17. The difference scores from the NonSequitur evaluation. Green dots represent difference scores above zero, yellow dots represent difference scores of zero, and red dots represent difference scores below zero.

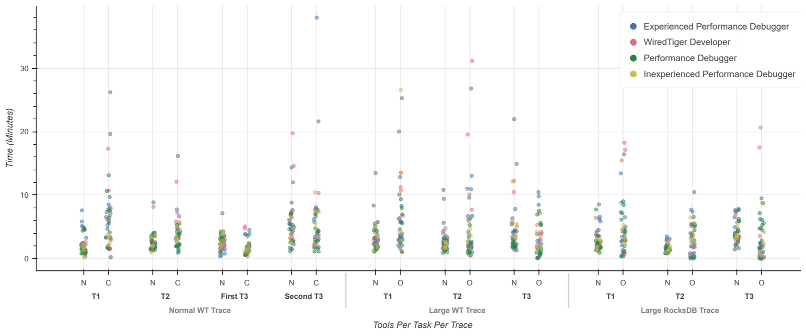


Fig. 18. The number of minutes the participants spent on each performance analysis task. 'N' on the x-axis stands for NonSequitur, while 'C' and 'O' stand for Chrome TraceViewer and OpTrack respectively.

ask participants to identify what threads experienced a latency spike and when the latency spike occurred (task T2). To earn the full two points, participants had to correctly identify both the threads and the timing of the latency spikes. In contrast, questions with only one correct answer were worth one point. An example of a one-point question is asking participants to identify which thread executed a specific kind of activity over time (task T1).

Participants who answered a question correctly received full marks. Those who answered incorrectly received zero points. For questions worth multiple points, partial marks were awarded if some but not all parts of the answer were correct. For example, a participant who correctly identified the thread with a latency spike but not the timing would receive one point instead of two. Participants never received negative points.

To compare how well a participant did on a performance analysis task with NonSequitur compared to the alternatives, we used *difference* scores. Each difference score was calculated by taking the score a participant obtained after completing a performance analysis task on an execution trace using NonSequitur and subtracting the score the same participant obtained using a different tool. Thus, a difference score greater than zero indicates the participant performed a performance analysis task on an execution trace more accurately with NonSequitur than with the other tool. Conversely, a difference score less than zero means the participant performed better using the other tool. A difference score of zero means the participant achieved the same score with both tools.

Table 5. P-values of the McNemar's test and the Wilcoxon signed-rank test. P-values lower than 0.05 suggest that the difference in scores achieved with NonSequitur and the scores with another tool is statistically significant, leading us to reject the null hypothesis that there is no difference. Conversely, p-values above 0.05 indicate insufficient evidence to reject the null hypothesis, suggesting that any observed difference in scores might not be statistically significant.

Trace	**Task**	**Max score**	**Stat test**	**P-Value**
WT normal	T1	1	McNemar	0.182
WT normal	T2	3	Wilcoxon	0.883
WT normal	First T3	3	Wilcoxon	1
WT normal	Second T3	3	Wilcoxon	0.236
WT large	T1	1	McNemar	1.192×10^{-7}
WT large	T2	3	Wilcoxon	9.537×10^{-7}
WT large	T3	1	McNemar	0.248
RDB large	T1	1	McNemar	1.192×10^{-7}
RDB large	T2	3	Wilcoxon	6.110×10^{-6}
RDB large	T3	3	Wilcoxon	6.162×10^{-8}

Figure 17 displays a scatter plot of *difference* scores. Note that we asked participants to perform two T3 tasks for the normal WiredTiger trace (referred to as First T3 and Second T3 in Figure 17). Figure 18 depicts a scatter plot showing the number of minutes participants spent on each task.

We used the McNemar's test and the Wilcoxon signed-rank test to validate the statistical significance of these findings. McNemar's test is a non-parametric statistical test used to assess paired binary outcome data; consequently, we used this test to assess the statistical significance of the difference scores from questions worth one point. We used the Wilcoxon signed-rank test to assess the statistical significance of the difference scores from questions worth multiple points. The Wilcoxon signed-rank test is a non-parametric statistical test for comparing paired data to find out if their population means ranks differ. Table 5 shows the p-values from each statistical test.

Figure 17, shows the following:

- When conducting performance analysis tasks on the *normal* WiredTiger trace, which was a rather small trace that could be easily visualized with other tools, participants performed equally well with both NonSequitur and Chrome Traceviewer.
- When conducting performance analysis tasks on the *large* WiredTiger trace, participants performed better with NonSequitur than with OpTrack. However, most of the difference scores for T3 on this trace were zero, meaning that participants generally achieved the same score with both NonSequitur and OpTrack in this instance.
- When conducting performance analysis tasks on the large RocksDB trace, participants performed better with NonSequitur than with OpTrack.

These data suggest the following: (1) Since users did equally well on the small trace, which can be effectively visualized with Chrome TraceViewer, we can conclude that RegTime's dropping of information from the trace does *not* impede users from finding correct answers (RQ4). After all, Chrome TraceViewer doesn't drop any information from the trace, and our users did no worse with NonSequitur. (2) For large traces, NonSequitur is generally more effective at providing the needed information than OpTrack, the tool that splits the trace across many smaller visualizations (RQ1-RQ3). At the same time, some tasks, such as T3 for WT large, may be completed equally well with a simpler tool.

To further substantiate these conclusions, we dig deeper into how the participants used the tools.

T1 tasks To perform task T1, participants needed to identify what threads were executing functions of interest and then observe when these threads execute these functions. From analyzing the Zoom video recordings, we saw that participants liked to use the search bar in NonSequitur to determine what threads were executing these functions. The participants also clicked on the functions in the legend to highlight them. Because RegTime summarized the behavior of threads with multiple RegTime expressions, participants could identify, at a high-level, when threads executed functions. For example, participants could see if a thread executed a function at the beginning, middle, or end of its lifetime.

Although participants were able to perform the T1 task for the normal WT trace using Chrome TraceViewer, we saw from our review of the Zoom video recordings that participants needed to make repeated use of Chrome TraceViewer's navigation features to complete T1. For example, participants would attempt to highlight functions of interest with Chrome TraceViewer, only to find that calls to those functions were too small to be easily spotted despite being displayed in prominent colours. Participants then needed to zoom into the visualization to determine what threads executed these functions. If participants zoomed into a Chrome TraceViewer visualization, they would need to scroll horizontally along the timeline of the visualization so they can see when the threads executed the functions over time. The participants' need to constantly use the navigation features with Chrome TraceViewer explains why we see in Figure 18 that participants tended to spend more time using Chrome TraceViewer than NonSequitur.

On average, participants scored three times higher on the T1 task with the large WT trace with NonSequitur than with OpTrack. Participants also on average scored 11 times higher on the T1 task with the large RocksDB trace with NonSequitur than with OpTrack. Participants struggled to use OpTrack to perform T1 tasks for large traces because OpTrack does not provide users with any navigation features beyond selecting what time intervals to display. Thus, participants found it difficult to use OpTrack to identify what threads called functions of interest or what threads were doing over time. Participants' strategies when using OpTrack were limited to a) randomly or methodically selecting time intervals, and b) spending time studying the visualization of the time interval to determine whether a thread invoked a function of interest. Consequently, Figure 18 shows that participants spent more time using OpTrack than NonSequitur. The participants spent on average twice as much time using OpTrack to perform the T1 task on the large WT trace. The average amount of time the participants spent using OpTrack to perform the T1 task on the large RocksDB trace was 1.7 times longer than the average time spent using NonSequitur.

T2 tasks We observed from the Zoom video recordings that, when using NonSequitur, participants used the search bar and legend together to determine what threads executed outlier events. Because RegTime leaves outlier events uncompressed, participants were also able to identify when threads executed these outlier events. On average, participants scored 2.6 times higher performing the T2 task on the large WT trace with NonSequitur than with OpTrack. Participants also on average scored twice as high when performing the T2 task on the large RocksDB trace with NonSequitur than with OpTrack.

Figure 18 shows that participants generally spent more time performing T2 tasks with both Chrome TraceViewer and OpTrack than with NonSequitur. With Chrome TraceViewer, participants needed to spend time scrolling to find the threads that executed the outlier events. Meanwhile, participants using OpTrack needed to spend a lot of effort studying multiple time intervals to find outliers. The participants spent on average twice as much time using OpTrack on the T2 task with the large WT trace. The average amount of time the participants spent using OpTrack on T2 task with the large RocksDB trace was three times longer than the average time using NonSequitur.

T3 tasks When using NonSequitur, participants made extensive use of linked highlighting to explore what threads were doing when the outlier events occurred. When the boundaries of the RegTime expressions lined up with the start and end of outlier events, as depicted in Figure 5, participants scored four times higher with NonSequitur than with OpTrack.

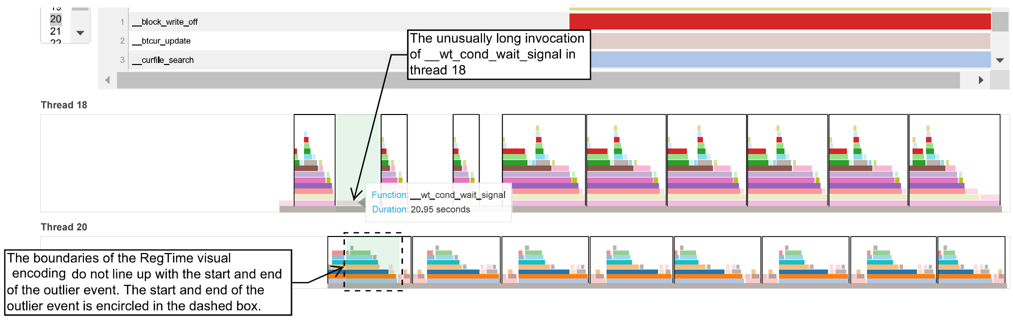

__block_write_off
__btcur_update
__curfile_search
The unusually long invocation of __wt_cond_wait_signal in thread 18
Thread 18
Function: __wt_cond_wait_signal
Duration: 20.95 seconds
Thread 20
The boundaries of the RegTime visual encoding do not line up with the start and end of the outlier event. The start and end of the outlier event is encircled in the dashed box.

Fig. 19. An example of when NonSequitur is not able to assist software developers with T3 tasks.

However, participants struggled to perform the T3 task if the RegTime Visual Encoding boundaries did not line up. For instance, Figure 19 shows what happened when participants attempted to use linked highlighting to perform the T3 task for the large WT trace. Participants were able to identify the outlier event, which was an unusually long invocation of the function __wt_cond_wait_signal in thread 18. However, they were unable to understand what thread 20 was doing when the unusually long invocation of the function __wt_cond_wait_signal occurred because the RegTime Visual Encoding boundaries in thread 20 did not line up with the start and end of this outlier event.

Interestingly, the average time participants spent performing the T3 task with OpTrack was less than with NonSequitur. From reviewing the Zoom video recordings, we believe many participants simply gave up using OpTrack to perform this task. One participant told us after completing the study that *"using OpTrack for this will be very time consuming and tedious."*

Impact of Performance Debugging Experience We see from Figure 18 that the participants who were willing to spend the most time performing the performance analysis tasks tended to be WT developers and experienced performance debuggers. We speculate that these participants were used to spending a lot of time debugging performance and were therefore more patient with completing the performance analysis tasks during our study.

Summary of Findings. We conclude that RegTime and NonSequitur are well suited for understanding what threads were doing over time (RQ1); detecting outlier events (RQ2); and understanding what threads were doing over time fwhen these events occurred (RQ3). For these performance analysis tasks, the fact that RegTime drops information from traces did not prevent users from completing the task successfully. Even for small traces, the users answered questions correctly and more quickly than with conventional tools (RQ4).

For larger traces, participants were able to accomplish the task successfully more often with NonSequitur than with other tools, and took less time doing so. At the same time, some of the visual encoding techniques used in NonSequitur prevented users from identifying what threads were doing during long outlier events and in certain trace conditions. Finding ways to improve NonSequitur for these scenarios is a promising direction for future work.

7 RegTime Algorithm Evaluation

Table 6 shows the compression ratios achieved by RegTime when compressing the traces of threads in WT, RocksDB, and the Chrome browser. The WT and RocksDB traces were collected using XRay, a function call tracing system developed by Google [8]. XRay is supported by LLVM, which automatically instruments the application given a compilation flag. We obtained Chrome traces using Windows process Monitor [7].

Table 6. Compression ratios achieved with RegTime. n_{before} is the number of events in the trace before compression. n_{after} is the number of events in the trace after compression. The compression ratios of the trace sizes in MB were approximately the same.

System	**Thread**	**# unique events**	n_{before}	n_{after}	**Compression Ratio**
RocksDB	0	74	2,466,295	323	7,635
RocksDB	1	316	2,762,774	2,485	1,111
WiredTiger	0	115	7,784,936	546	14,258
WiredTiger	1	81	6,935,791	209	33,185
Chrome	0	34	274,361	96	2,857
Chrome	1	8	46,464	56	829

For this experiment, we configured RegTime with the following threshold values. We empirically found that these values tend to produce most compact yet useful visualizations.

- $CallDurationThresh$: Function calls are considered to be short if its duration is less 1% of the thread's total execution time.
- $CallGapThresh$: The time between two consecutive function calls is considered short if that time is less than 0.1% of the thread's total execution time.
- $TotalTimeFractionThresh$: The time interval occupied by a compressible function sequence cannot exceed 13% of a thread's total execution time.

Table 6 shows that RegTime compression ratios have a high variance, which depends on the shape of the call stacks in the trace. However, in all cases RegTime produced a compressed trace that *can* be easily inspected using NonSequitur. This is critical because existing tools are frequently unusable at scale. Achieving the high compression ratios in Table 6 enables user-friendly visualization of large traces, but also requires dropping some information. This is a trade-off in our design, which means that RegTime and NonSequitur are suitable for certain performance-analysis tasks, but not for others. For example NonSequitur would not be useful for within-function optimizations or tasks where the precise order of executed functions is crucial information.

8 Conclusion

Debugging performance issues in multi-threaded applications requires tools that scale to accommodate large amounts of runtime information. We described a trace compression algorithm called RegTime to assist developers with investigating performance bugs in multi-threaded applications. RegTime achieves compression ratios exceeding 1,000, allowing developers to analyze execution traces with millions of events. We also described NonSequitur, a trace visualization tool for developers to analyze the RegTime-compressed execution traces. We presented three case studies which demonstrate how we used NonSequitur to analyze real-world performance issues with WiredTiger and RocksDB. We also carried out a structured user study with 42 participants. Our study participants scored 11 times higher when using NonSequitur for performance analysis on large traces as compared with other tools. Additionally, for some performance analysis tasks, the participants spent on average three times longer with other tools than with NonSequitur.

Data-Availability Statement

The software described in Sections 3 and 6 is available on Zenodo [46].

Acknowledgements

This work is generously supported by an NSERC Discovery Grant (RGPIN-2017-04170) and Huawei Technologies Co.

References

[1] [n. d.]. Chrome TraceViewer. https://github.com/catapult-project/catapult/blob/main/tracing/README.md. [Accessed 04-02-2024].

[2] [n. d.]. G Datacenter Computers: Modern Challenges in CPU Design (invited talk, UNC Distinguished Alumni Speaker Series 2015). https://www.youtube.com/watch?v=QBu2Ae8-8LM. [Accessed 07-31-2024].

[3] [n. d.]. G Datacenter Computers: Modern Challenges in CPU Design (invited talk, UNC Distinguished Alumni Speaker Series 2015). https://www.pdl.cmu.edu/SDI/2015/slides/DatacenterComputers.pdf. [Accessed 07-31-2024].

[4] [n. d.]. Getting Storage Engines Ready for Fast Storage Devices. https://www.mongodb.com/blog/post/getting-storage-engines-ready-fast-storage-devices. [Accessed 07-31-2024].

[5] [n. d.]. OpTrack in WiredTiger. http://source.wiredtiger.com/10.0.0/tool-optrack.html. [Accessed 04-02-2024].

[6] [n. d.]. Slow read performance after update workload with LSM tree. https://jira.mongodb.org/browse/WT-4637. [Accessed 07-31-2024].

[7] [n. d.]. Windows Process Monitor. https://learn.microsoft.com/en-us/sysinternals/downloads/procmon. [Accessed 04-02-2024].

[8] [n. d.]. XRay. https://research.google/pubs/xray-a-function-call-tracing-system/. [Accessed 04-02-2024].

[9] Nahla J Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I Maletic. 2019. Developer reading behavior while summarizing java methods: Size and context matters. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 384–395.

[10] Luay Alawneh, Abdelwahab Hamou-Lhadj, and Jameleddine Hassine. 2016. Segmenting large traces of inter-process communication with a focus on high performance computing systems. *Journal of Systems and Software* 120 (2016), 1–16.

[11] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2018. Inferring hierarchical motifs from execution traces. In *Proceedings of the 40th International Conference on Software Engineering*. 776–787.

[12] Mona Attariyan, MIchael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 307–320. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/attariyan

[13] Luiz André Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60(4) (2017), 48–54.

[14] Bas Cornelissen and Leon Moonen. 2008. On large execution traces and trace abstraction techniques. *Software Engineering Research Group, Delft* (2008).

[15] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie Van Deursen, and Jarke J Van Wijk. 2008. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software* 81, 12 (2008), 2252–2268.

[16] Rémy Dautriche, Renaud Blanch, Alexandre Termier, and Miguel Santana. 2016. Traceviz: A visualization framework for interactive analysis of execution traces. In *Actes de la 28ième conference francophone sur l'Interaction Homme-Machine*. 115–125.

[17] Wim De Pauw and Steve Heisig. 2010. Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*. 143–152.

[18] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[19] Alexandra Fedorova, Craig Mustard, Ivan Beschastnikh, Julia Rubin, Augustine Wong, Svetozar Miucin, and Louis Ye. 2018. Performance comprehension at WiredTiger. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 83–94.

[20] Yang Feng, Kaj Dreef, James A Jones, and Arie van Deursen. 2018. Hierarchical abstraction of execution traces for program comprehension. In *Proceedings of the 26th Conference on Program Comprehension*. 86–96.

[21] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[22] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 135–151. https://doi.org/10.1145/3445814.3446700

[23] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. 2002. Compression techniques to simplify the analysis of large execution traces. In *Proceedings 10th International Workshop on Program Comprehension*. IEEE, 159–168.

[24] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 145–155.

[25] Katherine E Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. 2014. State of the Art of Performance Visualization. *EuroVis (STARs)* (2014).

[26] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 469–478. https://doi.org/10.1145/3038912.3052649

[27] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 155–170.

[28] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[29] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3132747.3132749

[30] Benjamin Karran, Jonas Trümper, and Jürgen Döllner. 2013. Synctrace: Visual thread-interplay analysis. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–10.

[31] Richard Li, Min Du, Zheng Wang, Hyunseok Chang, Sarit Mukherjee, and Eric Eide. 2022. LongTale: Toward Automatic Performance Anomaly Explanation in Microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (Beijing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 5–16. https://doi.org/10.1145/3489525.3511675

[32] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024.

[33] David Lo, Siau-Cheng Khoo, and Chao Liu. 2007. Mining temporal rules from program execution traces. *Proc. of Int. Work. on Program Comprehension through Dynamic Analysis* (2007).

[34] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Trans. Comput. Syst.* 35, 4, Article 11 (dec 2018), 28 pages. https://doi.org/10.1145/3208104

[35] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 237–246.

[36] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[37] Michael John Pacione, Marc Roper, and Murray Wood. 2004. A novel software visualisation model to support software comprehension. In *11th working conference on reverse engineering*. IEEE, 70–79.

[38] Heidar Pirzadeh, Akanksha Agarwal, and Abdelwahab Hamou-Lhadj. 2010. An approach for detecting execution phases of a system for the purpose of program comprehension. In *2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications*. IEEE, 207–214.

[39] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[40] Ana B Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. 2020. Tandem: A taxonomy and a dataset of real-world performance bugs. *IEEE Access* 8 (2020), 107214–107228.

[41] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. http://research.google.com/archive/papers/dapper-2010-1.pdf

[42] Richard Sites. 2021. *Understanding software dynamics*. Addison Wesley.

[43] Jonas Trümper, Johannes Bohnet, and Jürgen Döllner. 2010. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th international symposium on Software visualization*. 133–142.

[44] Jonas Trümper, Jürgen Döllner, and Alexandru Telea. 2013. Multiscale visual comparison of execution traces. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 53–62.

[45] Neil Walkinshaw, Sheeva Afshan, and Phil McMinn. 2010. Using compression algorithms to support the comprehension of program traces. In *Proceedings of the Eighth International Workshop on Dynamic Analysis*. 8–13.

[46] Augustine Wong, Paul Bucci, Ivan Beschastnikh, and Alexandra Fedorova. 2024. *NonSequitur Source Code and User Study Result Data for the paper "Making Sense of Multi-Threaded Application Performance at Scale with NonSequitur."*. https://doi.org/10.5281/zenodo.13446443

[47] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can we trust profiling results? Understanding and fixing the inaccuracy in modern profilers. In *Proceedings of the ACM International Conference on Supercomputing*. 284–295.

[48] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2012. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 199–208.

[49] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[50] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2023. Dissecting Overheads of Service Mesh Sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3620678.3624652

Received 2024-04-06; accepted 2024-08-18