Argus : Debugging Performance Issues in Modern Applications with Interactive Causal Tracing

Abstract

Prior systems used causal tracing, a powerful technique that traces low-level events and builds dependency graphs, to diagnose performance issues. However, they all assume that accurate dependencies can be inferred from low-level tracing by either limiting applications to using only a few supported communication patterns or relying on developers to manually provide dependency schema upfront for all involved components. Unfortunately, based on our own study and experience of building a causal tracing system for macOS, we found that it is extremely difficult, if not impossible, to build accurate dependency graphs. We report patterns such as data dependency, batch processing, and custom communication primitives that introduce imprecision. We present Argus, a practical system for effectively debugging performance issues in modern desktop applications despite the imprecision of causal tracing. Argus lets a user easily inspect current diagnostics and interactively provide more domain knowledge on demand to counter the inherent imprecision of causal tracing. We implemented Argus in macOS and evaluated it on 11 real-world, open spinning-cursor issues in widely used applications. The root causes of these issues were largely previously unknown. Our results show that Argus effectively helped us locate all root causes of the issues and incurred 7% CPU overhead in its system-wide tracing.

1. Introduction

Today's web and desktop applications are predominantly parallel or distributed, making performance issues in them extremely difficult to diagnose because the handling of an external request is often spread across many threads, processes, and asynchronous contexts instead of in one sequential execution segment [20]. To manually reconstruct this graph of execution segments for debugging, developers have to sift through a massive amount of log entries and potentially code of related application components [13, 32, 28, 23, 30]. More often than not, developers give up and resort to guessing the root cause, producing "fixes" that sometimes make the matter worse. For instance, a bug in the Chrome browser engine causes a spinning (busy) cursor in macOS when a user switches the input method [8]. It was first reported in 2012, and developers attempted to add timeouts to work around the issue. Unfortunately, the bug has remained open for seven years and the timeouts obscured diagnosis further.

Prior work proposed what we call *Causal tracing*, a powerful technique to construct request graphs (semi-)automatically [31]. It does so by inferring (1) the beginning and ending boundaries of the execution segments (vertices in the graph) involved in handling a request; and (2) the causality between the segments (edges)—how a segment causes others to do additional handling of the request. Prior causal tracing systems all assumed certain programming idioms to automate inference. For instance, if a segment sends a message, signals a condition variable, or posts a task to a work queue, it wakes up additional execution segments, and prior systems assume that wake-ups reflect causality. Similarly, they assume that the execution segment from the beginning of a callback invocation to the end is entirely for handling the request that causes the callback to be installed [31, 24]. Compared to debuggers such as `spindump` that capture only the current system state, causal tracing is quite effective at aiding developers to understand complex causal behaviors and pinpoint real-world performance issues.

Unfortunately, based on our own study and experience of building a causal tracing system for the commercial operating system macOS, we found that modern applications frequently violate these assumptions. Hence, the request graphs computed by causal tracing are inaccurate in several ways. First, an inferred segment may be larger than the actual event handling segment due to batch processing. Specifically, for performance, an application or its underlying frameworks may bundle together work on behalf of multiple requests with no clear distinguishing boundaries. For instance, WindowServer in macOS sends a reply for a previous request and receives a message for the current request using one system call `mach_msg_overwrite_trap`, presumably to reduce user-kernel crossings.

Second, the graphs may be missing numerous causal edges. For instance, consider data dependencies in which the code sets a flag (e.g., "`need_display` = 1" in macOS animation rendering) and later queries the flag to process a request further. This pattern is broader than ad hoc synchronization [27] because data dependency occurs even within a single thread (such as the buffer holding the reply in the preceding WindowServer example). Although the number of these flags may be small, they often express critical causality, and not tracing them would lead to many missing edges in the request graph. However, without knowing where the flags reside in memory, a tool would have to trace all memory operations, incurring prohibitive overhead and adding many superfluous edges to the request graph.

Third, in any case, many inferred edges may be superfluous because wake-ups do not necessarily reflect causality. Consider an `unlock()` operation waking up an thread waiting in `lock()`. This wake-up may be just a happens-stance and the developer intent is only mutual exclusion. However, the actual semantics of the code may also enforce a causal order between the two operations.

We believe that, without detailed understanding of application semantics, request graphs computed by causal tracing are *inherently* inaccurate and both over- and under-approximate reality. Although developer annotations can help improve precision [12, 25], modern applications use more and more third-party libraries whose source code is not available. In the case of tech-savvy users debugging performance issues such as a spinning cursor on her own laptop, the application's code is often not available. Given the frequent use of custom synchronizations, work queues, and data flags in modern applications, it is hopeless to count on manual annotations to ensure accurate capture of request graphs.

In this work, we present Argus, an interactive system for debugging performance issues in modern desktop applications despite the imprecision of causal tracing. Argus does not construct a per-request causal graph which requires extremely accurate causal tracing. Instead, it captures an approximate event graph for a duration of the system execution to aid diagnosis, and tracks both true causality edges as well as weak ones that may or may not reflect causality. It keeps humans in the loop, as a debugger should rightly do, and lets users easily inspect current diagnostics and guide the next steps to counter the inherent imprecision of causal tracing. Specifically, during debugging, Argus queries users a judicially few times to (1) resolve a few inaccurate edges that represent false dependencies, (2) select one out of several vertices as a baseline to the buggy vertex in question; and (3) identify potential data flags. Argus represents a dramatically different approach in the design space of causal tracing because it enables users to provide necessary schematic information on demand, as opposed to full manual schema upfront for all involved applications and daemons [12].

We implemented Argus in macOS, a widely used commercial operating system. macOS is closed-source, as are its common frameworks and many of its applications. This environment therefore provides a true test of Argus. We address multiple nuances of macOS that complicate causal tracing, and built a system-wide, low-overhead, always-on tracer. Argus enables users to optionally increase the granularity of tracing (e.g., logging call stacks and instruction streams) by integrating with existing debuggers such as `lldb`.

We evaluated Argus on 11 real-world, open spinning-cursor issues in widely used applications such as the Chromium browser engine and macOS System Preferences, Installer, and Notes. The root causes of all 11 issues were previously unknown to us and, to a large extent, the public. Our results show that Argus is effective: it helped us non-developers of the applications find all root causes of the issues, including the Chromium issue that remained open for seven years. Argus needs only less than 3 user queries per issue but they are crucial in aiding diagnosis. Argus is also fast: its systems-wide tracing incurs only 7% CPU overhead overall.

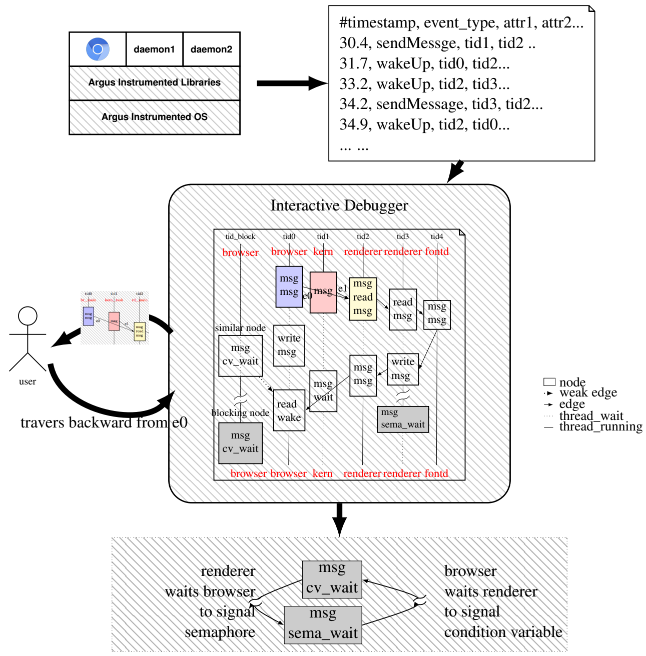

Figure 1: Argus Work Flow

This paper makes the following contributions: our conceptual realization that causal tracing is inherently inaccurate and that interactive causal tracing is superior than prior work in debugging performance issues in modern applications; our system Argus that performs system-wide tracing in macOS with little overhead and handles several macOS trickeries that complicate causal tracing; and our results diagnosing real-world spinning cursors and finding root causes for performance issues that have remained open for seven years.

This paper is organized as follows. In Section 2, we present an overview of using Argus and a Chromium example. Section 4 describes our event graph from causal tracing, and Section 5 describes our tracing implementation and tools for user interaction. In Section 6 we present other case studies, and Section 7 contains performance evaluation. We summarize related work in Section 8, and end with conclusion in Section 9.

2. Overview

2.1. Argus Work Flow

In this section, we describe the steps a user takes to investigate a performance anomaly with Argus. Figure 1 shows Argus's work flow with an example of a user investigating a performance problem in Chromium. The system wide tracing tool, which collects data from Argus instrumented library and kernel, generates logs. They are transformed into an graph in Argus's graph construction component. Central to our system is our *event graph*, a generalized control-flow graph which includes inter-thread and inter-process dependencies. The generated graph is used by the interactive debugger for causal path slicing and diagnosis. Argus supports interactive search, by providng information to the user and asking for decision, in case of multiple predecessors in a vertex. As shown in Figure 1, the debugger asks user to choose one edge in a subgraph. After this step, Argus performs diagnosis algorithm and reports the root cause vertices. In the example, the root cause is two vertices which form a circular wait acrossing multiple threads.

Next, we describe how Argus assists the user to diagnose a performance issue.

2.2. Diagnosis with Graph

Consider a common performance bug on macOS, the *spinning cursor*, which indicates the current application's main thread has not processed any UI events for over two seconds. To initialize debugging a spinning cursor, Argus first constructs an event graph from the system-wide event log recorded. It then queries the event graph to find the ongoing event in the application's main thread concurrent to the display of the spinning cursor. Given the event graph and the spinning vertex, Argus runs Algorithm 1 to interactively pinpoint the root cause.

Specifically, upon examining what the main thread is actually doing, there are three potential cases.

- **LongRunning** (lines 3 - 5). The main thread is busy performing lengthy CPU operations. This case is the simplest, and Argus traverses the event graph backwards to find a slice originating from the offending UI event to the long running CPU operations. This slice is particularly useful for further diagnosing the bug. As shown in FunctionXXX, Argus may encounter vertices with multiple incoming edges or weak edges that may not reflect causality when traversing the graph. It queries the user to resolve them.
- **RepeatedYield** (lines 6 - 11). The main thread is in a yield loop, which is highly indicative it is waiting on a data flag (e.g., "while(!done) thread_switch();"). If Argus cannot find any record of data flags in the spinning vertex, it terminates debugging by prompting the user to identify data flags and re-trace the application. Here we assume that the performance issue reproduces with a reasonable probability because, fortunately, a one-off issue that never reproduces is not as annoying as one that occurs frequently. If Argus finds the data flag the spinning vertex is waiting for, it falls through to the next case.
- **LongWait** (lines 12 - 22). The main thread is in a lengthy blocking wait and the wake-up has been missing. Argus handles this case by finding a baseline scenario where the wake-up indeed arrives, and then figures out which wake-up edge is missing in the spinning scenario along the expected wake-up path. Specifically, Argus first finds a similar vertex to the spinning one based solely on the semantical events such as system calls in each vertex. It then traverses backwards from the similar vertex to find the baseline wake-up path. For each thread in the wake-up path, it examines the vertex in the thread right before the spinning vertex waits. If this vertex is also abnormal, Argus appends it to the path of root cause vertices, and applies Function DiagnoseXX recursively diagnose "the culprit of the culprit." For each such vertex, it queries the user to determine whether to proceed or stop because based on our experience the user needs to inspect only a few vertices to find the root cause.

Based on our results and experience, the first case is the most common, but the second and third represent more severe bugs. Long-running CPU operations tend to be more straight-

Algorithm 1 Diagnosis algorithm.

Input: g - EventGraph; spinning_vertex- the vertex in the UI thread when the spinning cursor occurs
Output: root_cause_vertices-collecting root casuse vertices for user inspect

```
1:  function DIAGNOSE(g, spinning_vertex)
2:      switch spinning_vertex.block_type do
3:          case LongRunning
4:              slice ← InteractiveSlice(spinning_vertex)
5:              return vertex contains UI event
6:          case RepeatedYield
7:              if DataFlagEvent ∉ {event types in spinning_vertex } then
8:                  Require users to annotate data flag
9:                  abort()
10:             end if
11:             /* Fall through */
12:         case LongWait
13:             similar_vertex ← vertex has similar event sequence to spinning_vertex
14:             baseline_path ← InteractiveSlice(similar_vertex)
15:             for each t in {threads in baseline_path} do
16:                 vertex_t ← vertex in t before spinning_vertex gets spinning
17:                 if vertex_t ∈ {LongRunning, RepeatedYield, LongWait} then
18:                     root_cause_vertices.append(vertex_t)
19:                     root_cause_vertices.append(Diagnose(g, vertex_t)
20:                 end if
21:                 /* if t is normal running, disgnose the next thread */
22:             end for
23:     end switch
24:     return root_cause_vertices
25: end function

26: function INTERACTIVESLICING(g, vertex)
27:     loop
28:         path_slice.append(vertex)
29:         if vertex has 1 incoming edge then
30:             vertex ← predecessor vertex
31:         else if vertex has multiple incoming edges then
32:             vertex ← ask user to pick from predecessors
33:         else if vertex had weak edges then
34:             vertex ← ask user to pick from predecessors
35:         else
36:             /* The first vertex of current thread */
37:             return path_slice
38:         end if
39:         if vertex is invalid then
40:             /* user chooses to stop traversal with invalid input */
41:             return path_slice
42:         end if
43:     end loop
44: end function
```

forward to diagnose with existing tools such as `spindump` except they do not connect CPU operations back to UI events. Repeated yielding or long waiting cases involve multiple threads and processes, and are extremely hard to understand and fix even for the application's original developers. Therefore, issues remain unaddressed for years and significantly impact the user experience. Algorithm 1 is semi-automated but can integrate user input to leverage hypotheses or expert knowledge as to why a hang may occur. Our results show that user inputs, albeit few, are crucial in this process (§6).

2.3. Chromium Spinning Cursor Example

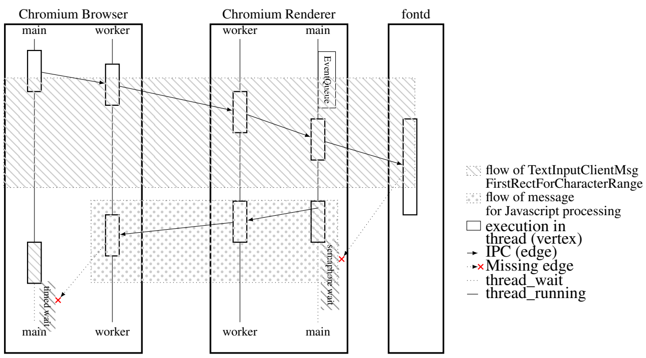

Figure 2: Chromium Example

One of the authors experienced first-hand the aforementioned performance issue in Chromium, an open-source browser engine that powers Google Chrome and, starting recently, Microsoft Edge [7]. She tried to type in the Chromium search box a Chinese word using SCIM, the default Chinese Input Method Editor that ships with MacOS. The browser appeared frozen and the spinning cursor occurs for a few seconds. Afterwards everything went back to normal. This issue is reproducible and always ruins her experience, but it is quite challenging to diagnose because two applications Chromium and SCIM and many daemons ran and exchanged messages. This issue was reported by other users for other non-English input methods, too.

To diagnose this issue with Argus, the author started system-wide tracing, and then reproduced the spinning cursor with a Chinese search string typed via SCIM while the page was loading. It produced normal cases for the very first few characters, and the browser got blocked with the rest input as spinning cases. The entire session took roughly five minutes. She then ran Argus to construct the event graph. The graph was highly complex, with 2,749,628 vertexes and 3,606,657 edges, almost fully connected. It spanned across 17 applications; 109 daemons including `fontd`, `mdworker`, `nsurlsessiond` and helper tools by applications; 126 processes; 679 threads, and 829,287 IPC messages. Given the scale of the graph and the diverse communication patterns, it would be extremely challenging for prior automated causal tracing tools [10, 31, 11, 15] because they handle a fairly limited set of patterns. Tools that require manual schema [12, 25], would be prohibitive because developers would have to provide schema for all involved applications and daemons.

Next she ran Argus to find the spinning vertex in the main thread of the browser process. Argus returned a `Wait` event on condition variable with timeout that blocked the main thread for a few seconds. Thus Argus compares the spinning vertex to a similar one in normal case where the `Wait` was signaled quickly. Argus reported three, and confirmed with the user which one she wanted.

Argus then found the normal-case wake-up path which connects five threads, as is shown in the example in Figure 1. The browser main thread was signaled by a browser worker thread, which received IPC from a worker thread of `renderer` where the rendering view and WebKit code run. The worker thread is woken up by the `renderer` main thread, which in turn woken by fontd, the font service daemon. Argus further compared the wake-up path with the spinning case, and returned the `wait` event on semaphore in the `renderer` main thread, the culprit that delayed waking up the browser main thread over 4 seconds.

What caused the wait in the `renderer` main thread though? She thus continued diagnosis and recursively applied Argus to the wait in `renderer`, and got the wake-up path. The culprit that delayed the semaphore was the timeouts in the browser's main thread. At this point, a circular wait formed, as is shown in Figure 2. To understand what exactly happens in the situation, she inspected the full call stacks by Argus scripts, taking the reported vertice from the renderer and the browser as input. Inspection reveals that the `renderer` requested the browser's help to render Javascript and waited for reply with semaphore. The browser was waiting for the `renderer` to return the string bounding box and the `renderer` was waiting for the browser to help render Javascript. This circular wait was broken by a timeout in the browser main thread (the `wait` on cv timeout was 1,500 ms). While the system was able to make progress, the next key press caused the spinning cursor to display for another 1,500 ms. The timeout essentially converted a deadlock into a livelock.

Finally, we verified our finding within chromium source code. Shortening the timeout interval in the main browser thread proportionally shortens the waiting of the main render thread on processing the javascript. Skipping certain javascripts processing in the renderer thread cuts down the success rate of spinning case reproducing.

2.4. Limitations

Argus is designed to support interactive debugging of performance issues. It sometimes requires the user to reproduce a performance issue so Argus can capture more fine-grained event traces such as accesses to data flags. Fortunately, a performance issue that almost never reproduces is probably not as annoying as one that occurs frequently.

```
//worker thread in fontd:
//enqueue a block
block = dispatch_mig_sevice;
dispatch_async(block);
```

```
//main thread in fontd:
//dequeue blocks
block = dequeue();
dispatch_execute(block);
```

```
//implementation of dipatch_mig_server
dispatch_mig_server()
for(;;) //batch processing
  mach_msg(send_reply, recv_request)
  call_back(recv_request)
  set_reply(send_reply)
```

Figure 3: Dispatch message batching

```
//inside a single thread
while() {
  CGXPostReplyMessage(msg) {
  // send _gOutMsg if it hasn't been sent
    push_out_message(_gOutMsg)
    _gOutMsg = msg
    _gOutMessagePending = 1
  }
  CGXRunOneServicePass() {
    if (_gOutMessagePending)
      mach_msg_overwrite(MSG_SEND | MSG_RECV, _gOutMsg)
    else
      mach_msg(MSG_RECV)
    ... // process received message
  }
}
```

Figure 4: Batching in event processing

```
//Worker thread:
//needs to update UI:
obj->need_display = 1
```

```
//Main thread:
//traverse all CA objects
if(obj->need_display == 1)
  render(obj)
```

Figure 5: CoreAnimation shared flag

We implemented Argus in the closed-source macOS which presents a harsh test for Argus, but we have not ported Argus to other operating systems yet. It is possible that the ideas and techniques do not generalize to other operating systems. However, modern operating systems share many similarities, and inspire each others' designs, so we are hopeful that the ideas in Argus are generally applicable. Similarly, the applications and performance issues used in our evaluation may be non-representative, though we strive to cover a diverse set of common applications ranging from browsers to text editors.

3. Inherent Inaccuracies in Causal Tracing

As explained in §1, causal tracing builds a graph to connect execution segments on behalf of a request that spread across separate threads and processes. Based on our experience building a causal tracing system on commercial, closed-source macOS, we believe such graphs are inherently inaccurate and contain both *over-connections* – edges that do not really map to causality) – and *under-connections* – missing edges between two vertices with one causally influencing the other. In this section, we present several inherently inaccurate patterns we observed and their examples in macOS.

3.1. Over Connections

Over connections usually occur when (1) intra-thread boundaries are missing due to unknown batch processing programming paradigms or (2) superfluous wake-ups that do not always imply causality.

Dispatch message batching While traditional causal tracing assumes the entire execution of a callback function is on behalf of one request, we found some daemons implement their service loop inside the callback function and create false dependencies. In the code snippet below from the `fontd` daemon, function `dispatch_execute` is installed as a callback to a work from dispatch queue. It subsequently calls `dispatch_mig_server()` which runs the typical server loop and handles messages from different apps.

To avoid incorrectly linking many irrelevant processes through such batching processing patterns, Argus adopts the aforementioned heuristics to split an execution segment when it observes that the segment sends out messages to two distinct processes. Any application or daemon can implement its own server loop this way, which makes it fundamentally difficult to automatically infer event handling boundaries.

Batching in event processing Message activities inside a system call are assumed to be related traditionally. However, to presumably save on kernel boundary crossings, WindowServer MacOS system daemon uses a single system call to receive data and send data for an unrelated event from differnt processed in its event loop. This batch processing artificially makes many events appear dependent.

Mutual exclusion In a typical implementation of mutual exclusion, a thread's unlock operation wakes up a thread waiting in lock. Such a wake-up may be, but is not always, intended as causality. However, without knowing the developer intent, any wake-up is typically treated as causality by traditional causal tracing tools.

3.2. Under Connections

We observe that under connections mostly result from missing data dependencies. This pattern is more general than shared-memory flags in ad hoc synchronization [27] because it occurs even within a single thread.

Data dependency in event processing The code for Batching in event processing above also illustrates a causal linkage caused by data dependency in one thread. WindowServer saves the reply message in variable `_gOutMsg` inside function `CGXPostReplyMessage`. When it calls `CGXRunOneServicePass`, it sends out `_gOutMsg` if there is any pending message.

CoreAnimation shared flags As shown in the code snippet below, worker thread can set a field `need_display` inside a CoreAnimation object whenever the object needs to be repainted. The main thread iterates over all animation objects

```
//NSEvent thread:
CGEventCreateNextEvent() {
  if (sCGEventIsMainThreadSpinning == 0x0)
    if (sCGEventIsDispatchToMainThread == 0x1)
      CFRunLoopTimerCreateWithHandler{
        if (sCGEventIsDispatchToMainThread == 0x1)
          sCGEventIsMainThreadSpinning = 0x1
          CGSConnectionSetSpinning(0x1);
      }
}
```

```
//Main thread:
{
  ... //pull events from event queue
  Convert1CGEvent(0x1);
  if (sCGEventIsMainThreadSpinning == 0x1){
      CGSConnectionSetSpinning(0x0);
      sCGEventIsMainThreadSpinning = 0x0;
      sCGEventIsDispatchedToMainThread = 0x0;
  }
}
```

Figure 6: Spinning Cursor Shared Flags

and reads this flag, rendering any such object. This shared-memory communication creates a dependency between the main thread and the worker so accesses to these field flags need to be tracked.

Spinning cursor shared flag As shown in Figure 6, whenever the system determines that the main thread has hung for a certain period, and the spinning beach ball should be displayed, a shared-memory flag is set. Access to the flag is controlled via a lock, i.e. the lock is used for mutual exclusion, and does not imply a happens before relationship.

4. Handling Inaccuracies

In this section, we first describe the basics of Argus event graphs (§4.1), and then discuss how Argus mitigates over-connections (§4.2) and under-connections (§4.2) in them.

4.1. Event Graph Basics

To construct event graphs, Argus collects three categories of events in its systems-wide event logs. The first category contains semantical events, such as system calls, call stacks collected when certain operations such as macOS message operations are run, and user actions such as key presses. These events indicate what the developer intents might be, and are stored as contents in each vertex in the event graph. They are primarily for providing information to user during diagnosis and for finding similar vertices (§2).

The second category of events are boundary events that mark the beginning and ending of execution segments or vertices in the graph. Argus handles common callback invocations such as `dispatch_invoke` and `runloop_invoke` and mark their entry and return as boundaries.

The third category of events are for forming edges in the graph. For instance, an operation that installs a callback is connected to the execution of the callback. A message send is connected to a message receive. The arming of a timer is connected to the execution of the timer callback. A unique design in Argus is to trace general wake-up and wait operations inside the kernel to ensure coverage across many diverse user-level, possibly custom wake-up and wait operations because their implementations almost always use kernel wake-up wand wait. This approach necessarily includes spurious edges in the graph, including those due to mutual exclusion; Argus handles them by querying the user when it encounters a node with multiple incoming causal edges during diagnosis (see §2). We also observed that a waiting kernel thread is frequently woken up to perform tasks such as interrupt handling and scheduler maintenance; Argus recognizes them and culls them out from the graph automatically.

Compared to tools such as `spindump` that capture only the current system state, event graphs capture the causal path of events, enabling users to trace across threads and process to events happened in the past (hence cannot be captured by `spindump`) that explain present anomalies.

4.2. Mitigating Over-Connections

From a high-level, Argus deals with over-connections by heuristically splitting an execution segment that appears mixing handling of multiple requests. It adds weak causal edges between the split segments in case the splitting was incorrect. When a weak edge is encountered during diagnosis, it queries the user to decide whether to follow the weak edge or stop (§2).

Specifically, Argus splits based three criteria. First, Argus recognizes a small set of well-known batch processing patterns such as `dispatch_mig_server()` in §3 and splits the batch into individual items. Second, when a wait operation such as `recv()` blocks, Argus splits the segment at the entry of the blocking wait. The rationale is that blocking wait is typically done at the last during one step of event processing. Third, if a segment communicates to too many peering processes, Argus splits the segment when the set of peers differs. Specifically, for each message, Argus maintains a set of two peers including (1) the direct sender or receiver of the message and (2) the beneficiary of the message (macOS allows a process to send or receive messages on behalf of a third process). Argus splits when two consecutive message operations have non-overlapping peer sets.

4.3. Mitigating Under-Connections

Under-connections are primarily due to data dependencies. Currently Argus queries the user to identify the memory locations of the data flags.It is conceivable to leverage memory protection techniques to infer them automatically, as demonstrated in previous record-replay work [22, 17], it is out of the scope of this paper and we leave it for future work. Currently, to discover a data flag, the user re-runs the application with Argus to collect instruction traces of the concurrent events in both the normal and spinning cases and detects where the control flow diverges. She then reruns the application with Argus to collect register values for the basic blocks before the divergence and uncovers the address of the data flag. Once the user identifies a data flag, Argus traces it using either binary instrument, such as the `need_display` flag in CoreAnimation (§3), or with watchpoints. Argus add a causal edge between a write to a data flag to the corresponding read to the flag.

5. Implementation

In this section, we discuss how Argus collects tracing events from both kernel and libraries, and user interactions which Argus leverages to ease diagnosis.

5.1. Event Tracing

Current macOS systems support a system-wide tracing infrastructure built by Apple [1]. By default, the infrastructure temporarily stores events in memory and flushes them to screen or disk when an internal buffer is filled. We extended this infrastructure to support larger-scale tests and avoid filling up the disk with a file-backed ring buffer. Each log file is 2GB by default, which users can override. This size corresponds corresponds to approximately 19 million events (about 5 minutes with normal operations).

The default tracing points in macOS provide too limited information to enact causal tracing. As a result, we instrument both the kernel [2] (at the source level) and key libraries (at the binary level), to gather more tracing data. We instrumented the kernel with 1,193 lines of code, and binary-instrumented the following libraries: `libsystem_kernel.dylib`, `libdispatch.dylib`, `libpthread.dylib`, `CoreFoundation`, `CoreGraphics`, `HIToolbox`, `AppKit` and `QuartzCore` in 57 different places.

5.2. Instrumentation

Most libraries as well as many of the applications used day-to-day are closed-source in macOS. To add tracing points to such code, techniques such as library preloading to override individual functions are not applicable on macOS, as libraries use two-level executable namespace [3]. Hence, we implemented a binary instrumentation mechanism that allows developers to add tracing at any location in a binary image.

Like Detour [21], we use static analysis to decide which instrumentation to perform, and then enact this instrumentation at runtime. Firstly, the user finds a location of interest in the image related to a specific event by searching a sequence of instructions. Then the user replaces a call instruction to invoke a trampoline target function, in which we overwrite the victimized instructions and produce tracing data with API from Apple. All of the trampoline functions are grouped into a new image, as well as an initialization function which carries out the drop-in replacement. Then command tools from Argus helps to configure the image with the following steps: (1) re-export all symbols from the original image so that the original code can be called like an shared library; (2) rename the original image, and use original name for the new one to ensure the modifications are properly loaded; (3) invoke the initialization function externally through `dispatch_once` during the loading.

5.3. Tracing Data Flags

As described in (§3), under-connection due to the missing data dependency requires users' interaction. Users specify that reads and writes to a given global variable should be considered data dependencies. Global-variable tracing is possible through Argus's binary rewriting, but we also provide a simple command line tool which uses watchpoint registers to record `data_flag_write` and `data_flag_read` events. The tool takes as input the process ID, path to the relevant binary image, and the symbol name of the global variable. Here is a simple example of how a user would ask Argus to trace `_gOutMsgPending`:

```
./bp_watch pidofWindowServer Path/to/CoreGraphics \
    _gOutMsgPending
```

At load time, Argus hooks the watchpoint break handler in CoreFoundation to make sure that it is loaded correctly into the address space of our target application. The handler invokes Apple's event tracing API to record the value of the data flag and the operation type (read or write).

5.4. Tracing Instructions and Calls

Users may need to gather more information, such as individual instructions and call stacks, to come up with and verify a binary patch. Argus integrates with `lldb` to capture this information and add it to the corresponding vertices in the event graph.

We gather call stacks only at relevant locations, to reduce the data collection overhead. Our `lldb` scripts go through the instructions of apps and frameworks step by step to capture the parameters tainted by user inputs. Only at each beginning of a function call does the script record a full call stack. We also step over and record the return value of APIs from low-level libraries (i.e. those with the filename extension `.dylib`).

The combination of instruction-level tracing and occasional call-stacks offers more than enough detail to diagnose even the most arcane issues, and in our experience has been very helpful in multiple steps of an Argus diagnosis.

5.5. Find Baseline Scenario

Argus leverages a baseline scenario to discover missing wake-up edges in **Long Wait** cases.

The baseline scenario is identified from the vertexes which share the same high level semantics as the spinning vertex, but exposes different execution results. Argus recognizes high level semantics of vertices with semantical events, including system calls, call stacks, user actions. Their sequential order and runtime unrelated attributes are treated as hallmarks.

For example, Argus compares vertices with system call numbers and symbol names in call stacks. To improve the accuracy, Argus also checks the events forming edges, including messages, dipatch queue operations, runloop operations, `data_flag_read` and `data_flag_write`, as they might reflect the low-level behaviors of developers' intent. More over, a user can also instruct Argus to inspect proceeding vertices for accuracy purpose. By default, Argus discriminate the execution results with the time cost and wait results.

If multiple similar vertex are identified, Argus usually asks users for a confirmation, or choose the most recent one heuristically.

6. Case Studies

In this section, we demonstrate how Argus helps to diagnose 11 spinning-cursor cases in 11 popular applications. Table 1 describes these spinning-cursor cases, which are detailed in the sections that follow. We compared Argus with traditional causual tracing methods on edges and vertices Argus mitigated. We studied the ratios of over-connection filted with message peers heuristics, and incognito under-connections disclosed by data flags and wait hueristics. Even with our technique, the filted graph remains too imprecise for automatic causual tracing. Fortunately, user interactions compensates the inaccuracy in causual path slicing, while not overwhelming the diagnosis process.

Our results in Table 2 show 0 to 3 user queries usually suffice to find root cause path precisely. Although complex softwares like 7-MSWord requires 22 queries, many repeated patterns can be easily identified by uers, as discussed below. The paths exclude much noise, and are shorter and easier to inspect, compared to the traditional slicing automatically with heuristics.

How Argus Detects Spinning Cursors When the spinning cursor shows up, a hang reporting tool, `spindump` usually kicks in automatically to sample callstacks for debugging. To figure out the spinning vertex in the main thread, we turn to the event graph, and slice path backward from the launch of `spindump`.

The path shows that `spindump` is launched after receiving a message from `WindowServer`, which received a message from the `NSEvent thread` of the freezing app. The call stack attached to the messages further reveals `NSEvent thread` per process fetches `CoreGraphics` events from `WindowServer`, converts and creates `NSApp events` for the main UI thread. If the main thread is not spinning, a timer is armed to count down for the NSApp event. If the main thread fails to process it before the timer fires, `NSEvent thread` sends a message to `WindowServer` via the API "`CGSConnectionSetSpinning`" from the timer handler, and `WindowServer` notifies the `CoreGraphics` to draw a spinning cursor over the application window. Moreover, while exploiting how `NSEvent thread` communicates with the main thread, we found two variables, "`is_mainthread_spinning`" and "`dispatch_to_mainthread`", indicating the main thread status. As a result, Argus can make use of either the API or the data flag to identify the spinning vertex in the main thread.

Bug ID	Application	Bug description
0-Chormium	Chormium	Typing non-english in search box causes webpage freeze.
1-SystemPref	System Preferences	Disabling an online external monitor and rearranging windows causes System Preferences freeze.
2-SequelPro	Sequel Pro	Lost connection freezes the APP.
3-TexStudio	TeXStudio	Modification on bib file with vim causes its main window hang.
4-Installer	Installer	Moving cursor out of an authentication window causes freeze.
5-Notes	Notes	Launching Notes where stores a long note before causes freeze.
6-TextEdit	TextEdit	Copying text over 30M causes freeze.
7-MSWord	Microsoft Words	Copying a document over 400 pages causes hang.
8-SlText	Sublime Text	Copying in a file over 49000 lines causes freeze.
9-TextMate	TextMate	Pating text over 4000 lines causes freeze.
10-CotEditor	CotEditor	Pasting in file with 4000 lines context causes freeze.

Table 1: Bug Descriptions. We assign each bug in Column Bug ID to ease discussion

6.1. Long Running

In this section, we discuss the cases where the spinning vertex is busy on the CPU. Most of the text editing apps fall into this bug category. We studied TeXstudio, TextEdit, Microsoft Word, Sublime Text, Text Mate and CotEditor to reveal their root causes.

3-TeXStudio TeXstudio [4] is an integrated writing environment for creating LaTeX documents. We noticed a user reported spinning cursor when he modified his bib file. Although the issue was closed by the developer, due to insufficient information to reproduce the bug, we reproduced it with a large bib file opened in a tab. Each time we touched the file through another editor, vim for example, the application window showed a spinning cursor.

Argus recognizes the spinning vertex belongs to the category of **Long Running**. Slicing causal path from the vertex, Argus reaches daemon "`fseventd`" and figures out that the long-running function is invoked by a callback function from this daemon. The advantage of Argus over other debugging tools is it helps to narrow down the root cause with the path. If the user's bug report had included details captured with Argus, it may have provided the developer with enough information to reproduce the bug successfully.

6-TextEdit TexEdit is a simple word processing and text editing tool shipped by Apple, which often hangs on the editing of large files.

Bug ID	rate of connections filted by Argus msg heuristics	rate of connections added by share flag and Argus wait hueristics	# of user provided data flag	length of Argus baseline/spinning path slicing	# of user interaction	length of auto baseline/spinning path slicing
1-SystemPref	0.0056	0.0248	2	2	1	30
2-SequelPro	0.0049	0.0035	0	5	2	264
3-TeXStudio	0.0243	0.0058	0	6	3	44
4-Installer	0.0439	0.0283	0	6	2	36
5-Notes	0.0297	0.1153	0	10	2	42
6-TextEdit	0.0797	0.0072	0	21	3	21
7-MSWord	0.0672	0.0104	0	67	22	136
8-SlText	0.0407	0.0092	0	3	1	3
9-TextMate	0.0215	0.0218	0	3	0	3
10-CotEditor	0.0481	0.0532	0	4	1	6

Table 2: Graph Comparison

Argus reveals the same causal path with heuristics as with user interaction. We observed a communicating pattern in the vertices where a kernel thread was woken up from blocking IO by another kernel thread; and it processed the timer armed by TextEdit and woke up one of its threads. The first incoming edge is from the second kernel thread, and the second incoming edge is from TextEdit(from vertex where the timer armed to where it is processed). Users can make decision on the vertex base on the event sequence, which implies the story: TextEdit first arms the timer for IO work, then kernel threads work for it, and finally it processes the timer and wakes up TextEdit when finished.

It is not surprising because the pattern of vetices in the case fits the hueristics in Argus, which chooses the most recent incomming edge. Although the automatic heuristics works for the particular simple tool, it is not general enough to make decision for all patterns on complex softwares.

7-MSWord Microsoft Word is a large and complex piece of software. Argus can analyze the event graph, but it identifies multiple possible root causes: the length of path interactively sliced from the spinning vertex is 67, while the automatic slicing generates a path of 136 vertices.

We compared the path and find that the earliest difference exists in the predecessor of the third vertex in backward paths. In the vertex, user can learn from the callstack that `Microsoft Word` launches a service `NSServiceControllerCopyServiceDictionarie` after being woken by another `Microsoft Word` thread; this thread then sends a message to `launchd` to register the new service and waits for a reply message. With the most recent edge heuristics in automatic slicing, Argus chose `launchd` as its precedessor, but the user can more precisely identify that the execution segment is on behalf of the first thread. We rely on user interaction in this case to find the true root cause, since Argus has identified multiple possibilities.

Other Editing Apps Select, copy, paste, delete, insert and save are common operations for text editing. However, these operations on a large context usually trigger spinning cursors. Depending on their implementations, CotEditor and TextMate successfully avoid hangs on copy and selection operation. Argus can helps the developer to figure out the more efficient way to implement event handlers. We briefly list the reports from spinning vertex, including the event handler and most costly functions. We also list corresponding user input event from the path slicing in Table 3.

BUG-ID	costly API	UI
5-Notes	1)`NSDetectScrollDevices ThenInvokeOnMainQueue`	`system define event`
6-TexEdit	1)`[NSTextView(NSPasteboard) _write RTFDInRanges:toPasteboard:]` 2)`get_vImage_converter` 3)`get_full_conversion_code_fragment`	`key c`
7-MSWord	1)`-[NSPasteboard setData: forType:index:usesPboardTypes:]` 2)`_CFStringCreateImmutableFunnel3` 3)`platform_memmove` 4)`lseek`, 5)`fstat64`, 6)`fcntl`	`key c`
8-SlText	1)`px_copy_to_clipboard` 2)`__CFToUTF8Len`	`key c`
9-TextMate	1)`-[OakTextView paste:]` 2)`CFAttributedStringSet` 3)`TASCIIEncoder::Encode`	`key v`
10-CotEditor	1)`CFStorageGetValueAtIndex` 2)`-[NSBigMutableString characterAtIndex:]`	`key Return`

Table 3: Root cause of spinning cursor in editing Apps

6.2. Long Wait and Repeated Yield

In this section, we discuss the cases where the spinning vertex is blocking on wait event or yielding loop, corresponding to **Long Wait** and **Repeated Yield**.

1-SystemPreferences System Preferences provides a central location in macOS to customize system settings, inclduing configuring additional monitors. A tool called `DisableMonitor` [9] completes its function with enable/disable monitors online. We blocked on the spinning cursor while disabling an external monitor and rearranging windows in `Display` panel.

The log collected with Argus contains 1) baseline scenario where the displays are re-arranged with the enabled external monitor, and 2) spinning scenario in which we disable the external monitor with `DisableMonitor` and re-arrange the displays. The spinning vertex in the main thread is dominated by system calls, `mach_msg` and `thread_switch`, which falls into the category of **Repeated Yield**. We discovered two missing data flags with `lldb`, "`_gCGWillReconfigureSeen`" and "`_gCGDidReconfigureSeen`", which signify the status of configuring and break the loop of thead yielding. Argus learns from the baseline scenario that the main thread is responsible to set both of them after receiving specific datagrams from WindowServer. Conversly, setting of "`_gCGDidReconfigureSeen`" is missing in the spining case, where the main thread yields repeatedly to send messages to WindowServer for such datagram.

In conclusion, we discovered that the bug is inherent in the design of the CoreGraphics library, and would have to fixed by Apple. We verified this diagnosis by creating a dynamic binary patch with lldb to fix the deadlock. The patched library makes DisableMonitor work correctly, while preserving correct behavior for other applications.

2-SequelPro Sequel Pro [5] is a fast, easy-to-use Mac database management application for working with MySQL databases. It allows user to connect to database with a standard way, socket or ssh.

We experienced the non-responsiveness of Sequel Pro while its network connection got lost and it tried re-connections. The tracing data collected by Argus contains 1) a quick network connection during login, and 2) Sequel Pro lost connection for a while. Although Argus identified the spinning vertex and corresponding (baseline) similar vertex with ease, it can hardly get the correct causal path in the baseline scenario without a user's interaction. The backward slicing on vertex, which has multiple incomming edges, including one from kernel thread, makes it hard to rely on heuristics. Our interactively search is extremely helpful in the step, greatly reducing the noise in the path. Close examine of the spinning vertex based on the causual path tells that the main thread is waiting for the kernel thread, which in turn waits for the ssh thread.

Existing debugging tools like `lldb` and `spindump` could hardly figure out the root cause, in that both of them diagnose with only call stacks, missing the dependency across process boundaries.

4-Installer Installer [6] is an application included in macOS that extracts and installs files out of `.pkg` packages. When `Installer` pops up a window for privileged permission during the installation of `jdk-7u80-macosx-x64`, moving the cursor out of the popup window triggers spinning cursor.

As we put in the password directly before the round of triggering the spinning cusor, Argus successfully records the baseline scenario. Examing the spinning vertex and its similar vertex, Argus figures out the daemon `authd` blocks on semaphore. The blocking synchronous authentication of user's previledge in the main thread (user stops input) is the root casue, instead of the cursor moving handler. The result was verified with `lldb`. The main thread sends a synchromous message via `[IFRunnerProxy requestKeyForRights:askUser:]` to `authd`.

We also discovered a communication pattern in `Installer` underpinning the crucial of interactive debugging. It involves four vertices in four threads, vertex $Vertex_{main}$ in the main thread, and $Vertex_1$ to $Vertex_3$ in three worker threads. First, the main thread wakes up three worker threads. Then one worker thread is scheduled to run. At its end, another worker thread, which waits on mutex lock, is woken in $Vertex_2$, which in turn wakes up the next worker thread in $Vertex_3$. While Argus is slicing backward, $Vertex_3$ has two incoming edges: one is from $Vertex_{main}$, and the other one is from $Vertex_2$. Since users can peek the edges before making decision, they are likely to figure out that the three worker threads contend with mutex lock, and all of them are the heirs of $Vertex_{main}$.

	1st	2nd	3rd	4th	5th
without Argus	5.98	6.23	6.18	6.05	6.28
with Argus	6.29	6.01	6.09	6.28	6.01
average overhead	0.13%				

Table 4: Score From iBench

	kb/s	With Argus	Without Argus	overhead
bonnie++	read char	21922	22149	0.01
seqential	read block	226931	244089	0.07
	rewrite	246807	267491	0.08
	write char	22924	22936	0.00
	write block	4073361	4396387	0.07
seq	file create	17391	17381	0.00
	file delete	18089	19401	0.07
random	create	17472	17887	0.02
	delete	8849	9567	0.08
iozone	initial write	1199453	1318572	0.09
	rewerite	3663066	4059912	0.10
	average	-	-	0.05

Table 5: IO throughput with bonnie++ and iozone

7. Performance Evaluation

In this section we present the performance impact of the live deployment of Argus. We deploye Argus on a MacBookPro9,2, which has Intel Core i5-3210M CPU with 2 cores and 4 thread, 10GB DDR3 memory and a 1T SSD.

Argus has a very small space overhead with the configuration of its tracing tool. It uses the ring buffer with configured 2G by default to collect tracing events. The memory used to store events is fixed to 512M by Apple, which is pretty low with regards to the memory usage of modern applications. In the remaining of this section, we measure Argus's overhead overall with iBench scores, IO throughtput degradation with bonnie++, iozone and CPU overhead with chromium benchmarks.

iBench We first show the five runs of iBench with and without Argus to evaluate the overall performance. The machine is clean booted for each run, and the higher score means it perfoms better. As shown in Table4, their performance are almost of no difference, only 0.13% degradation on average.

Chromium Benchmark (in seconds)	with Argus			without Argus			Overhead		
	real	user	sys	real	user	sys	real	user	sys
system_health.memory_desktop	11592	18424	1821	11317	18401	1415	0.02	0.00	0.29
rasterize_and_record_micro.top_25	1579	2142	135	1654	2166	116	-0.05	-0.01	0.16
blink_perf	16210	17227	959	15877	16724	766	0.02	0.03	0.25
webrtc	726	2023	225	725	2130	168	0.00	-0.05	0.34
memory.desktop	1231	2238	267	1188	2200	190	0.04	0.02	0.41
loading.desktop.network_service	24580	52751	6294	23696	52327	4197	0.04	0.01	0.50
dromaeo	206	227	15	192	212	12	0.07	0.07	0.29
dummy_benchmark.histogram	49	48	8	33	36	4	0.50	0.32	0.96
v8.browsing_desktop	2462	4489	491	2325	4440	303	0.06	0.01	0.62
octan.desktop	112	142	8	98	124	5	0.14	0.15	0.44
speedometer	618	802	31	600	782	24	0.03	0.03	0.32
page_cycler_v2.typical_2	8020	14435	1453	7847	14215	1019	0.02	0.02	0.43
smoothness.oop_rasterization.top_25_smooth	864	1450	156	833	1412	126	0.04	0.03	0.24
AVERAGE		-			-		0.07	0.05	0.4

Table 6: Chromium benchmark

IO Throughput Next, we evaluate the IO throughput with bonnie++ and iozone. As shown in the Table 5, the throughputs of sequential read and write by characters with and without Argus are almost same. Read and write by block imposes less than 10% overhead in the both microbenchmarks, bonni++ and iozone. With the selected event types in our system, the tracing tool integrated in Argus only adds 5% IO overhead on average.

CPU We evaluate Argus's CPU overhead with chromium benchmarks by recording their time usage on real, user and sys. Although the sys time overhead is realatively high due to the tracing events usually across the kernel boundary, they are not tiggered too frequently in our daily software usage, including browsers. The time cost is mostly under 5%, except the `dummy_benchmark.histogram`. As shown in Table 6, the time overhead for real, user and sys are 7%, 5% and 40% respectively.

8. Related Work

While there is currently no system that can help users debug performance issues in closed-source applications on proprietary macOS, several active research topics are closely related.

Event tracing. Panappticon [31] monitors a mobile system and uses the trace to characterize the user transactions of mobile apps. Although it aims to track system-wide events and correlate them without developer input, it supports only two models of communication: work queue and thread pooling. AppInsight [24] instruments application to identify the critical execution path in a user transaction. It supports the event callback pattern, and does not trace across process or app boundaries. Magpie [12] monitors server applications in Windows with the goal to model the normal behaviors of a server application in response to a workload. This model further helps detecting anomalies statistically. Magpie requires a manual-written event schema for all involved applications to capture precise request graphs, whereas Argus has a simple, application-agnostic schema for system-wide tracing and enables users to provide more application-specific knowledge on demand.

Aguilela [10] uses timing analysis to correlate messages to recover their input-output relations while treating the application as a black box. XTrace, Pinpoint and etc [18, 13, 14] trace the path of a request through a system using a unique identifier attached to each request and stitch traces together with the identifier. Argus comes up violation patterns and does not assume the presence of a unified identifier in closed-source, third-party applications, frameworks, and libraries.

Performance anomaly detection. Several systems detect performance anomalies automatically. [19, 30] leverage the user logs and call stacks to identify the performance anomaly. [15, 26, 28, 16] apply the machine learning method to identify the unusual event sequence as an anomaly. [29] generates the wait and waken graph from sampled call stacks to study a case of performance anomaly.

These systems are orthogonal to Argus as Argus's goal is to diagnose an already-detected performance anomaly. These systems can help Argus by detecting more accurately when a performance issue arises.

9. Conclusion

Our key insight in this paper is that causal tracing is inherently imprecise. We have reported patterns we observed that pose big precision challenges to causal tracing, and built Argus, a practical system for effectively debugging performance issues in macOS applications despite the imprecision of causal tracing. To do so, it lets a user provide domain knowledge interactively on demand. Our results show that Argus effectively helped us locate all root causes of the issues, including a bug in Chromium, 01 and incurred 7% CPU overhead overall in its system-wide tracing.

References

[1] https://opensource.apple.com/source/system_cmds/system_cmds-671.10.3/trace.tproj.

[2] https://opensource.apple.com/source/xnu.

[3] http://mirror.informatimago.com/next/developer.apple.com/releasenotes/DeveloperTools/TwoLevelNamespaces.html.

[4] https://www.texstudio.org.

[5] https://www.sequelpro.com.

[6] https://en.wikipedia.org/wiki/Installer_(macOS).

[7] The Chromium Projects. https://www.chromium.org, 2008.

[8] Issue 115920: Response time can be really long with some IMEs (e.g. Pinyin IME (Apple), Sogou Pinyin IME). https://bugs.chromium.org/p/chromium/issues/detail?id=115920, 2012.

[9] https://github.com/Eun/DisableMonitor, 2018.

[10] Aguilera, Marcos K and Mogul, Jeffrey C and Wiener, Janet L and Reynolds, Patrick and Muthitacharoen, Athicha. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, 2003.

[11] Attariyan, Mona and Chow, Michael and Flinn, Jason. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation OSDI 12*, 2012.

[12] Barham, Paul and Donnelly, Austin and Isaacs, Rebecca and Mortier, Richard. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[13] Chen, Mike Y and Kiciman, Emre and Fratkin, Eugene and Fox, Armando and Brewer, Eric. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, 2002.

[14] Chow, Michael and Meisner, David and Flinn, Jason and Peek, Daniel and Wenisch, Thomas F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation OSDI 14)*, 2014.

[15] Cohen, Ira and Chase, Jeffrey S and Goldszmidt, Moises and Kelly, Terence and Symons, Julie. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, 2004.

[16] Du, Min and Li, Feifei and Zheng, Guineng and Srikumar, Vivek. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[17] Dunlap, George W and Lucchetti, Dominic G and Fetterman, Michael A and Chen, Peter M. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2008.

[18] Fonseca, Rodrigo and Porter, George and Katz, Randy H and Shenker, Scott and Stoica, Ion. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, 2007.

[19] Han, Shi and Dang, Yingnong and Ge, Song and Zhang, Dongmei and Xie, Tao. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[20] Harter, Tyler and Dragga, Chris and Vaughn, Michael and Arpaci-Dusseau, Andrea C and Arpaci-Dusseau, Remzi H. A file is not a file: understanding the I/O behavior of Apple desktop applications. *ACM Transactions on Computer Systems (TOCS)*, 2012.

[21] Galen Hunt and Doug Brubacher. Detours: Binaryinterception ofwin 3 2 functions. In *3rd usenix windows nt symposium*, 1999.

[22] King, Samuel T and Dunlap, George W and Chen, Peter M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[23] Nagaraj, Karthik and Killian, Charles and Neville, Jennifer. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[24] Ravindranath, Lenin and Padhye, Jitendra and Agarwal, Sharad and Mahajan, Ratul and Obermiller, Ian and Shayandeh, Shahin. AppInsight: mobile app performance monitoring in the wild. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation OSDI 12*, 2012.

[25] Reynolds, Patrick and Killian, Charles Edwin and Wiener, Janet L and Mogul, Jeffrey C and Shah, Mehul A and Vahdat, Amin. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[26] Saidi, Ali G and Binkert, Nathan L and Reinhardt, Steven K and Mudge, Trevor. Full-system critical path analysis. In *ISPASS 2008-IEEE International Symposium on Performance Analysis of Systems and software*, 2008.

[27] Xiong, Weiwei and Park, Soyeon and Zhang, Jiaqi and Zhou, Yuanyuan and Ma, Zhiqiang. Ad Hoc Synchronization Considered Harmful. In *OSDI*, 2010.

[28] Xu, Wei and Huang, Ling and Fox, Armando and Patterson, David and Jordan, Michael I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[29] Yu, Xiao and Han, Shi and Zhang, Dongmei and Xie, Tao. Comprehending performance from real-world execution traces: A device-driver case. In *ACM SIGPLAN Notices*, 2014.

[30] Yuan, Ding and Park, Soyeon and Huang, Peng and Liu, Yang and Lee, Michael M and Tang, Xiaoming and Zhou, Yuanyuan and Savage, Stefan. Be conservative: enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation OSDI 12)*, 2012.

[31] Zhang, Lide and Bild, David R and Dick, Robert P and Mao, Z Morley and Dinda, Peter. Panappticon: event-based tracing to measure mobile application and platform performance. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[32] Zhao, Xu and Rodrigues, Kirk and Luo, Yu and Yuan, Ding and Stumm, Michael. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation OSDI 16)*, 2016.