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 only 1% 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 [16]. 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 [10, 27, 23, 18, 25]. 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 [5]. 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 [26]. 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 [26, 19]. 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 [22] 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.

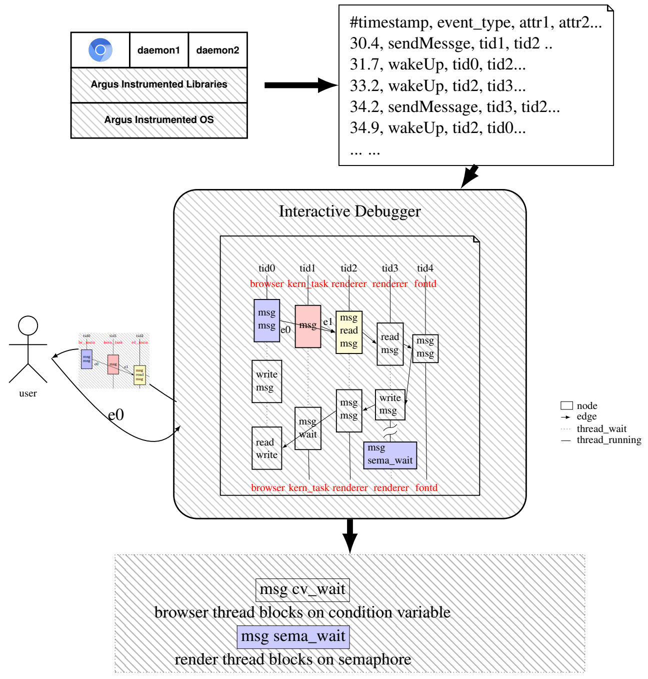
Third, in any case, many inferred edges may be superfluous because wake-ups do not necessarily reflect causality. Consider an `unlock()` operation waking up an thread waiting in `lock()`. This wake-up may be just a happens-stance and the developer intent is only mutual exclusion. However, the actual 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 [9, 20], 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 [9].

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 XXX user queries per issue but they are crucial in



```
#timestamp, event_type, attr1, attr2...
30.4, sendMessge, tid1, tid2 ..
31.7, wakeUp, tid0, tid2...
33.2, wakeUp, tid2, tid3...
34.2, sendMessage, tid3, tid2...
34.9, wakeUp, tid2, tid0...
... ...
```

msg cv_wait
browser thread blocks on condition variable
msg sema_wait
render thread blocks on semaphore

Figure 1: Argus Work Flow

aiding diagnosis. Argus is also fast: its systems-wide tracing incurs only 1% CPU overhead overall.

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, which consists of two phases. A user runs command "`Argus start`" to enter the system-wide tracing phase, within which Argus logs events as listed in Table **??** (§4.1). Whenever a user detects a performance issue such as a spinning cursor, she runs "`Argus debug`" to enter the diagnosis phase.

Central to our system is our *event graph*, a generalized control-flow graph which includes inter-thread and inter-process dependencies. Diagnosis and inferences are performed within this graph, in a semi-automated fashion: Argus performs searches to trace logical events as they flow through the system, and it judicially queries the user for guidance. 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 node, 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 XX-XX). 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 nodes 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 XX-XX). 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 node, 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 node is waiting for, it falls through to the next case.
- **LongWait** (lines XX-XX). 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 node to the spinning one based solely on the semantical events such as system calls in each node. It then traverses backwards from the similar node to find the baseline wake-up path. For each thread in the wake-up path, it examines the node in the thread right before the spinning node waits. If this node is also abnormal, Argus appends it to the path of root cause nodes, and applies Function DiagnoseXX recursively diagnose "the culprit of the culprit." For each such node, it queries the user to determine whether to proceed or stop because based on our experience the user needs to inspect only a few nodes to find the root cause.

Algorithm 1 Diagnosis algorithm.

Input: g - EventGraph; spinning_node - the node in the UI thread when the spinning cursor occurs
Output: root_cause_nodes-collecting root casuse nodes for user inspect

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

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

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 straightforward to diagnose with existing tools such as `Spindump` except they do not connect the 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 **??** is semi-automated but can integrate user input at each stage 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 (§**??**).

2.3. Chromium Spinning Cursor Example

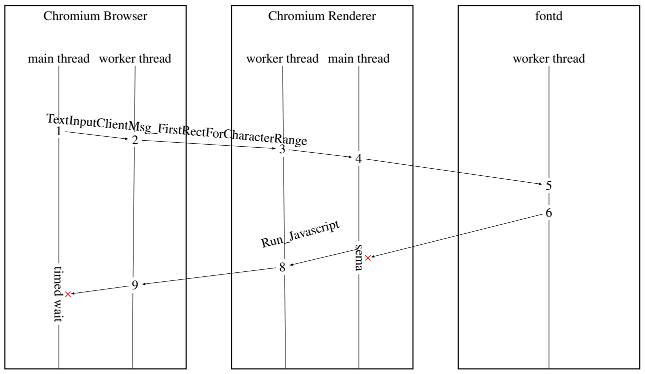


Figure 2: Chromium Example

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 [4]. 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 had 2,749,628 vertexes and 3,606,657 edges, almost fully connected. It spans 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 [7, 26, 8, 12] because they handle a fairly limited set of patterns. Tools that require manual schema [9, 20], would be prohibitive because developers would have to provide schema for all involved applications and daemons.

Next she ran Argus to find the spinning node 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 node to a similar one in normal case where the `Wait` was signaled quickly with Algorithm **??**. 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. 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 with the Algorithm **??**, 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. To understand what exactly happens in the situation, she inspected the full call stacks by Argus scripts, taking the reported nodes 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.

The finding was verified with 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.

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 good ideas tend to flow both ways, 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 causual 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 diffemt processed in its event loop. This batch processing artificially makes many events appear dependent.

```
//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)
```

```
//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
  }
}
```

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 [22] 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 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.

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

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

Spinning cursor shared flag As shown in Figure 3, 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 (§**??**) and under-connections (§**??**) 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 vertexes (§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 ř 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 [**?**], 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 (§**??**), or with watchpoints. Argus add a causal edge between a write to a data flag to the corresponding read to the flag.

5. Implementation

We now discuss how we collect tracing events from both kernel and libraries.

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 infras-

```
//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 3: Spinning Cursor Shared Flags

tructure to support larger-scale tests and avoid filling up the disk with a file-backed ring buffer. Subject to configuration, it allows at most 2GB of data per log, which corresponds to approximately 18,560,187 events (about 5 minute with normal operations).

The default tracing points in macOS provide too limited information to enact causal tracing. As a result, we both patch source code of kernel [2] and binary instrument libraries to gather more tracing data. In Argus, we patched the kernel with 1193 lines of code, and binary instrumented the libraries including: libsystem_kernel.dylib, libdispatch.dylib, libpthread.dylib, CoreFoundation, CoreGraphics, HIToolbox, AppKit and QuartzCore.

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 [17], 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 invokes 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. User Interaction

Tracing Custom Primitives As described in (§**??**), underconnection due to the missing share data dependency requires users' interaction. Argus provides a command line tool which sets the watch point registers to record share_flag_write and share_flag_read events in ad-hoc manner. This is one way users can easily collect the tracing events, and the more tech-savvy users can also instrument the binary. The tool takes the process id, path to image where the variable is defined and the symbol of the variable as input. We show the simple example how a user ask Argus to trace _gOutMsgPending in the following command.

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

Argus hooks the watch point break handler in CoreFoundation to make sure that it is loaded correctly into the address space of our target application. The handler invokes the event tracing API from Apple to record the value of the shared variable and the operation type: read or write.

Capturing Instructions for Diagnosis Since the output of Argus is the nodes and user input events suspected to cause the busy spinning, more detail may required to verify and fix the bug. After the offline analysis on the graph, Argus provides tools for user to exact the Back_trace events from the output and generate a script for conditional debugging.

The debugging scripts go through the instructions of apps and frameworks step by step to capture the parameters tainted by user inputs. At each beginning of a function call, the script records a full call stack for it. Considering the overhead and usefulness, it steps over and only record the return value of APIs from libraries with a filename extension .dylib.

The supplementary information are subject to the users review to pinpoint the root cause of spinning beachball on macOS.

6. Case Studies

In this section, we demonstrate how Argus helps to diagnose the spinning beachballs in 11 popular applications. We begin with how we found out the design of spinning beachball, signifying an app is non-responsive in macOS, as it is the situation we start diaganosis. Then we describe the cases in three categories corresponding to three bug categories **B1 - B3** in § 2. Since chromium case has discussed there, it will not be present here.

When the spinning beachball shows up, a hang reporting tool, spindump usually kicks in automatically to sample for debugging. To figure out the spinning node in the main thread, we turn to the event graph, and slice path backward from the launch of spindump.

The path shows the spindump is launched after receiving a message from WindowServer, which received a message from the NSEvent thread of the freezing app. The Back_trace events attached to the messages further reveal NSEvent thread per process fetches CoreGraphics events from WindowServer, converts and creates NSApp_event for the main 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 wait cursor over the application window. Moreover, while exploiting how NSEvent thread communicates with the main thread, we found two shared variables, "`is_mainthread_spinning`" and "`dispatch_to_mainthread`" are used to identify the main thread status. As a result, Argus can make use of either the API or the shared variables to identify the spinning node in the main thread.

6.1. Blocking Cases

In this section, we discuss the cases where the spinning node is blocking on Wait event or yielding loop, correponding to **B2** and **B3**.

SystemPreferences System Preferences provides a central location in macOS to customize system settings, inclduing the configuration of additional monitors. A tool called DisableMonitor [6], distributed via GitHub completes its function with enable/disable monitors online. We blocked on the spinning beachball when we disabled an external monitor, and rearrange the windows in the "Display" panel afterward.

We collected data with the Argus, which contains 1) with external monitor on and rearrange the windows, and 2) disable the external monitor and rearrange the windows. The spinning node in the main UI thread is dominated by *mach_msg* and *thread_switch*, which falls into the category of yielding loop. We finds two missing shared variables via the reverse engineering and "`_gCGWillReconfigureSeen`" and "`_gCGDidReconfigureSeen`", which are userd to signify the display configuration status.

Argus compares the two cases and reveals in both cases the first variable is set to indicate the begin of display configuration. However, in case 1) the main thread of System Preferences receives a datagram which set the variable "`_gCGDidReconfigureSeen`" in *display_notify_proc* and finish display reconfiguration, while in case 2) the main thread yields and sends message for the available of such datagram ping until timed out.

Our FindSimilarNode reports a lot of similar node due to the limit of semantics in the node. As a result, we instruct Argus to extends the comparison with its proceeding nodes in the main thread. In conclusion, the bug is from the design in CoreGraphics library and would have to fixed by Apple. We verified the bug by a dynamic binary fixing with lldb script.

SequelPro Sequel Pro 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 noticed spinning beachballs while the network connection was lost and Sequel Pro tried re-connections.

We collected tracing data in two cases: 1) network connected quickly when we login, and 2) Sequel Pro lost connection for a while. Argus first identifies the spinning node from SequelPro's main thread. With the Algorithm FindSimilarNode, Argus gets a baseline node(N_1) in case 1). From there, Argus goes along the weak edge, from wait event to wake_up event, to figure out the node missing in case 2), and slices path backwards from the node(N_1). Argus carries out the comparison of the nodes in the thread before the normal node gets waken and before the spinning beachball shows up. The close examination tells that the main thread is waiting for the kernel thread, which in turn waits for the ssh thread.

Argus helps to filter out overconnetions by dividing XXX combined batching node with hueristics, and recovers XXX missing edges due to the shared flag. In the diagnosis process, we get XXX similar node, XXX times of queries on backward slicing , XXX nodes for inspect.

Installer 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 beachball. Examing the spinning node reported by Argus, it is easy to figure out the main thread blocks in *[IFRunnerProxy requestKeyForRights:askUser:]*. The function sends a synchronous message to daemon *authd*. The root cause is the authentication of user's previledge synchronously in the main thread, instead of the handler for moving cursor.

6.2. Busy Cases

In this section, we discuss the cases where the spinning node is busy on CPU computing, corresponding to the bug category **B1**. 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.

TeXStudio TeXstudio [?] is an integrated writing environment for creating LaTeX documents from github. We noticed a user reported bug when he modified his bib file, the TexStudio hanged for mimutes. The issue was closed by author for incomplete information to reproduce. We reproduced the bug with a large bib file editing and collected the data. With the file opened in a tab, every time we touched the bib file through other editor, vim for example, it shows a spinning beachball on the window.

Argus figures out the spinning node is busy processing. Slicing the path from the busy node, Argus reaches the daemon "`fseventd`" and figures out the busy processing invoked by the callback function from the daemon. The advantage of Argus over other debugging tools is it helps to narrow down the root cause with the path. If the user reported the infomation with Argus, it should have provided the author more hints to reproduce the bug successfully.

Rest Cases
- Notes
- TextEdit
- Microsoft Words
- Sublime Text
- TextMate
- CotEditor

7. Performance Evaluation

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

As we use the ring buffer to collect events, the space cost for disk is fixed to the configured size 2G by default, and the memory used to collect tracing items are fixed to 512M by Apple internal, so the memory overhead is pretty low with regards to the memory usage of modern applications. We measure the the CPU overhead and throughput of Argus by running benchmarks in our machine with the tracing disabled and enabled respectively, which we name them bare_run and tracing_run.

	1st	2nd	3rd	4th	5th
bare run	5.98	6.23	6.18	6.05	6.28
tracing run	6.29	6.01	6.09	6.28	6.01

Table 1: Score From iBench

iBench We first show the five runs of iBench for both cases. The machine is clean booted for each run, and the higher score means it perfoms better. As shown in Table1, their performance are almost of no difference.

TPC-C Next, we evaluation the throughput with tpc-c on Argus with local mysql service.

Chromium Benchmark We also perform the evaluation on benchmark of Chromium, while recording the time usage. The result is shown in Table 2.

	bare run	tracing run
real	27.7s	28.0s
user	28.3s	28.3s
sys	5.0s	5.7s

Table 2: Chromium benchmark

Conclusion As shown in Table 2, the overhead for real, user and sys are 1%, 0% and 14% 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. Magpie [9] is perhaps the closest to our work. It 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. In contrast, Argus's goal is to identify the root causes of performance issues. Its graphs are not request graphs, but rather graphs that may contain many requests. In addition, it logs normal and abnormal executions in the same event trace. In addition, 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.

Panappticon [26] 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 [19] 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.

XTrace, Pinpoint and etc [14, 10, 11] 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 does not assume the presence of a unified identifier in closed-source, third-party applications, frameworks, and libraries.

Aguilela [7] uses timing analysis to correlate messages to recover their input-output relations while treating the application as a black box.

Performance anomaly detection. Several systems detect performance anomalies automatically. [15, 25] leverage the user logs and call stacks to identify the performance anomaly. [12, 21, 23, 13] apply the machine learning method to identify the unusual event sequence as an anomaly. [24] 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, and incurred only 1% CPU overhead 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] The Chromium Projects. https://www.chromium.org, 2008.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.