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

Abstract

Confusing behaviors are everywhere in GUI Applications, which usually make the user feel annoying. Spinning beachball is one of them in macOS. However, even unveiling the mystery of its design is not easy due to the closed source. Consequently, it is difficult for users to begin the debugging task in the wild. A tool to understand the complex behavior of GUI Apps in MacOS is essential, especially for a user who is neither a system expert nor the developer of the applications, to aid debugging, compile concise bug report, or even come up a temporary binary patch.

Argus is the framework where system-wide activity is recorded until the mystery occurs in macOS, and analysis tools are built to explain the behavior. It monitors the system with low-overhead instrumentation running 24X7, from kernel and libraries, and constructs the relationship graph with the schema defined in the framework. The schema identifies execution boundaries in threads and the causality among them. The graph usually contains not only the problematic case but also the normal execution. Upon the graph, analysis tool compares them to manifest the root cause of the anomaly. Results are represented with the suspicous control flows in anomaly case. Hence users can precisely point out the path leading to the performance anomaly. In this paper, we describe and study the usage of our Argus to reveal root causes for the impenetrable behaviors from the real world applications in macOS.

ACM Reference Format:
. 2019. **Argus: Debugging Performance Issues in Modern Applications with Interactive Causal Tracing**. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 sometimes machines instead of in one sequential execution segment. 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. More often than not, developers give up and resort to guessing the root causes, producing "fixes" that sometimes make the matter worse. For instance, a bug in the Chrome browser causes a spinning cursor in MacOS when a user switches the input method to XX [**?**]. It was first reported in 2000 XX time, and developers attempted to add timers to work around the issue. Unfortunately, the bug remained and the timers added made diagnosis even harder. The bug remains open for XXX years.

Prior work proposed *Causal tracing*, a powerful technique to construct request graphs (semi-)automatically [**?**]. It does so by inferring (1) the beginning and ending boundaries of the execution segments (vertexes 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 []. They are shown to be quite effective aiding developers to understand complex application behaviors and debug real-world performance issues in XXX.

Unfortunately, based on our own study and experience of building a causal tracing system for commercial operating system MacOS, we found that modern applications frequently violate these assumptions, rendering the request graphs computed by causal tracing imprecise 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 work on behalf of multiple requests together without no clear boundaries between them. For instance, Windows server in MacOS sends a reply for a previous request and receives a message for the current request using one system call SendRecvXXX presumably to reduce user-kernel crossings.

Second, the graphs may be missing numerous causal edges. For instance, consider ad hoc synchronization [**?**] via shared-memory flags: a thread may set "`flag = 1`" and wake up another thread waiting on "`while(!flag);`"

to do additional work. 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 shared-memory operations, incurring prohibitive overhead and adding many superfluous edges to the request graph.

Third, 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* imprecise and both over-approximate and under-approximate the reality. Although developer annotations can help improve precision [], modern applications use more and more third-party libraries whose source code is not available. In the case of 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 shared memory flags in modern applications, it is hopeless to count on manual annotations to ensure precise capture of request graphs.

We present Argus, a practical system for effectively debugging performance issues in modern desktop applications despite the imprecision of causal tracing. We designed Argus to be interactive, as a debugger should be, so that its users can easily inspect current diagnostics and guide the next steps of debugging to counter the inherent imprecision of causal tracing. For instance, Argus's request graph contains likely missing and superfluous edges for users to view and confirm during debugging.

Moreover, Argus enables users to dynamically control the granularity of tracing using a couple of intuitive primitives. It starts off with always-on, lightweight, system-wide tracing. When a user observes a performance issue (e.g. , a spinning cursor), she can inspect the current graph Argus computes, configure Argus to perform fine-grained tracing (e.g. , logging call stacks and instruction streams) for events she deems relevant, and constructs a more detailed graph for diagnosis. Argus also supports comparison of request graphs of normal vs buggy executions because the difference of the similarly imprecise graphs often reveal precise root causes.

We implemented Argus in MacOS, a widely used commercial operating system. MacOS is closed-source, so are its common frameworks and many applications. It therefore provide a true test of Argus. We addresses numerous nuances of MacOS that complicates causal tracing, and built a system-wide, low-overhead tracer. Our evaluation using XXX real-world spinning cursors shows that Argus is fast (XX% overhead) and effective (XX).

This paper makes the following contributions: our conceptual realization that causal tracing is inherently imprecise 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 our results diagnosing real-world spinning cursors and finding root causes for performance issues the remained open for XXX years.

This paper is organized as follows.

2 Approach Overview

The Argus prototype is designed to expose the thread relationships with temporal constraints as an graph and allow analysis toolkit built on for diagnosis. Workloads running on MacOS usually send requests to various sets of daemons. One request can take paths through multiple daemons, while one daemon can also accept a batch of comming messages and demultiplex them in a daemon thread. For example, the window server communicates with every application to pass the user input events and draw things on the display. Our framework records the system-wide activities with minimal instrumentation and constructs the thread relationship with our defined event schema built in the framework.

2.1 Instrumentation

The instrumentation in Argus leverages the tracing technology from Apple's event logging infrastructure. An event consists of a timestamp, an event type name to notate current activity and arbitrary attributes. One example is as the following line.

timestamp, Mach_msg_send, local_port, remote_port, ...

The API that produces a stream of timestamped events of an event type is called a tracing point. The goal of our Instrumentation is to attach the tracing points in significant spots where the thread state can be captured. The code where the asynchronous task is submitted is one of the significant spots. In the situation, the asynchronous handler should be captured from the local stack or registers in the thread.

There are three main categories of tracing points:

1. Tracing points implies the relationship across thread boundaries.
2. Tracing points identifies the requests boundaries inside a thread.
3. Tracing points improves the comprehension of the system activity, e.g. the call stacks.

Most of them are from the kernel which is open source and pre-existing in the current version of MacOS. We augment them with more attributes to support our extensive use. Libraries and frameworks providing batch processing programming paradigms are instrumented to detangle multiple requests. We can instrument anywhere if necessary, but rarely touch the user space to allow our toolkits more general.

2.2 Graphs Construction

In the process of graphs construction, we extract the built-in event schema that indicates how a subset of events connected, and if an event acts as a delimiter, which implies a thread reaches the beginning or end of a request. We link the threads with connections defined in the subset of events and split the threads with the implied delimiters. A thread is split into multiple execution segments, decomplexing the execution of different requests on the thread. Therefore, multiple graphs are generated to represent the system activities, with the execution segments mapped to the nodes and the links among them to the edges.

However, the graphs constructed with event schema is neither accurate nor complete definitely. The missing connections may be caused by shared variables which are widely used for the synchronizations among threads but is too exhaustive to explore. On the other hand, the delimiters are not explicit somewhere due to undocumented programming paradigms. For example, a batch processing programming paradigm is not uncommon in current systems from user-defined server thread to the kernel thread. Furthermore, the relationship between a thread waits, and another thread wakes it up later, are not equal to the dependency between them. To improve the constructed graphs, we come up with an ad-hoc hardware watchpoint tool to monitor shared variables if necessary to improve the completeness, and heuristics to ensure each node is only on behalf of one request. We present the details in the following sections.

2.3 Analysis Toolkit

With the graphs generated, analysis tools can be built on for various purposes, studying requests, comparison multiple executions or retrieve the bug paths. Some design mechanism can be revealed with the graphs. Our roughly generated graph on a toy app used particular API tells how the API gets implemented and what daemons are accessed to complete it. For example, the NSLog is implemented by sending messages to the server. The design of the spinning cursor in MacOS is tapped by checking the path that triggers spindump. Timestamps carried by the tracing events helps to calculate the time cost of the execution segments, as well as the blocking time of a thread on specific resources. By checking the long execution segments or the long time blocking in UI thread, users can confine the performance anomaly to the execution interval.

3 Built-in Event Schema

The instrumentation framework must support the capture of performance anomalies now and then without adding too much time overhead or exhausting the system storage. The built-in event schema alleviates the pain of the users without expert knowledge across the whole system suffers from constructing an event schema to capture the Application and system states. To balance the limited information amount capacity and its usefulness, we only record events with the purposes of threads temporal link, requests splits and minimal comprehensible information. We make extensive use of the Tracing points added by Apple and augment them with necessary attributes in the kernel. Libraries and Frameworks are instrumented only when it is necessary.

3.1 Link

- Inter-process Communications: mach_msg is the low-level IPC mechanism widely adopted by libraries and user applications. The message sent and received will be matched to produce the tie between the sender and receiver threads, as well as the flow of the reply message. The connection will be based on the port names in userspace and their kernel representation we collected with unique address slides.
- Thread-scheduling: Contentions for virtual resources usually result in thread block in a wait queue, and get unblocked by the other thread when the resource is available. For the patterns like producer and consumer, we will connect the producer and consumer based on the resource id in the kernel.
- Timer-arm-and-fire: Timers are widely used in the Cocoa Framework. We add tracing points in the kernel to capture the where it is armed and where it is fired or canceled. The timer mechanism is implemented by encapsulating the callout function in a kernel object and linking/removing the object to/from the timer lists. Therefore, we record the slid address of the object and the user passed in function address as attributes in the kernel, which is used to connect events of timer armed, timer fired and timer cancellation.
- Dispatch Queue synchronization: Dispatch queue is one of the main synchronization mechanism. As the implication of the name, tasks can be added into the queue and later get dequeued and executed by another thread. We add tracing points in the

binary code spots where enqueue, dequeue and block execution are performed in the library.

- Runloop synchronization: A RunLoop is essentially an event-processing loop running on a single thread to monitor and call out to callbacks of the objects: sources, timers, and observers. We leverage the binary instrument in the Cocoa Framework, where the callouts get submitted and performed. Runloop object reference and input source signal are recorded for sources; Runloop reference and callout block address are recorded for observers; As timers are implemented with the system timer-arm-and-fire, no instrumentation is needed in the binary framework.
- CoreAnimation-set-and-display synchronization: The display of Applications are implemented in Cocoa Frameworks in a batch processing manner. The display bits will be set when necessary, and later the CoreAnimiation Layers get drawn batch at the end of an iteration in RunLoop. The address of CA Layer object and display bits are recorded for the connection with binary instrumentation in the Cocoa Framework.
- Shared Variable: Last but not less important, variables are an important channel for thread synchronization. The synchronizations above are less or more making use of the shared variables in objects. It is hard to explore, for a reason enclosed in a structure.

3.2 Split

Continuously processing unrelated tasks is not uncommon in threads, from userspace, system services, and kernel threads. Tracing points are required to split the events in a thread with the request boundary to exclude the false linkage in the graph. With the boundary, the interleaving execution of requests on the same thread can be decomplexed, which benefit the following analysis when a comparison is required. Three main categories are covered in our built-in schema.

- System interference: User threads are randomly interrupted by the system activities, for example, interrupts, timeshare maintenance. We recognize the boundary of them and isolate them from activities triggered by a user application.
- Batch processing: The second category is the batch processing in Daemons and Application Services. Boundaries in this category are usually implied by certain programming paradigms. A checking tool is created on our framework to unveil these programming paradigms.
  In our builtin schema, we cover not only the traditional programming model, dispatch queue and run loops, but also recognize special ones. For example, WindowServer sends out pending messages with the receiving of an unrelated message from kernel via one system call, due to Apple's design of the port set.
  Kernel thread is a important one who processes requests without boundaries. While processing the timer, the kernel thread sends out messages one by one to a list of user processes who armed timer before.
- Heuristics: It is not always possible to manifest all programming paradigms before tracing. In our framework, more can be added with heuristics by iteratively checking on new data.

3.3 Comprehension

To make the output of our data more comprehensible, we add tracing points on system call and insert a lightweight call stack on demand.

- system calls: we record the system call number and corresponding parameters. It helps to understand what system service is required by the execution.
- Lightweight call stacks: the user stacks are unwinded with the valid rbp in the system without processing the complex jump instructions. It benefits the understanding of our data with rare overhead.

4 Implementation

We now discuss how we collect tracing events of interest.

4.1 Tracing Tool

Current MacOS systems support a system-wide tracing infrastructure built by Apple. [traces what] 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 without filling up the disk by implementing a ring buffer backed by a file. We store at most 2GB of data [per log?], which corresponds to approximately XXX events (XXX time).

4.2 Instumentation

The libraries, as well as lots of the daily used Apps on MacOS, are closed source. Adding tracing points to libraries and Frameworks requires the instrumentation of the binary image. Techniques, such as library preload with trampolines over the targeted functions, do not meet the requirements due to the two-level namespace executables in Mac. We hence implement the binary instrumentation lib for developers to add tracing points anywhere.

Input: $VictimFuncs_1 \ldots VictimFuncs_N$
$VictimInstOffsets_1 \ldots VictimInstOffsets_N$
$ShellFuncPtrs_1 \ldots ShellFuncPtrs_N$

1: **procedure** INIT
2: **for** $i \leftarrow 1$ to N **do**
3: $InstrVaddr \leftarrow virtual\ addr\ of\ VictimFuncs_i + VictimInstOffsets_i$
4: $Offset \leftarrow InstrVaddr + 5 - ShellFuncPtrs_i$
5: $CallqInstr \leftarrow callq\ Offset$
6: $mprotect(page\ of\ InstrVaddr, R|W|E)$
7: $memcpy(InstrVaddr, CallqInstr)$
8: $mprotect(page\ of\ InstrVaddr, R|E)$

Figure 1. Pseudocode algorithm.

To complete the instrumentation, we need to find the instruction in the event execution path, where the desired attributes are accessible, and the length of the instruction can hold a call instruction. We call such instruction as a victim instruction. It will be substituted with the callq instruction on the fly. The parameter of the callq instruction is a shell function, which we define to simulate the victim instruction and add tracing points of interest. With the input of the address of victim instruction and its corresponding shell function, our tool will generate a new library. In the new library, we define an init function, as shown in Figure **??**, to calculate offsets and replace the victim instruction with the shell function. The replacement only happened once in the memory by calling the init externally with dispatch_once.

Finally, the generated library will re-export all the symbols from the original library and replace the original one. Although one of the shortcomings is that only the short distance function call is supported in our implementation, as the new library and the original one are usually loaded close in memory, we do not need to apply techniques to achieve the long jump in instrumentation.

4.3 Hardware Watchpoint

For the shared variable of interest, we take advantage of hardware watchpoints. Tracing points are added in its handler when the variable is accessed. We hook the handler in the CoreFoundation to make sure that it is loaded correctly in the space of our interested application. Setting the hardware watchpoint is ad-hoc with a command line tool we built. Only the process id, the name, and size of the variable, operation type, read, write, execute, the watchpoint register id(from 0 to 3) need to be specified.

4.4 Incremental Instrumentation

Due to the transparent of programming paradigms designed by the developer, it does not always generate a complete and accurate dependency graph based on the builtin event schema. The graph should be incrementally improved with new tracing points somehow.

We built the toolkit upon the currently generated graph to guide the exploration of the over-connections and miss-connections of thread temporally. Vouchers, which are propagated through the system to record if a process works on behalf of the other process, are also taken advantage of to check the false connections. Threads connecting to multiple process are allowed only if their are vouchers reflecting their relationships. Last but not least, checking the path of the connection, as well as the call stack provides hints on the graph improvements.

In our experiment, we discovered multiple programming paradigms in the libraries that are not realized from the initial composing of the schema. WindowServer will compact the request of send and receive in a mach_msg_overwrite_trap system call, although they are on behalf of different userspace applications. The function dispatch_mig_service, as well as the calling out functions in runloop, will finish works one by one in a loop without blocking between them. This procedure to discover such programming paradigm can be repeated on regular executions before tracing for diagnosis. On the contrary, the missing connections are much harder to explore. As long as the remaining connections in the current graph help diagnosis, it is not necessary to explore.

5 Analysis Methodology

To fit the nodes into limited categories are widely utilized in previous work to represent the dependency graphs and critical paths[**? ? ?**]. However, compared to the work with user input schema and limited thread models for mobile apps, the purpose of the event sequence in Argus is not straightforward to generalize to a high-level semantics. Daemons and services make MacOS more like a distributed system with overwhelming IPCs. Handlers in these background threads are versatile and not obvious to identify. Besides, it is not essential to recognize the semantics for every execution segment, as long as all the events included in the same execution segment are on behalf of the same request, and the integrity of a request can be preserved with connections.

5.1 Generate the relationship graph

We first reveal the causalities among threads by matching the patterns of tracing events with the built-in schema. For example, the RPC implemented with mach_msg usually have more than two threads involved in MacOS. The thread that sends out the request is not the same thread that receives the reply message. As a result, both

the ports and the process information are required to ensure the correct connections.

Secondly, execution segments are produced by identifying the boundaried of different requests with the built-in schema. In addition to the boundaries for asynchronous tasks callout, we make use of the connected peers as a heuristic to infer the boundaries in the background threads. The heuristic is based on the truth that the request from an application processing in the daemon rarely has causality to another user application. Only the daemons may constradict the heuristics. Therfore, we correct the heuristic with the vouchers from Apple, which indicates how one process send message to a second process on behalf of a third process. Also, the traditional assumption that an asynchronous call is for one task is not true in MacOS. As a result, we keep all events happen inside the callout of the asynchronous task from the dispatch queue in one execution segment unless dispath_mig_service is invoked. In the case, we isolate every service to an individual node.

Finally, we generate a directed graph with the execution segments mapped into nodes, and the causalities mapped into edges.

Since the edges between nodes are of various types and not unique for each node, the graph is not acyclic. For example, node A of the execution of callout from the dispatch queue has a wake-up edge to node B and node B has a message sent to node A later. The numbers of nodes and edges for a real-world application are tremendous. As a result, we do not throw the whole graph to users. Instead, we build a search tool to assist the user in digging into the suspicious part of the graph only. Users can also define their algorithm for different usage leveraging the rich information captured in the graph. We now describe the typical use cases on our framework.

5.2 Extract control flow

Although overlapping user transactions are hard to separate in that hidden batch processing exists everywhere in background threads, display updates are always batched, and some event handlers require multiple user input events to trigger, the backward path slicing is helpful to extract a portion of control flow from a targeted node. We can check the incoming edges in the corresponding node to add the precedent node into the path. Although one node can have multiple incoming edges, it is likely that the sources of the edges are the same.

For example, if a thread sends a request to dispatch queue and picks a worker thread to process, it will be likely to wake up the latter thread too. Therefore, edges of thread scheduling and asynchronous task causality appear between the same pair of nodes. In addition to choose the precedent node that connects repeatedly, we can also make use of the human pattern recognition ability to choose the correct precedent node by allowing interference in this process,

Besides, the backward path helps to exclude false connections. The node containing two requests will finally encounter branches in the backward path. We can either check the node or only pick the more reasonable branch. As we reserve the causality source information, it is not hard to discriminate them and make a decision. We show the example in Figure **??**.

5.3 Indentify root cause of anomaly

It is not limited to the spinning cursor showed up in the GUI application in MacOS. As long as an indicator of anomaly is found, the analysis method can be applied.

For the cases of main thread non-responsive, three main steps are taken. It first identifies the node in the main thread corresponding to the anomaly in the graph. As we figured out the design of the spinning cursor in Section 6.1 with the control flow extracted, the timer that triggers spinning cursor in NSEvent thread can be identified along the path. With the timing information, it is not hard to get the anomaly node in the main thread.

It is quite common that the main thread is busy process or always waiting(with or without timeout) base on our studies.

The second step is to get a control path for the normal case. Correctly checking the similarity of a node is the key in this step. Normalization of nodes is applied in the step to exclude the noise from the minor difference of events. Only a subset of events and a subset of attributes are included in the normalized node. By comparison, we figure out a corresponding normal node which has the same thread attribute as the anomaly one, while has difference on specific attributes. The attributs usually includes the return value of the system call, the wait intervals of wait event inside, and the time cost of the node. These nodes are quite possible to reflect the expectation of normal executions at the point.

For the anomaly node waits for a long while, we will choose the node that wakes up the corresponding normal node to begin the backward slicing. As a result, we get the path that represents the normal execution. On the other hand, anomaly nodes that have lengthy processing are likely to reveal the root causes by themselves.

The third step aims to discover the root causes of the difference between the normal and anomaly paths. The normal node is from the path sliced in the last step, while the anomaly node is from the same thread as the normal node. It is noted that the anomaly node must happen after the normal node. The comparison algorithm is as shown in the figure **??**

5.4 Validation of bug fix

With the root cause revealed, users can come up with a binary patch with our instrumentation tool by themselves if necessary. Argus also provides a way to check how the fix works either by comparing to the anomaly extract in section 5.3. Alternatively, the user can add the tracing points in the binary patch for verification. For the case that has a timeout, we can narrow the timeout to check if the time interval when the main thread is non-responsive get reduced.

6 Case Studies

We apply our tools to study the design of spinning cursors in MacOS and analyze real-world bugs. We illustrate two of the bugs in this section and leave more cases in section **??**. Both of them are longlisted in public.

The chromium bug has been reported multiple times in the chromium bug report. It was initially a deadlock bug in which the main thread in the browser is waiting on a condition variable. As no one knows how the deadlock triggered, timeout is added as a programming trick to eliminate the deadlock. However, the bug is not resolved, and the main thread would always timeout in some situation for a relatively long time. Hence, a further patch introduced cache mechanism to alleviate the non-responsive, which did not solve the problem from the root cause and the main thread still get hang from time to time.

The bug of the System Preferences is exposed by an app called DisableMonitor. The app can change some system settings and eventually reveal the inappropriate displayer management in CoreGraphics Framework.

6.1 Spinning Cursor

The spinning cursor is a painful sight for Mac users, signifying that the application is non-responsive. It usually remains for munites, leaving the users at a loss . To understanding the design of the beachball can make the user more confident to take the next step, based on his expectation of the GUI apps. From the console in MacOS, we realize that every time the spinning wait cursor appears, the spindump would be triggered too. We first identify the execution segment where the spindump launched with the tool built on the graph. Then the tool does a path slicing backward to find out the control flows. By examing the path with the call stacks, we can figure out the processes involved inside the design. Every GUI app on MacOS contains the main UI thread and event thread. The event thread communicates with the WindowServer to fetch the events for the application, and place it in a queue for the main thread to process. When the queue does not proceed in a certain amount of time, with the timer set, the event thread would check the application state and send a message to WindowServer. Finally, the WindowServer notifies the CoreGraphics to draw spinning cursor over the application window. While the spinning cursor is displayed, the main thread is still working and strive to eliminate the cursor once the current event processing is done, unless a deadlock happens there. With the understanding of the design, not only we can capture user input sequences that trigger the high processing to reproduce it for diagnosis, but also we can make use of it to identify the hanging execution in the graph for further design of analysis toolkits.

6.2 Chromium IME responsive

One of the long-lasting performance issues in chromium is hanging caused by the non-English input. When users try to type non English to textFields, such as search box, the main thread of the browser becomes not responsive. With lldb, it is not hard to tell that the main thread gets stuck on FindFirstRect, where the main thread waits for the signal of the condition variable. According to the history in the bug report, the developers realized there were deadlocks somewhere. However, it was hard to pinpoint due to the multiprocess and multithread programming paradigms. As a result, a timeout was added to prevent the deadlock, but not the long latency. Although a further bug patch introducing cache helps to eliminate the long hanging mostly, the performance issue still appears from time to time. The scenario we can reproduce is to launch the website of Yahoo and immediately quickly type Simplified Chinese.

The ground truth we revealed with our tool is as shown in picture XXX. In chromium, there are one browser process and multiple renderer processes. The main thread of the browser process tries to get the caret position. It sends out the message and anticipates for the reply message with a condition variable. Usually, a worker thread in the browser process will return the firstrect and wake up the main thread. However, it requires the message from the main thread of a renderer process to proceed. Without the message from the renderer process, the worker thread is not able to signal the main thread. Thus, the main thread will always time out.

Our trace tool will collect the data system-wide. Therefore, all the thread relationships are captured. With the trace log size, both the hanging case and non-hanging case are recorded. From the shared condition variable between threads, we are able to align the logs of the two cases and discover the missing message in the hanging case.

As we have known the unresponsive of the main thread in the renderer process, we further consult the analyzed trace log and observe that it is waiting on a semaphore, and eventually waken up by the main thread of the browser process.

Our tool further explains the root cause of the livelock with conditional debugging. We can either apply the binary instrument or modify the source code to make the renderer thread accept the attachment of lldb. The concrete call stacks from lldb disclose the task processing in the renderer thread is related to running javascript.

6.3 System Preferences spin

System Preferences is the application in MacOS for users to modify various settings. The Preference Pane named Displays allows the user to rearrange the position of displays, the location of the menu bar and set parameters for display, such as the resolution, brightness, and rotation, but to disable a monitor online is not supported in Mac. DisableMonitor is an app to easily disable/enable a monitor. It is implemented by calling to the API from Apple, CGSBeginDisplayConfiguration, CGSConfigureDisplayEnabled, and CGSCompleteDisplayConfiguration. The bug appears when an external monitor is disabled with DisableMonitor, and the user drags the windows under the tab of arrangement in System Preference. It makes the System Preferences window freeze for a few seconds.

One straightforward method for the single process app is to attach lldb, resume the process until the system throws out the spinning cursor. However, in the very case, it only tells when the spinning cursor shows up, the main thread is busy calling thread_switch from function CGSCompleteDisplayConfiguration in the call stack. Further static analysis, for example, reverse engineering on the binary, cast light on the programming paradigms of the repeating. In our case, it reveals the main thread is stuck in a loop repeating thread_switch until a variable is set or time out. Nevertheless, it is still hard to answer the question of why and how the variable gets set or if it is cleared by mistake.

Only the comparison of the buggy case and the normal execution can exhibit what is not expected. One advantage of our tool is it is lightweight and does not impact the responsiveness of the thread, we can identify the buggy case and normal case, compared to using lldb to trace through the whole API execution step by step. Another advantage is that it records the activities system-wide, and the data covers from the beginning of the user input, which makes the case more explainable. By adding the hardware breakpoints with our instrumentation tool, we also record the whole history of the variable activity. The tracing on the variable helps to make alignment of the graphs produced in the normal execution and freeze case, which makes the further comparison feasible. It reveals datagram from the WindowServer will post notifications for the application. The particular datagram makes the application to finish reconfiguration and set the variable, while in the spinning case the reconfiguration gets initiated but not completed. The handler display_notify_proc is not appropriately implemented to make the application exit the loop when there is an exception in the reconfiguration.

With the findings of our tool, as is shown in Figure **??**, we can now explain the root cause with the pseudocode in Figure **??** and Figure **??**, which provides insight for the developer to fix the bug.

7 Evaluation

We first presents the results from the live deployment of Argus. We list the softwares which trigger spinning wait cursors in MacOs and the root cause we figure out with our framework. Then, we show how much work our tools take over from the user and make the diagnosis much easier in the wild.

- tracing overhead
- number of edges and nodes in the graphs.
- portions of data the user need to examine to figure out the root cause.

8 Related Work

While there is no system target for providing tools for users to assist the debugging of closed source applications on MacOS, several active research topics are closely related.

textbfCorrelate tracing events: Aguilela [**?**] use timing analysis to correlate messages and treat the system as a black box. Argus encompass the system knowledge to correlate messages, asynchronous operations, and share variables to achieve a finer grained analysis. Magpie [**?**] is the most closely one which is a system monitoring and modeling server workload on Windows system. The goal of Magpie is to model a workload with normal behavior and with the data set to detect the anomaly one with statistic method. Our goal is to identify the root cause of anomaly with the relationship graph, which includes not only the normal execution but also the anomaly one. Meanwhile, for the boundary of a request and causality of traced events, Magpie depends on the detailed knowledge of application semantics from the developer, while Argus defines them in the framework, and the semantics of the application is not required.

Panappticon [**?**] monitors the mobile system and use the trace to characterize the user transactions of mobile apps. Although it aims to track system-wide and correlate events without developer input, the design is base on two thread models in its assumption.

AppInsight [**?**] instruments application to identify the critial execution path in user transaction. It leverages the semantics and opcode in high-level frameworks, which is not available in MacOs. It does not track the request

across process/app boundary. Thus it cannot reveal the root cause due to other processes.

XTrace and Pinpoint [**? ?**] both 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 use identifier due to the maintained overhead and propagation of such identifiers through the developing environments is not feasible with the closed source applications, frameworks, and libraries.

Performance anomaly:[] leverage the user logs and call stacks to identify the performance anomaly. From our daily sense, the anomaly bugs whose root cause is hard to reveal are from the inefficient codes that escape the developers' bullets.

[] apply the machine learning method to identify the unusual event sequence as an anomaly. [] generates the wait and waken graph from sampled call stacks to stidy a case of performance anomaly.