Argus: Understanding App Behavior Through Full-System Tracing

SOSP 2019 submission #XX

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.

1 Introduction

Tools to understand behaviors of GUI Applications from system-wide tracing are essential for performance diagnosis. GUI Applications are widely installed in desktops. There usually exists complicated synchronizations to support the sophisticated graphical design, with the powerful libraries. Hundreds of daemons and services are running in the background in MacOS, for instance. A simple API can access multiple daemons, some of which the developers may be unaware of [**?**]. Performance anomaly, under the circumstance, happens with specific environment settings. Logs either from the system or the software are often insufficient to reproduce, let alone figure out its root cause. To address the problem, we build Argus to understand the application behavior and causality with the system-wide tracing.

Statistic methods are widely proposed in the past years[] to identify root causes for performance anomaly. Insights on the possible causes are presented in those work, which are too vague for the users to carry out debugging task. Event-based tracing is another preferred technique in debugging distributed systems. Previous works [**? ?**] try to model workloads in distributed systems to detect performance anomaly with the hints from the developer. [**?**] relies on the event schema from developers to identify the boundary and dependency of the traced events, while [**?**] requires the correct specification of the workload from developers, and compare them with the real execution in the system. Mobile apps are much more straightforward. [**?**] leverage the higher-level frameworks to overwritten the bytecode, which preserves the structure of the program, including types, methods, and inheritance information. With the instrumentation of the handlers, it captures the end to end control paths for user input transaction. [**?**] instrument limited programming paradigms base on the assumption of two thread models in Android. The message queue thread and pool thread model are not general enough to cover all threads in the system though, even for the less complicated mobile apps. Correlating the user input and the display is still an open research problem.

In our experience, the threads in MacOS are more complicated due to the widely used mach_msg and batch processing, from the RunLoop sources, dispach_mig_service, to timers, kqueues in the kernel. The programming paradigm like the nested dispatch queue structure challenges the traditional assumption that the activities in a task from the task queue are always on behalf of the same request. Moreover, previous work overlooks the false causalities introduced by a daemon, kernel thread and programming tricks like timeouts.

Argus is unique in the following contributions. First, we come up built-in event schema for synchronizations between threads from kernel and libraries without user input and support the incremental instrumentation based on the tool built to explore the completeness of the schema. Second, a root cause searcher is constructed with path slicing and comparison algorithm. With the toolkit on Argus framework, users can assist debugging without the source code or system knowledge, compile a concrete and helpful bug report, and event make a binary patch for the applications.

The remainder of 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

Tracing infrastructure builtin current MacOS is lightweight event logging technology which collects system-wide information, stores temporarily in memory and flushes to screen or disk when the buffer is full. To support the 24X7 tracing without exhaust the system resource, we modify it to dump data to a ring buffer in a file.

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.

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.

Figure 1 pseudo code for init

```
Input: $VictimFuncs_1 \dots VictimFuncs_N$
    $VictimInstOffsets_1 \dots VictimInstOffsets_N$
    $ShellFuncPtrs_1 \dots ShellFuncPtrs_N$
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:         $mprotect(page\ of\ InstrVaddr, R|E)$
```

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 Incrmental Instrumentaion

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

Comparing to the user input schema and limited thread model from mobile apps, the purpose of the event sequence is not easy to generalize without the high-level semantics. Daemons and services make MacOS more like a distributed system with overwhelming IPCs. Event handlers in the background threads are not straightforward to identify. We first reveal the causalities among threads and then make use of the connected peers as a heuristic to infer the boundaries of different requests processed in the background threads. Finally, we generate a directed graph with the execution segments inside the boundary are mapped into nodes, and the causalities are mapped into edges. To fit the nodes into limited categories of operation summaries are widely adopted to improve the comprehension of graphs in the previous work[**?**] However, it is hard for our system, which is similar to distributed systems. Execution segments for daemons are quite versatile. We do not need to recognize the programming paradigm 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 the edges.

Since the edges between nodes are of various types, 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 nodes and edges number for a real-world application is 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.

5.1 Indentify root cause of anomaly

Our search algorithm will first identify the node in the graph corresponding to the anomaly. For the non-responsive of UI thread, we will first search the graph to figure out when the event thread get a time out while waiting for the event queue get dequeued. With the timing information, our search algorithm finds the node. Mostly, the main thread is not get blocked. Instead, it will be waked up after a short timeout, or just in busy processing. To figure out why the anomaly happens, we need to make a comparison.

We first do a similarity searching. The first step is to figure out a similar node that has the same thread attribute before the anomaly happens. To find out the similarity, we need to normalize the node to preserve a subset of tracing events. For each type of events, only certain attributes are used for comparison. We reserve the peer process as an attribute for the event conneted to other threads with causality. System call names are reserved while the arguments and return value are disregard. All the identified similar nodes will push into the queue, and we will choose the ones that act differently from our examined anomaly node. The difference contains the return value of the system call, the wait intervals of wait event inside, and the time cost of the node. These nodes are entirely possible to reflect the expectation of normal executions at the point. The second step of the similarity searching is to find out the causes of the difference in the two nodes. If it is from the wait time, we will get the nodes that wake up the blocking as the initial node for comparison.

After the similarity searching process, the backward path slice from the node is carried out. The path stands for the regular control flow. Each of the nodes is compared to the nodes afterward in the same thread respectively.

The comparison algorithm is as shown in the figure **??**

5.2 Validation of bug fix

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 is initially a deadlock bug. Timeout was added, and it turned to a livelock bug. Cache mechanism is added to alleviate but not solve it ultimately. 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 the Core Graphics.

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

- retults for more cases: TextEdit, Notes, installer, Hopper Dissembler, ...

8 Related Work

References