Argus: Understanding App Behavior Through Full-System Tracing

SOSP 2019 submission #XX

Abstract

Contribution of the paper

- make use of a lightweight trace tool to enable 24X7 recording for the full system, casual buas can also be captured
- discover and integrating various type of events incrementally to provide a builtin event schema for constructing graphs in the system, more comprehensible.
- the root cause revealed is more precise. The causality paths can across the process boundary.

1 Introduction

2 Approach Overview

The XXX prototype is designed to expose the thread relationships with temporal constraints and further 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 tasks and demultiplex them. Our framework records the system-wide activities with minimal instrumentation and constructs the thread relationship with the implicit schema from them.

2.1 Instrumentation

XXX instrumentation 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. 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 system state can be captured. 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.

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 that 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. Consequently, multiple graphs are generated with the execution segment as a node and the links among them as the edge. They represent the system activities.

However, the graphs constructed with event schema is neither accurate nor complete. The missing connections may be caused by shared variables which are widely used for the synchronizations among threads, which is 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. 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 and heuristics to XXXXXXXXXXXXXXXX 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 independent tasks is not uncommon in worker threads, threads running runloop, kernel threads and threads from daemons like WindowServer. Tracing points are required to split the events in a thread with the request boundary to exclude the false linkage in the graph,

- Kernel activities: interrupts, timeshare maintainance, timer processing.

- Daemon activities: message sent and receive in WindowServer
- Programming paradigms: batch processing in dispatch queue and runloops.
- Heuristics:

3.3 Comprehension

- system calls: we record the system call number and corresponing parameters.
- Lightweight callstacks

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.

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

Figure 1 pseudo code for init

```
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:         $mprotect(page\ of\ InstrVaddr, R|E)$
```

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. Two steps are required.

- Add tracing points
- Parse and add event schemas on the link and split process if necessary.

We built the toolkit upon the currently generated graph to guild the exploration of the over-connections and miss- connections of thread temporally. We leverage the tracing points of voucher builtin in MacOS. Vouchers are propagated through the system to record if a process works on behalf of the other process. We check the over-connections in one execution segment by examing if multiple user applications connect to it but not in the vouchers. 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 before. 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 Applicance

Analysis toolkit can be built upon the current graphs.

- bug diagnosis via comparison
- interactive application behavier understanding
- programming paradigm discovery

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

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