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

Abstract

Prior systems used causal tracing, a powerful technique that traces low-level events and builds dependency graphs, to diagnose performance issues. However, they all assume that precise dependencies can be inferred from low-level tracing by either limiting applications to using only a few supported communication patterns or relying on developers to manually provide dependency schemas upfront for all involved components. Unfortunately, based on our own study and experience of building a causal tracing system for MacOS, we found that it is extremely difficult, if not impossible, to build precise dependency graphs. We report patterns such as data dependency, batch processing, and custom communication primitives that introduce imprecision. We present Argus, a practical system for effectively debugging performance issues in modern desktop applications despite the imprecision of causal tracing. Argus lets a user easily inspect current diagnostics and interactively provide more domain knowledge on demand to counter the inherent imprecision of causal tracing. We implemented Argus in MacOS and evaluated it on 11 real-world, open spinning-cursor issues in widely used applications. The root causes of these issues were largely previously unknown. Our results show that Argus effectively helped us locate all root causes of the issues and incurred only 1% CPU overhead in its system-wide tracing.

1. Introduction

Today's web and desktop applications are predominantly parallel or distributed, making performance issues in them extremely difficult to diagnose because the handling of an external request is often spread across many threads, processes, and asynchronous contexts instead of in one sequential execution segment [13]. 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 [6, 24, 20, 15, 22]. More often than not, developers give up and resort to guessing the root cause, producing "fixes" that sometimes make the matter worse. For instance, a bug in the Chrome browser engine causes a spinning cursor in MacOS when a user switches the input method [2]. It was first reported in 2012, and developers attempted to add timeouts to work around the issue. Unfortunately, the bug has remained open for seven years and the timeouts obscured diagnosis further.

Prior work proposed *Causal tracing*, a powerful technique to construct request graphs (semi-)automatically [23]. It does so by inferring (1) the beginning and ending boundaries of the execution segments (vertices in the graph) involved in handling a request; and (2) the causality between the segments (edges)—how a segment causes others to do additional handling of the request. Prior causal tracing systems all assumed certain programming idioms to automate inference. For instance, if a segment sends a message, signals a condition variable, or posts a task to a work queue, it wakes up additional execution segments, and prior systems assume that wake-ups reflect causality. Similarly, they assume that the execution segment from the beginning of a callback invocation to the end is entirely for handling the request that causes the callback to be installed [23, 16]. Causal tracing is quite effective at aiding developers to understand complex application behaviors and debug real-world performance issues.

Unfortunately, based on our own study and experience of building a causal tracing system for the commercial operating system MacOS, we found that modern applications frequently violate these assumptions. Hence, the request graphs computed by causal tracing are 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 together work on behalf of multiple requests with no clear distinguishing boundaries. For instance, WindowServer in MacOS sends a reply for a previous request and receives a message for the current request using one system call mach_msg_overwrite_trap, presumably to reduce user-kernel crossings.

Second, the graphs may be missing numerous causal edges. For instance, consider ad hoc synchronization [19] via shared-memory flags: a thread may set "`flag = 1`" and wake up another thread waiting on "`while(!flag);`" to do additional work. Even within one thread, the code may set a data variable derived from one request and later uses it in another request (e.g., the buffer that hold the reply in the preceding WindowServer example). Although the number of these flags may be small, they often express critical causality, and not tracing them would lead to many missing edges in the request graph. However, without knowing where the flags reside in memory, a tool would have to trace all memory operations, incurring prohibitive overhead and adding many superfluous edges to the request graph.

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

semantics of the code may also enforce a causal order between the two operations.

We believe that, without detailed understanding of application semantics, request graphs computed by causal tracing are *inherently* imprecise and both over- and under-approximate reality. Although developer annotations can help improve precision [5, 17], modern applications use more and more third-party libraries whose source code is not available. In the case of tech-savvy users debugging performance issues such as a spinning (busy) mouse cursor on her own laptop, the application's code is often not available. Given the frequent use of custom synchronizations, work queues, and data flags in modern applications, it is hopeless to count on manual annotations to ensure precise capture of request graphs.

In this work, we present Argus, a practical system for effectively debugging performance issues in modern desktop applications despite the imprecision of causal tracing. Argus's goal is not to construct a per-request causal graph which requires extremely precise causal tracing. Instead, it captures an approximate event graph for a duration of the system execution to aid diagnosis. We designed Argus to be interactive, as a debugger should rightly 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 event graph contains many inaccurate edges that represent false dependencies, which the user can address in an interactive manner. When debugging a performance issue using Argus, a user need only make edges relevant to the issue precise. In other words, she can provide schematic information on demand, as opposed to full manual schema upfront for all involved applications and daemons [5].

Moreover, Argus enables users to dynamically control the granularity of tracing using a number of intuitive primitives. The system begins by using 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 finer-grained tracing (e.g., logging call stacks and instruction streams) for events she deems relevant, and trigger the issue again to construct a more detailed graph for diagnosis. Argus also supports interactive search over the event graphs that contain both normal and buggy executions for diagnosis.

We implemented Argus in MacOS, a widely used commercial operating system. MacOS is closed-source, as are its common frameworks and many of its applications. This environment therefore provides a true test of Argus. We address multiple nuances of MacOS that complicate causal tracing, and built a system-wide, low-overhead tracer.

We evaluated Argus on 11 real-world, open spinning-cursor issues in widely used applications such as the Chromium browser engine and MacOS System Preferences, Installer, and Notes. The root causes of all 11 issues were previously unknown to us and, to a large extent, the public. Our results show that Argus is effective: it helped us find all root causes of issues, including the Chromium issue that remained open for seven years. Argus is also fast: its systems-wide tracing incurs only 1% CPU overhead overall.

This paper makes the following contributions: our conceptual realization that causal tracing is inherently 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 (busy) cursors and finding root causes for performance issues that have remained open for seven years.

This paper is organized as follows. In Section 2, we present an overview of using Argus and a Chromium example. Section **??** describes our approach to identifying semantic dependencies, and Section 4 describes our tracing implementation. In Section 5 we present other case studies, and Section 6 contains our performance evaluation. We summarize related work in Section 7, and end with conclusion in Section 8.

2. Overview

In this section, we describe the steps a user takes to investigate a performance anomaly with Argus. We assume the user has a program to execute and a buggy test case that can be reliably reproduced; if possible, the user can also provide a similar test case which executes normally for comparison purposes. The goal of Argus is to discover a sequence of user-interface (UI) actions that trigger the buggy test case, and a detailed execution trace for these events. In the cases we consider, the flow of information across threads and processes is essential to discovering the system state that leads to a performance bug. Argus recovers UI actions from logged data rather than being told the actions that a user performs, because not all of them may be relevant to the true bug.

Central to our system is our *event graph*, a generalized control-flow graph which includes inter-thread and inter-process dependencies. All diagnoses and inferences are performed within this graph, in a semi-automated fashion: Argus performs searches and subgraph matching to trace logical events as they flow through the system, and the user can interactively query this graph to understand the problem or provide guidance to Argus. The event graph is described in further detail in §3.1. Next, we describe the graph operations Argus performs leading to a diagnosis.

2.1. Graph Exploration and Diagnosis

The exact operations performed on the graph depend on the bug under investigation. Consider a common performance bug on MacOS, the *spinning cursor*, which indicates the current application's main thread has not processed any UI events for over two seconds. Upon examining what the main thread is actually doing, the user may encounter three potential cases.

First, the thread may be busy performing lengthy CPU operations (which take longer than two seconds). In this case, Argus will examine the stack trace and find a sequence of

events that led to the current CPU processing. If more specific tracing is required, the user can rerun the program with more heavyweight instrumentation enabled for any portion of the code's execution, gathering a precise sequence of calls or even instructions executed.

Second, the thread may be in a blocking wait, in which case Argus tries to automatically determine which event would normally wake up the thread. The system does so by diffing the current subgraph with other instances in the recorded execution (assuming the user has carried out a typical baseline test case before triggering the bug). Essentially, Argus traces blocking events and messages backwards through history until relevant UI events are found. The user may manually intervene at different stages to guide the system towards the most relevant portions of execution, leveraging hypotheses or expert knowledge as to why a hang may occur. The full diff algorithm is described in §XXX.

Third, the main thread may be in a yield loop, which is highly indicative it is waiting on a data flag (e.g., "while(!done) thread_switch();"). To discover a data flag, the user re-runs the application with Argus to collect instruction traces of the concurrent event in both the normal and spinning cases and detects where the control flow diverges. She then reruns the application with Argus to collect register values for the basic blocks before the divergence and uncovers the address of the data flag. She then configures Argus to log accesses to the flag during system-wide tracing. Finally, she can recursively apply Argus to further diagnose "the culprit of the culprit".

Based on our results and experience, the first case is the most common, but the second and third represent more severe bugs. Long-running CPU operations tend to be more straightforward to diagnose with existing tools. Blocking or yielding cases involve multiple threads and processes, and are extremely hard to understand and fix even for the application's original developers. Therefore, issues remain unaddressed for years and significantly impact the user experience.

2.2. Argus Algorithm

Main algorithm Next, we describe Argus's operation more precisely. Figure 1 shows the steps a user takes to investigate a performance anomaly with Argus, given a buggy test case and (potentially) a baseline test case for comparison purposes.

First (step 1) the user executes the program with Argus instrumentation enabled, running through the baseline test case (if applicable) and then the buggy test case, capturing execution logs. Next (steps 2-3), Argus computes a generalized control-flow graph which includes inter-thread and inter-process dependencies; this graph is described in further detail in §3.1. In step 4.a and 4.b, the goal is to track events and synchronization primitives throughout the system to figure out the root cause, in terms of user input events, which caused a hang or performance bug. We locate the node N_1 in our graph where the performance bug is triggered. This node could be executing code (step 4.a) or in a deadlock situation (step 4.b).

```
Main Algorithm:
    Input: Program to run + buggy test case
        + (optional) similar baseline test case
    Output: Sequence of actions or UI events which
        trigger the performance problem, and trace
        of execution across event handlers

1. Run program under \xxx, trigger baseline case
    (if possible) and then buggy test case
2. Set heuristics = { default heuristics }
3. Compute graph using current heuristics with
    Algorithm ComputeControlFlowGraph
4.a. If the buggy case is busy executing code (livelock):
    run backward traversal of edges until UI event found
4.b. Else, given a hanging node N_1, use Algorithm
    FindSimilarNode to obtain an equivalent
    in the baseline test case,
    N_0 -> FindSimilarNode(N_1)
    5.a. Compare nodes N_0 and N_1, and automatically
        diff the two cases, moving through history
        semi-automatically with user input
    until useful UI events are found
        [Algorithm AssistedGraphDiff]
    5.b. If new heuristics were added, go to step 3.
6. Return set of UI events found
```

Figure 1: Main Argus algorithm.

```
Algorithm AssistedGraphDiff:
  Input: Backward slicing path from the baseline Node
          + Spinning Node + Graph
  Output: Possible root cause of thread blocking
  1. get timestamp of Spinning Node as t_spinning
  2. For every thread, node in the backware slicing path
     get timestamp of node in the baseline as t_normal
     check if thread block during (t_normal. t_spinning)
     if true, put the blocking node in the result set
  3. return the result set
```

Figure 2: Argus Assisted graph diff algorithm.

The most common and most complex situation is the latter, where we use a diff between the buggy test case and a baseline test case to further diagnosis. We locate a node N_0 in our graph that corresponds to the baseline version of the hanging node N_1 (see §XXX), and diff the subgraphs to determine the differences between the cases. This diffing technique, detailed in §2.1 below, is semi-automated but can integrate user input at each stage to leverage hypotheses or expert knowledge as to why a hang may occur.

Graph diff algorithm We mentioned previously in §2.1 that Argus explores the event graph semi-automatically to narrow in on relevant nodes for diagnosis. This full algorithm, which includes steps that accept manual user intervention, is shown in Figure 2.

Other algorithms More detailed algorithms that require no user intervention are described in later sections. Specifically, ComputeControlGraph is in §XXX and FindSimilarNode is in §XXX.

```
Algorithm FindSimilarNode:
  Input: Checking Events + CurrentNode + Graph
  Output: Similar node set
1. Get the thread id of CurrentNode
2. Iterate the node from the same thread N_i,
   compare them to CurrentNode N_c:
   while event{iter_c} in N_c is not checking event
   and iter_c not reaches end
     iter_c++
   while event{iter_i} in N_i is checking event
   and iter_i not reaches end
     iter_i++
   if iter_c and iter_i do not reach end:
      Compare Event{iter_c} and Event{iter_i}:
        2.a wait events compares the wait resource
        2.b connection events compare their peer
        2.c system call events and user input events
            compare their syscallname and eventname
   if iter_c and iter_i both reach end:
      put N_i into the result set
3. filter the result set if:
   3.a if the wait events in the end of node
       have the same wait result
   3.b if the difference of node time span
       is within threshold
4. return the result set
```

Figure 3: Argus Find similar node algorithm.

2.3. Chromium Spinning Cursor Example

One of the authors experienced first-hand the aforementioned performance issue in Chromium, an open-source browser engine that powers Google Chrome and, starting recently, Microsoft Edge [1]. She tried to type in the Chromium search box a Chinese word using SCIM, the default Chinese Input Method Editor that ships with MacOS. The browser appeared frozen and the spinning cursor occurs for a few seconds. Afterwards everything went back to normal. This issue is reproducible and always ruins her experience, but it is quite challenging to diagnose because two applications Chromium and SCIM and many daemons ran and exchanged messages. This issue was reported by other users for other non-English input methods, too.

To diagnose this issue with Argus, the author started system-wide tracing, and then reproduced the spinning cursor with a Chinese search string typed via SCIM while the page was loading. It produced normal cases for the very first few characters, and the browser got blocked with the rest input as spinning cases. The entire session took roughly five minutes.

She then ran Argus to construct the event graph. The graph had 2,749,628 vertexes and 3,606,657 edges, almost fully connected. It spans across 17 applications; 109 daemons including `fontd`, `mdworker`, `nsurlsessiond` and helper tools by applications; 126 processes; 679 threads, and 829,287 IPC messages. Given the scale of the graph and the diverse communication patterns, it would be extremely challenging for prior automated causal tracing tools [3, 23, 4, 8] because they handle a fairly limited set of patterns. Tools that require manual schema [5, 17], would be prohibitive because developers would have to provide schema for all involved applications and daemons.

Next she ran Argus to find the event in the main thread of the browser process. Argus returned a `cv_timed_wait` event (Figure **??**) that blocked the main thread for a few seconds. Inspection of the lightweight call stack revealed that this wait happened within a call to `TextInputClientMac::GetFirstRectForRange`. Without knowing the application's semantics, she could not understand this method. Thus she ran Argus to compare the spinning case to a normal case. Argus searched in the main thread of the browser process for vertexes similar to this wait waiting vertexes (contain `write_file, cv_timed_wait` in this case) similar to this wait, found three, and confirmed with the user which one she wanted.

Argus then found the normal-case wake-up path shown in the figure, which connects five threads. The browser main thread was signaled by a browser worker thread as shown in step ① of backward slicing in Figure **??**, which in turn `read_file` in step ② for IPC from a worker thread of `renderer`, the daemon for rendering screens. The `renderer` worker thread is woken up by the `renderer` main thread to `read_file` ③, which in turn `recv_msg` ④ from `fontd`, the font service daemon. From this path, we could guess that `GetFirstRectForRange` was for the browser to understand the bounding box of the search string. Argus further compared the wake-up path with the spinning case, and returned the `wait_semaphore` event in the `renderer` main thread, the culprit that delayed waking up the browser main thread over 4 seconds.

What caused the wait in the `renderer` main thread though? She thus continued diagnosis and recursively applied Argus to the wait in `renderer`, and got the wake-up path shown in the figure for this wait. Inspection reveal that the `renderer` requested the browser's help to render Javascript and was waiting for a reply. At this point, a circular wait formed because the browser was waiting for the `renderer` to return the string bounding box and the `renderer` was waiting for the browser to help render Javascript. This circular wait was broken by a timeout in the browser main thread (the `cv_timed_wait` timeout was 1,500 ms). While the system was able to make progress, the next key press caused the spinning cursor to display for another 1,500 ms. The timeout essentially converted a deadlock into a livelock.

2.4. Limitations

Argus is designed to support interactive debugging of performance issues. To incrementally obtain more fine-grained event traces, it needs to rerun an application to reproduce a performance issue. Thus, if the issue is difficult to reproduce, we have to rely on the log collected by the lightweight system-wide tracing for debugging, and lose the benefits of interactivity. Fortunately, a performance issue that almost never reproduces is probably not as annoying as one that occurs

frequently.

We implemented Argus in the closed-source MacOS which presents a harsh test for Argus, but we have not ported Argus to other operating systems yet. It is possible that the ideas and techniques do not generalize to other operating systems. However, modern operating systems share many similarities, and good ideas tend to flow both ways, so we are hopeful that the ideas in Argus are generally applicable. Similarly, the applications and performance issues used in our evaluation may be non-representative.

3. Argus Graph Computing

3.1. Event Graph

Argus constructs dependency graphs with the events from tracing logs. Tracing logs contain sequence of events per thread. Each event stands for an execution step in a thread. They are grouped into nodes and IPCs, asynchronouns calls and thread wakeups serve as edges; some edges can be inside a single thread.

The events traced in Argus are carefully selected for three main purposes: preserve semantics for the node, indentify node boundary inside a thread, and provide connections between nodes. We classify them into three categories: semantics events, boundary events and connection events, as listed in Table 1.

Categories	Event types
Semantics Events provide hints for user interactive debugging	System_call Back_trace NSApp_event
Boundary Events construct nodes	Interrupts Sharetime_maintenance Wait Dispatch_invoke Runloop_invoke Mach_message
Connection Events add edges	Wake_up Mach_message Dispatch_enqueue Runloop_submit Share_flag_write Share_flag_read

Table 1: Event Type Categories.

Given the prevalent of multi-threading and multi-processing programs, bugs are much more complicated. The long opening bugs are usually have several threads involve, even across process boundaries. As an example, the always timeout on particular synchronization primitive in one thread usually need to trace back to find the other thread that was responsible for signal on the primitive. Compared to the existing debugging tools like lldb and spindump, the dependency graph is useful in that 1) it provides thread relationships all over the system across processes boundary and timing boundary and 2) it records execution history for an input event before users capture hangs with their eyes.

3.2. Inherent Inaccuracy

However, to construct an accurate and complete dependency graph is difficult, if not impossible. The graph is inherently inaccurate. That one thread wakes up the other thread does not always stand for a causility between them. In implementation, Argus filters out some of the definitive noise in the following types.

- interrupt processing and kernel sharetime maintanance that take over current thread context.
- timer expiration in the kernel which clears up all the waiting threads on an event source.

In addition to the known definitive noise, there exist false connections and miss data dependencies based on our experience while building dependency graph with traditional causality tracing. We define them as over connection and under connection respectively.

Over connections occur if intra-thread boundaries are missing from batch processing programming paradigms. (dispatch_mig_service, runloop)

Dispatch message batching The message dispatch service dequeues messages from many processes and staggers processing of the messages. This creates false dependencies between each message in the dispatch queue. As illustrated in the following code snipped from the `fontd` daemon, function `dispatch_execute` is installed as a callback to a dispatch queue. It subsequently calls `dispatch_mig_server()` which runs the typical server loop and handles many messages.

To avoid incorrectly linking many irrelevant processes through such batching processing patterns, Argus adopts the aforementioned heuristics to split an execution segment when it observes that the segment sends out messages to two distinct processes. This pattern does pose a challenge for automated causal tracing tools that assume that the entire execution of a callback function is on behalf of one request. The code shown uses a dispatch-queue callback, but inside the callback, it does work on behalf of many different requests. Any application or daemon can implement its own server loop this way, which makes it fundamentally difficult to automatically infer event handling boundaries.

```
Worker thread in fontd daemon:
dispatch_async(block)

Main thread in fontd daemon:
block = dispatch_queue.dequeue()
dispatch_execute(block)
  dispatch_mig_server()

dispatch_msg_server()
  for(;;)
    mach_msg(send_reply, recev_request)
    call_back()
```

```
    set_reply()
```

Runloop callbacks batch processing As is common in event driven programming, many methods can post a callback and MacOS uses runloop as a common idiom to process callbacks. As shown in the following step-by-step description of the MacOS runloop, an iteration of the runloop does 10 different stages of processing, each of which may do work on behalf of completely irrelevant requests. Since there are no obvious events (e.g., a wait operation) to split the execution, Argus uses instrumentation to add beginning and ending points for MacOS runloops. In general, any application or daemon can create its own version of the runloop, posing challenges for automated inference of event processing boundaries.

```
Run loop sequence of events //developer.apple.com
1-3.Notify observers
4.Fire any non-port-based input sources
5.If a port-based input source is ready and waiting to fire,
    process the event immediately. Go to step 9.
6.Notify observers that the thread is about to sleep.
7.Put the thread to sleep until:
    //one of the following events occurs
    An event arrives for a port-based input source.
    A timer fires.
    The timeout value set for the run loop expires.
    The run loop is explicitly woken up.
8.Notify observers that the thread just woke up.
9.Process the pending event.
  If a user-defined timer fired,
    process the timer event
    restart the loop.
    Go to step 2.
  If an input source fired
    deliver the event.
  If the run loop was explicitly woken up, but not timed out,
    restart the loop. Go to step 2.
10.Notify observers that the run loop has exited.
```

Batching and data dependency in event processing The WindowServer MacOS system daemon contains an event loop which waits on Mach messages. Conceptually, it processes a series of independent events from different processes. However, to presumably save on kernel boundary crossings, it uses a single system call to receive data and send data for an unrelated event. This batch processing artificially makes many events appear dependent, and we split the execution segments to maintain the independence of the events.

This case also illustrates a causal linkage caused by data dependency within one thread. As the code shows, WindowServer saves the reply message in variable `_gOutMsg` inside function `CGXPostReplyMessage`. When it calls `CGXRunOneServicePass`, it sends out `_gOutMsg` if there is any pending message. This data dependency needs to be captured in order to establish a causal link between the handling of the previous request and the send of the reply. Interestingly, it is an example of a data dependency within the same thread. Argus uses watch point registers to capture events on these data flags and establish causal links between them.

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

Under connections are result from particular programing paradigms and data dependencies. Data dependencies inter/intra threads are usually hard to fully exploit in the initial pass of graph computing. (shared flags in Object, data dependency for delay work intra-thread)

CoreAnimation shared flags A worker thread can set a field `need_display` inside a CoreAnimation object whenever the object needs to be repainted. The main thread iterates over all animation objects and reads this flag, rendering any such object.

This shared-memory communication creates a dependency between the main thread and the worker so accesses to these field flags need to be tracked. However, since each object has such a field flag, Argus cannot afford to monitor each using a watch point register. Instead, it uses instrumentation to modify the CoreAnimation library to trace events on these flags.

```
Worker thread that needs to update UI:
ObjCoreAnimation->need_display = 1

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

Spinning cursor shared flag Whenever the system determines that the main thread has hung for a certain period, and the spinning beach ball should be displayed, a shared-memory flag is set. Access to this flag is controlled via a lock, i.e. the lock is used for mutual exclusion, and does not imply a happens before relationship. Thus, Argus captures accesses to these flags using watch-point registers to add causal edges correctly.

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

```
Main thread
Convert1CGEvent(0x1);
if (sCGEventIsMainThreadSpinning == 0x1)
  CGSConnectionSetSpinning(0x0);
  sCGEventIsMainThreadSpinning = 0x0;
  sCGEventIsDispatchedToMainThread = 0x0;
```

In addition, the spurious edge introduced by mutex lock is also a challenge in debugging. The synchronization on mutex lock reflects one thread wakes up the other thread. They can depend on each other like producer and comsumer, while no causality exists in the case of contention for a shared resource. Making the graph completely sound without user interaction is almost impossible given the essential attribite of commericial operating system as a grey box.

3.3. User Interactions

As we mentioned above, to figure out all the over connections and under connections before hand is almost impossible. Instead, users can find out the descrepency on the dependency graph while checking the computing result. For example, we noticed that two unrelated applications connects to one node in the graph, which leads to the manifestation of kernel thread batch processing on timers. Users can gradually inject their knowledge until the graph is reasonable for following debugging.

Over connections in this kind can be eliminated by adding heuristics to the process of graph computing. In addition, users can also binary instrument the image with the APIs provided by Argus to amending the missing boundaries. On contrary, if users discover under connections due to missing data dependency, they can make use of the Argus's hardware breakpointer tool to monitor the data and add rules to recompute the graph.

Like other causality tracing approaches, Argus is a general framework and tested on limited data set. Allowing user interaction makes the dependency graph more practical and useful case by case.

3.4. Graph Computing Algorithm

In the section, we describe the algorithm Argus used to generate an event graph. The algrithm has two main steps: construct nodes with heuristics base on boundary events, and generate edges from the connection events.

A node is a sequence of events derived from the execution of a task in a thread. As is shown in Algorithm 1, Argus checks events per thread and applies heuristics when a boundary event is encoutered. Argus provides 5 default heuristics. 4 of them shares the idea of general cuasual tracing, 1 is used to work around unknown programming pargdigms, and more are expected from users to improve graphs. We discuss them in the following paragraph.

As known noises, event sequences for interrupt processing and kernel maintainance are removed from threads, as is shown in 1.a. The second heuristics 1.b treats a wait event as an end of node. Wait event usually indicates a thread switches to other tasks, but it is not always true considering a thread may park due to low thread priority. One of the examples in MacOS is the pause of worker threads draining a low priority diapatch queue. Dispatch queues are FIFO queues to which an application can submit tasks in the form of block objects. Wait events should not divide the block objects, otherwise it may result in a missing connection. To save the intergrity of block objects, Argus GraphComputeAlgorithm keeps a counter `callout_level` to mark if the currently checking event is inside a block object. Only the begin and end of Dispatch_invoke are served as the boundary for the node, as is shown in the step 1.c. The hueristics 1.d is also for a batch processing mechanism. Runloop is an event processing loop that used to schedule work and coordinate the receipt of incoming events. The begin and end of the work invocation are served as boundaries. The last default heuristics 1.e is used to work around the problem that batch processing programming paragdigms are hard to exploit completely, as listed in previous subsection 3.2. Argus makes use of mach messages to avoid clustering multiple tasks due to unknown batch processing. It defines `IPC_peer_set` to compute the set of mach message receivers/senders for every node. For every mach message, the algorithm checks whether its peer process exists in `IPC_peer_set`, and adds the message into the current node if the condition is true or the set is empty. Otherwise it add a boundary for current node and begin a new node for the mach_message event.

Edges connect nodes, both intra-thread and iter-thread, with connection events. Argus walks through each connection event type to apply heuritics as follows. First, the return from a wait operation causally depends on the wake-up operation. An edge is defined from the wake-up event to the first event after the wait returns. Argus also add a weak edges from the wait event to the wake-up event in 2.a. Mach message is the core of ipc mechanism implemented in kernel, upon which higher level RPC are built. Argus connects the sender and receiver of a mach message. For messages that expect a reply, Argus also connects the receiver of the original message and the sender of the reply message in 2.b. As discussed above, dispatch queue and runloop are popular batch processing programming paradigms in MacOS, Argus connects submissions of a task and executions of the task, which are listed in 2.c and 2.d respectively. Similarly, Argus adds edge from a timer armed event to its triggered event in 2.e. Shared flag can either be traced with binary instrument, such as `need_display` flag for CoreAnimation in subsection **??**, or with breakpoint watcher command line tool provided by Argus. Edges from the flag set to its read are added as 2.f. Since share variables are hard to completely exploited, and the causality can be complicated than the writer-reader pattern, user interaction are expected to remedy event graphs.

Finally, the computing graph is generated and subject to the improvement with more input heuristics.

Algorithm 1 Argus Compute Graph algorithm.

Require: Heuristics set + parsed tracing events
Ensure: Control flow graph

```
1:  function ComputeGraph
2:      callout_level ← 0
3:      IPC_peer_set ← {}
4:      current_node ← NewNode
5:      for Event: Events in a thread do
6:          switch EventType do
7:              case Interrupt or Timeshare_maintenance
8:                  Remove following events before return
9:              case Wait
10:                 if callout_level equals 0 then
11:                     Add end boundary for current node
12:                     current_node ← NewNode
13:                 end if
14:             case Dispatch_invoke or Runloop_invoke
15:                 Divide each callout into a node
16:             case Mach_message
17:                 if peer ∈ IPC_peer_set or IPC_peer_set equals ∅ then
18:                     update IPC_peer_set
19:                 else
20:                     Add end boundary for current node
21:                     current_node ← NewNode
22:                 end if
23:         Other Heuristics
24:     end for
25:     for Event: Connection Events do
26:         switch EventType do
27:             case Wake_up
28:                 AddWeakEdge(corresponding wait, wake_up)
29:                 AddEdge(wake_up, first event after wait returns)
30:             case Mach_message
31:                 AddEdge(sender, receiver)
32:                 if ( thenneeds reply)
33:                     AddEdge(receiver, reply sender)
34:                 end if
35:             case Dispatch_enqueue
36:                 AddEdge(Dispatch_enqueue, Dispatch_invoke)
37:             case Runloop_submit
38:                 AddEdge(Runloop_submit, RunLoop_invoke)
39:             case Timer
40:                 AddEdge(Timer armed, Timer callout)
41:             case Share_flag_read
42:                 AddEdge(Share_flag_write, Share_flag_read)
43:         Other Heuristics
44:     end for
45: end function
```

4. Implementation

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

4.1. Instrumentation

Like Detour [14], we use static analysis to decide which instrumentation to perform, and then enact this instrumentation at runtime. On MacOS, most libraries as well as many of the applications used day-to-day are closed-source. Adding tracing points to such code requires binary instrumentation. Techniques such as library preloading to override individual functions are not applicable on MacOS, as libraries use two-level executable namespaces. Hence, we implemented a binary instrumentation mechanism that allows developers to add tracing at any location in a binary image.

To add instrumentation, we insert 5-byte call instructions into the program. The user finds a location of interest in the code related to a specific event, and we overwrite the victim instructions at that location. We create a new trampoline target function, whose first few instructions are those which were overwritten. All of the trampoline functions are grouped together by our tool and a new library is generated. This library provides the same public API as the original and is a drop-in replacement. We load and call the original code as an unmodified shared library. The detours or trampoline calls are added by an initialization function in our new library; we temporarily mark the code region as writable with `mprotect` to calculate offsets and perform the modifications. The initialization is called externally through `dispatch_once`. To use the modified libraries, we simply replace system libraries in their original locations (renaming them so that our code can access the originals).

One potential issue is that we use 5-byte call instructions with 32-bit displacements to jump from the original library to our new one. This design requires that the libraries be loaded within +/- 2GB of each other in the 64-bit process address space. However, since we list each original library as a dependency of our new libraries, the system loader will map each new and original library in sequence. In practice, the libraries ended up very close to one another and we did not see the need to implement a more general long-jump mechanism.

4.2. Tracing Events

Current MacOS systems support a system-wide tracing infrastructure built by Apple. 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 18,560,187 events (with 5 mintes).

4.3. Tracing Custom Primitives

The graph should be incrementally improved with new tracing points. The procedure to discover such programming paradigm can be repeated on regular executions before tracing for diagnosis. 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.

Argus provides two lightweight tools for users to collecting data with incremental tracing, instead of the lldb.

- For any given shared variable of interest, we take advantage of hardware watchpoints. Tracing points are recorded in the watchpoint handler when the variable is accessed. We hook the handler in CoreFoundation to make sure that it is loaded correctly into the address space of our target application.

We set the hardware watchpoint in an ad-hoc manner with a custom command-line tool.

- For any code location where user want to check its call stacks, our interface accepts a tag as input to distiguish. It unwindes the *rbp* from the user stack to store the valid return addresses in the buffer. The buffer is recorded into the log as tracing events. These address woulg go through the offline symbolicator in the graph construction phase.

In Argus, we patched the kernel with 1193 lines of code, and we instrumented the libraries including: libsystem_kernel.dylib, libdispatch.dylib, libpthread.dylib, CoreFoundation, CoreGraphics, HIToolbox, AppKit and QuartzCore with our binary instrumentation libraries. Based on the new libraries creatd, the user can easily add tracing points with exposed API and the usage sample inside.

4.4. Capturing Instructions for Diagnosis

After the offline analysis on the graph, we take the API covers the fine range as input to our debugging scripts. The debugging scripts go throught the instruction from application and higher level frameworks step by step. The purpose is to capture the parameters results from the user interaction. Once a new function begins by checking the instruction, we record the call stacks for comprehension. For API from the low level libraries, such as pthread, we step over and record the return value. The debugging log in this step records the instruction and its address, callstacks when a *call* instruction is reached, and return values of *req* instruction. As the operation are confined in the small range, the overhead is not too much.

Both the execution on normal case and problematic case are recorded, our tool further compares the log and report the difference, with the full call stack.

4.5. Finding Similar Events

The performance isssue caused by the busy processing in UI thread is quite straightforward to diagnoze with out tool. Debugging the UI thread blocking on the contention of resource is much more difficult. In this situation, our tool is required to recognize the corresponding node which obtained the resource in its normal execution.

Node comparision algorithm helps to allieviate users from the burden of inspecting large logs. We first normalize the nodes with selected events. In our system, we exclude the interrupts from the comparison since the number and type of interrupts are usually different from execution to exection. For the events that connected to other events, we normalize it with a peer attribute to record the process id of its connecting peer, We also record the name of the system calls, message id carried in mach_msg for corresponding events. The comparison algorithm omits the repeating times of the same events, by checking if one node contains all distinct events in the other node.

The above step only idenfity the similarity of nodes. We also define the differential attribtes to distinguish the normal node and spinning node, including the waiting time, execution time and system call return values.

5. Case Studies

We applied our tool to figure out the design of spinning wait cursor in MacOS, and uncovered root causes of spinning for 11 applications in total. As we have illustrated the Chromium case in Section2, we will not present it here again.

The spinning beachball is a painful sight for Mac users, signifying that the application is non-responsive. It usually remains for minutes at a time, leaving the user at a loss and unable to do anything productive.

Argus shreds light on the design of the spinning wait cursor with its backward path slicing. We begin the path from the node where spindump, a hang reporting tool, is launched, considering the spindump shares the same triggering condition. For our tracing data, we found out that Spindump is launched after receiving a message from WindowServer which receives a message from the NSEvent thread of the target application first.

We further added call stacks for the messages and revealed two shared variables, "`is_mainthread_spinning`" and "`dispatch_to_mainthread`", are critical in the desgin. The NSEvent thread of the target application fetches CoreGraphics events from WindowServer, converts and creates NSEvents for the main thread. If the main thread is not spinning with "`is_mainthread_spinning`" equals 0 , "`dispatch_to_mainthread`" will be set to 1 and a timer is armed. If the main thread processes the next event before the timer fires, nothing happens and the timer gets rearmed. Otherwise, NSEvent thread sends a message inside "`CGSConnectionSetSpinning`" to WindowServer from the timer handler, and WindowServer notifies the CoreGraphics to draw a spinning wait cursor over the application window.

5.1. System Preferences

System Preferences provides a central location in MacOS to customize system settings. The "Display" pane allows users to configure additional monitors, mirroring or extending their workspace. However, the software does not support disabling monitors that are online (the user must physically unplug monitors). There is a tool called DisableMonitor [10], distributed via GitHub, which addresses this functionality. Surprisingly, we find performance bug in System Preferences when we disable an external monitor, and rearrange the windows in the Display panel afterward. The System Preferences windows freezes for seconds in this situation.

To diagnose this issue, we run the System Preferences app with Argus. We rearrange the displays with two active monitors, and repeat the process with one of them deactivated. Argus collects 132MB data, and constructs dependency graph system-wide in the period, which contains 428,785 vertexes and 320,554 edges.

To diagnose the root cause, Argus first finds out when the NSEvent thread in System Preferences notifies WindowServer to draw the spinning cursor and marks it as *t*. The node in the main UI thread causing the non-responsiveness must overlap the time interval *(t - 2s, t)*. It is not hard for Argus to tell the spinning node in the main UI thread, either busy processing or blocking.

The spinning node in the main UI thread is dominated by *mach_msg* and *thread_switch*, both of which are meant to wait for available data ping from WindowServer. Noticing the timeouts of thread_switch, Argus classifies this case into the third category (§2.1) and heads to find a comparable normal node in the same thread. As its semantics are not descriptive enough to identify its comparable node, Argus extends the comparison with its proceeding nodes.

In the normal case, System Preferences gets rid of the thread_switch quickly after receiving messages from WindowServer. It proceeds to *displayReconfigured*. On the other hand, the spinning node ends up sending message for the available datagram ping with *CGSSnarfAndDispatchDatagrams*. By checking the lightweight callstacks, Argus figures out the messages from WindowServer in the previous nodes are responses to data available pings from *activeDisplayNotificationHandler*.

Given the information, we launched the interactive debugging by feeding the debugging script with those APIs mentioned above. We set the method *activeDisplayNotificationHandler* as a breakpoint where the script begins debugging. *displayReconfigured* and *CGSSnarfAndDispatchDatagrams* are used to indicate the end of debugging for the normal case and spinning case respectively.

By diff'ing the two logs, we notice the different branches in *display_notify_proc* called by *activeDisplayNotificationHandler*. The handler depends on received datagram and two shared variables "`_gCGWillReconfigureSeen`" and "`_gCGDidReconfigureSeen`" to finish a display configuration. In both case the first variable is set to indicate the begin of display configuration. In normal case, it receives a datagram which set the variable "`_gCGDidReconfigureSeen`" in *display_notify_proc* and finish display reconfiguration, while in the spinning case such datagram is never received. Instead, an alternative datagram just drive it through *display_notify_proc* without setting any variable, which causes the repeating *thread_switch* in the handler.

[In conclusion, the bug is... would have to be fixed by... we provide a binary fix...]

5.2. Sequel Pro

Sequel Pro is a fast, easy-to-use Mac database management application for working with MySQL databases. It allows user to connect to database with a standard way, socket or ssh. We noticed spinning beachballs while the network connection was lost and Sequel Pro tried reconnections. In the worst case, work in other tabs get lost if the reconnection fails.

We simulate the unstable network environment by randomly disabling the network for 1 second, and collect data in two cases: 1) network reconnection succeeds quickly, and 2) Sequel Pro loses its connection for a while. 2G tracing data are stored and the graph we generated contains 3,650,832 edges and 2,412,236 vertexes.

Argus starts analysis from the spinning node in NSEvent thread of Sequel Pro and finds the corresponding node in its main UI thead. We notice the execution sequence,

poll, write, wakeup, re-arm timer, read

in the node. Sequel Pro gets blocked on the read. As a result, Argus goes to search the similar normal node, which has the same sequence but is not blocked for long. From the node, Argus goes along the weak edge to figure out which node wakes it up and slice path backwards. Argus carries out the comparison of the nodes in the thread before the normal node gets waken and before the spinning beachball shows up. The close examination tells that the main thread is waiting for the kernel thread, which in turn waits for the ssh thread. Such result could hardly figure out by lldb or spindump as both of them only focus on the callstack snapshot for each process, instead of providing the connections based the execution history system wide.

5.3. TeXstudio

TeXstudio is an integrated writing environment for creating LaTeX documents. In their github we noticed some one reported while he was modifying his bib file, the TeXstudio hanged for minutes. Although the issue was closed by author for incomplete information for reproduction, we reproduce it in our machine. With a relatively large bib file from a LaTex project opened in a tab, once we touch the bib file through other editors, vim for exmaple, we would have a spinning beachball on the window.

While the beachball is spinning over the application window, the main thread is busy with *QDocumentPrivate::indexOF(QDocumentLineHandle const*, int)*. Slicing the path from the busy node, Argus tells the busy processing is invoked by the callback from daemon fseventsd. The advantage of Argus over other debugging tools is it helps to narrow down the root cause with the path. If the user reported the bug with Argus' report, it should have provided the author more hints to reproduce the bug successfully.

Another operation triggering spinning beachball in TeXstudio is pasting a large amount of context in any file. The spinning node is busy with *QEditor::insertFromMimeData(QMimeData const*)*, which always invokes *match(QNFAMatchContext*, QChar const*, int, QNFAMatchNotifier)*. Path slicing by Argus attributes it to the user's keybord inputs of cmd+v. The problematic code resides where developers copy data from the pasteboard.

5.4. Installer

As we were told, any spinning beachball in MacOS indicated a bug. In some situation, the "bugs" confuse users. Installer is one of the case. When Installer pops up a window for previleged permission during the installation of `jdk-7u80-macosx-x64`, moving the cursor out of the popup window triggers spinning beachball.

Argus collected 550M data for Installer, and generated a graph with 392199 edges and 294856 vertexes. It figuree out the main thread is blocked in the API *[IFRunnerProxy requestKeyForRights:askUser:]*, which sent a synchronous XPC to daemon *authd* with *xpc_connection_send_messag_with_reply_sync*. In conclusion, if the user does not focus on the popup window, it will be treated as a main thread non-responsive bug.

5.5. Notes

Notes is a notetaking app developed by Apple. We encounter spinning beachballs in the app on a relatively long note. Although it works well when the note is generated and stored, the spinning beachball appears when we try to launch Notes the next time. It prevent us from accessing the note for over one hour.

With the collected tracing data, Argus reports Notes is busy with *NSDetectScrollDevicesThenInvokeOnMainQueue*.

5.6. TextEdit

Select, copy, paste, delete, insert and save are common operations for Mac users who do text editing. However, these operations on a large amount of context usually trigger spinning cursors on text editing softwares, including TextEdit developed by Apple. We traced data from six popular text editing softwares. including TextEdit, Microsoft Word, SublimeText, TextMate, CotEditor and TeXStudio. Argus analyzed the tracing data and reported the spinning node for users to inspect. All of the cases are fell into the non-blocking categories. Our lightweight callstack tells the root cause directly.

For the selection in TextEdit, Argus reports the main thread is busy processing *[NSBigMutableString getCharacters:range:]*, *CFStringGetRangeOfCharacterClusterAtIndex*, *_CFStringInlineBufferGetComposedRange* and *NSFastFillAllGlyphHolesForCharacterRange*. All of them are related to the storage of characters.

For the operation of copy, Argus reports that TextEdit is busy with *get_vImage_converter* and *get_full_conversion_code_fragment* in its main thread. Both of them are called by *[NSTextView(NSPasteboard) _writeRTFDInRanges:toPasteboard:]*.

In the case of paste in TextEdit, it is busy with *DDLookupTableRefLookupCurrentWord_block_invoke_2*, *_RTFGetToken* and *platform_memmove*. All of them called by *-[NSTextView(NSPasteboard) _readSelectionFromPasteboard:types:]*

5.7. Microsoft Word

Copying context in Microsoft Word produced 1.14GB trace file, which contained 474,178 edges and 306,171 vertexes. While the spinning cursor showed up for the application, the main thread was busy with system calls, *lseek*, *fstat64*, *fcntl* and *read* on the same file, *__CFStringCreateImmutableFunnel3* and *platform_memmove* over 2 seconds. They are on the behalf of *-[NSPasteboard _setData:forType:index:usesPboardTypes:]*.

We collected 1.04GB tracing data for Paste operation. Its graph contains 161,921 edges and 110,680 vertexes. The spinning node are dominated by *lseek*, *fstat64*, *fcntl* and *write*.

5.8. SublimeText

We collected 1.16GB trace data for copy in SublimeText. It generates 173,071 edges and 127,488 vertexes. While the cursor is spinning, the main thread of SublimeText is busy encoding characters with *__CFToUTF8Len*. They are called from *CFPasteboardSetData*, which invokded by *px_copy_to_clipboard* in SublimeText source code.

Paste operation in SublimeText also results in the unresponsiveness of UI thread. The size of the tracing data is 505MB. The graph consists of 760,003 edges and 535,106 vertexes. The main thread is busy processing *decode_utf8*, *convert_utf8_to_utf32*, string *append* and *platform_memmove*. All are called from *px_copy_from_clipboard*.

Deleting large context in SublimeText also causes spinning beachball. Tracing file for it is 447MB, which contains 789,400 edges and 626,629 vertexes. The main thread is busy processing *TokenStorage::substr*.

5.9. TextMate

Spinning cursor appears when a large amount of context get pasted into a file opened with TextMate. We traced 62MB data and got 73,679 edges and 50,860 vertexes. The main thread is busy with *-[OakTextView paste:]* which iterates through paragraphs, fetchs characters and processed with *CFAttributedStringSet* and *TASCIIEncoder::Encode*.

5.10. CotEditor

Different from other text editing softwares, CotEditor does not trigger spinning beachball while copying a large amount of context. Although paste operation causes the spinning, the path slicing attributes it to the insertion of a new line. The main thread is busy on processing *-[NSBigMutableString characterAtIndex:]* and *CFStorageGetValueAtIndex* in the callstacks. They are invoked after the user hits the return key to insert a line.

Most of the text editing cases fall into the category of thread busy processing. Although sampling with lldb can also figure out the most frequently called APIs, Argus provides the path slicing over the execution history and can connect them to the

application events, such as user inputs, daemon notifications and so on.

6. Performance Evaluation

We deployed Argus on a Mac OS X x86 system, model MacBookPro9,2. The model has the Intel Core i5-3210M CPU with 2 cores and 4 thread, 10GB DDR3 memory, and we replaced the hard disk with a 1T SSD. The tracing tool is running in the background 24X7. Once the spinning cursor appears on the screen, we store the tracing data for root cause analysis.

In previous section, we studied real-world software, including TextEdit, Notes, Installer, System Preferences and Chromium. TextEdit, Notes, Installer and System Preferences are distributed by Apple. Our tool can take over most burden searching and comparing work from the user, and make the diagnosis much easier in the wild.

In this section we present the performance impact of the live deployment of Argus. As we use the ring buffer to collect events, the storage cost can be adjusted and is fixed to 2G in our experiments. Since the internal memory used to collect data is fixed to 512M, so the overhead of the memory is pretty low with regards to the memory usage of morden applications. We show the CPU overhead of Argus first for the iBench Running on the system with the tracing on and off. In the following description, we call the environment without the tracing on as bare run, and otherwise tracing run.

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

Table 2: Score From iBench

We show the five runs of both cases in Table2. For each run, the machine is clean boot for each run. The scores are quite close and show no difference on the system running with and without Argus.

We also perform the evaluation on benchmark of Chromium, while recoding the time usage. The telemetry from Chromium project is used to measure the CPU overhead of Argus. In the experiment, it uses the chromium to launch webpage and scroll over. The result is shown in Table 3.

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

Table 3: Chromium benchmark: telemetry

As shown in Table 3, the overhead for real, user and sys are 1%, 0% and 14% respectively.

7. Related Work

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

Event tracing. Magpie [5] is perhaps the closest to our work. It monitors server applications in Windows with the goal to model the normal behaviors of a server application in response to a workload. This model further helps detecting anomalies statistically. In contrast, Argus's goal is to identify the root causes of performance issues. Its graphs are not request graphs, but rather graphs that may contain many requests. In addition, it logs normal and abnormal executions in the same event trace. In addition, Magpie requires a manual-written event schema for all involved applications to capture precise request graphs, whereas Argus has a simple, application-agnostic schema for system-wide tracing and enables users to provide more application-specific knowledge on demand.

Panappticon [23] monitors a mobile system and uses the trace to characterize the user transactions of mobile apps. Although it aims to track system-wide events and correlate them without developer input, it supports only two models of communication: work queue and thread pooling.

AppInsight [16] instruments application to identify the critical execution path in a user transaction. It supports the event callback pattern, and does not trace across process or app boundaries.

XTrace, Pinpoint and etc [11, 6, 7] trace the path of a request through a system using a unique identifier attached to each request and stitch traces together with the identifier. Argus does not assume the presence of a unified identifier in closed-source, third-party applications, frameworks, and libraries.

Aguilela [3] uses timing analysis to correlate messages to recover their input-output relations while treating the application as a black box.

Performance anomaly detection. Several systems detect performance anomalies automatically. [12, 22] leverage the user logs and call stacks to identify the performance anomaly. [8, 18, 20, 9] apply the machine learning method to identify the unusual event sequence as an anomaly. [21] generates the wait and waken graph from sampled call stacks to study a case of performance anomaly.

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

8. Conclusion

Our key insight in this paper is that causal tracing is inherently imprecise. We have reported patterns we observed that pose big precision challenges to causal tracing, and built Argus, a practical system for effectively debugging performance issues in MacOS applications despite the imprecision of causal

tracing. To do so, it lets a user provide domain knowledge interactively on demand. Our results show that Argus effectively helped us locate all root causes of the issues, including a bug in Chromium, and incurred only 1% CPU overhead in its system-wide tracing.

References

[1] 2008.

[2] 2012.

[3] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 74–89. ACM, 2003.

[4] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation OSDI 12*, pages 307–320, 2012.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[6] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.

[7] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation OSDI 14)*, pages 217–231, 2014.

[8] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, volume 4, pages 16–16, 2004.

[9] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[10] Eun, 2018.

[11] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[12] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, pages 145–155. IEEE Press, 2012.

[13] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. A file is not a file: understanding the i/o behavior of apple desktop applications. *ACM Transactions on Computer Systems (TOCS)*, 30:10, 2012.

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

[15] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 26–26. USENIX Association, 2012.

[16] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation OSDI 12*, pages 107–120, 2012.

[17] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, volume 6, pages 9–9, 2006.

[18] Ali G Saidi, Nathan L Binkert, Steven K Reinhardt, and Trevor Mudge. Full-system critical path analysis. In *ISPASS 2008-IEEE International Symposium on Performance Analysis of Systems and software*, pages 63–74. IEEE, 2008.

[19] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *OSDI*, volume 10, pages 163–176, 2010.

[20] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.

[21] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ACM SIGPLAN Notices*, volume 49, pages 193–206. ACM, 2014.

[22] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation OSDI 12)*, pages 293–306, 2012.

[23] Lide Zhang, David R Bild, Robert P Dick, Z Morley Mao, and Peter Dinda. Panappticon: event-based tracing to measure mobile application and platform performance. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2013.

[24] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation OSDI 16)*, pages 603–618, 2016.