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

Abstract

Prior systems use causal tracing, a powerful technique that traces low-level events and builds request graphs, to diagnose performance issues. However, they all assume that accurate causality can be inferred from low-level tracing with supported communication patterns or obtained from a developer's schema upfront for all involved components. Unfortunately, based on our study and experience of building a causal tracing system on macOS, we show it is difficult, if not impossible, to get accurate request graphs. We present Argus, a practical system for debugging performance issues in modern desktop applications despite the inaccuracy of causal tracing. Argus lets a user inspect current diagnostics and provide domain knowledge on demand to counter the inherent inaccuracy of causal tracing. We evaluated Argus on 11 real-world, open spinning-cursor issues in popular macOS applications. The root causes of these issues were largely previously unknown. Our results show that Argus effectively helps us locate all root causes of the issues and incurs only 7% CPU overhead for 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 [23]. To manually reconstruct a graph of execution segments for debugging, developers have to sift through massive amounts of log entries and potentially code of related application components [16, 26, 31, 33, 35]. 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 causing a spinning cursor in macOS when a user switches the input method [10], was first reported in 2012. Developers attempted to add timeouts to work around the issue. Unfortunately, the bug has remained open for seven years and the timeouts obscured diagnosis further.

Prior work proposed what we call *causal tracing*, a powerful technique to construct request graphs automatically [15, 21, 27, 28, 34]. 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. Compared to debuggers such as *spindump* that capture only the current system state, causal tracing is effective at helping developers understand complex causal behaviors and pinpoint the root causes for real-world performance issues.

Prior causal tracing systems all assume 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. 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 related work in a request. Unfortunately, based on our study and experience of building a causal tracing system for macOS, we find that modern applications violate these assumptions. Hence, the request graphs computed by causal tracing are inaccurate in several ways.

First, an inferred segment can 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 can miss numerous causal edges. For instance, consider data dependencies in which the code sets a flag (e.g., "*need_display* = 1" in macOS animation rendering) and later queries the flag to process a request further. This pattern is broader than ad hoc synchronization [30] because data dependency occurs even within a single thread (such as the buffer holding the reply in the preceding WindowServer example). Although the number of these flags may be small, they often express critical causality, and not tracing them would lead to many missing edges in the request graph. Third, inferred edges can be superfluous because wake-ups do not necessarily reflect causality. Consider an *unlock()* operation waking up an thread waiting in *lock()*. This wake-up can just be happenstance and the developer's intent is mutual exclusion. However, the two operations can also enforce a causal order.

We believe that, without fully understanding of application semantics, request graphs computed by causal tracing are *inherently* inaccurate and both over- and under-approximate reality. Although developer annotations can help improve accuracy [21, 28], modern applications use more and more third-party libraries, whose source code is not available. Given the frequent use of custom synchronizations, work queues, and data flags in modern applications, it

is hopeless to count on manual annotations to ensure accurate capture of request graphs.

In this work, we present Argus, a dramatically different approach in the design space of causal tracing. As opposed to full manual schema upfront for all involved applications and daemons [14, 21, 28], Argus calculates an event graph for a duration of the system execution with both true causal edges and weak ones, and enables users to provide necessary schematic information on demand in diagnosis. Specifically, it keeps humans in the loop, as a debugger should rightly do. Argus queries users a judicially few times to (1) resolve a few inaccurate edges that represent false dependencies and (2) identify potential dependency due to data flags.

Argus targets two groups of users in mind: (1) technical users who wanted to compile informative bug reports for the applications crucial to them and (2) developers of the application or system who wanted to either pinpoint the issue in their component or identify which component is responsible so they can assign the issue to the corresponding developers. Today's applications are often built on top of many third-party, closed-source libraries and frameworks such as Cocoa [**?**], and the operating system itself is often closed-source. Besides technical users, even developers of the application or system may not have access to the code of some critical components in the software stack. We thus explicitly built Argus to work in closed-source settings.

We implement Argus in macOS whose frameworks and applications are mostly closed-source. This closed-srouce environment therefore provides a true test of Argus. We address multiple nuances of macOS that complicate causal tracing, and build a system-wide, low-overhead, always-on tracer. Argus enables users to optionally increase the granularity of tracing (e.g., logging call stacks and instruction streams) by integrating with existing debuggers such as *lldb*.

We evaluate Argus on 11 real-world, open spinning-cursor issues in widely used applications such as 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 helps us non-developers of the applications find all root causes of the issues, including the Chromium issue that remained open for seven years. Argus mostly needs only less than 3 user queries per issue but they are crucial in aiding diagnosis. Argus is also lightweight: its systems-wide tracing incurs only 7% CPU overhead overall.

This paper makes three main contributions: (1) our conceptual realization that causal tracing is inherently inaccurate, and our approach of interactive causal tracing that explores a novel point in the design space; (2) our system Argus that performs system-wide causal tracing with little overhead, embraces inaccuracy in the event graphs, and diagnoses issues using human-in-the-loop algorithms; and (3) our results

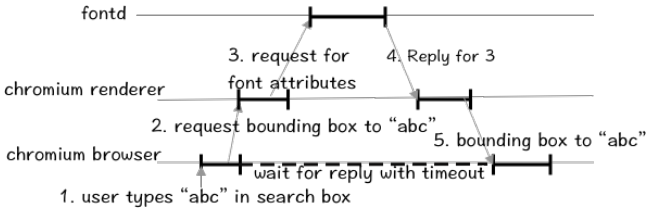


Figure 1. Handling User Typing in Chromium. The *browser* and *render* each split the work among a main thread and a worker thread. Front service daemon *fontd* uses thread pooling. For clarity, only processes are shown, not their threads.

using Argus to uncover the root causes of 11 real-world macOS spinning cursors, many of which have remained open for years.

This paper is organized as follows. In Section 2, we introduce the causal tracing and prior works. In Section 3, we present an overview of using Argus and a Chromium example. In Section 4, we report inherently inaccuracy patterns observed in macOS , and Section 5 describes our tracing implementation and tools for user interaction. Section 6 demonstrates the methodology and results of case studies, and the performance evaluation. We summarize related work in Section 7, and end with conclusion in Section 8.

2 Background: Casual Tracing

Modern applications are often highly concurrent, spreading the handling of an external request in multiple threads, processes, and asynchronous contexts, each of which may multiplex on different requests. Causal tracing thus aims to reconnect the execution segments physically separated in different execution contexts but logically on behalf of the same request. Figure 1 shows an example in Chromium, an open-source browser engine that powers Google Chrome and, starting recently, Microsoft Edge [9]. When a user types a string in Chromium, the *browser* process sends an IPC message to the *render* process where the rendering view and WebKit code run, to calculate the bounding box of the string. *render* then queries fontd, the font service daemon, to calculate the height and length of the string in the given font. Once *fontd* finishes, it sends an IPC reply to wake up *render* which in turn wakes up *browser*. The handling of user typing is thus split among at least three processes and their relevant threads, and an ideal causal tracing tool would capture these segments (vertices) and their connections (edges) into a generalized control-flow graph to aid developer reasoning.

Compared to tools such as *spindump* that capture only the current system state such as the call stack snapshots of each thread, causal tracing captures the dependency path of events, enabling users to trace across threads and processes to past events. For instance, if *render*'s reply hangs *browser*, causal tracing would enable a developer to examine *fontd*'s

reply to *render*, whereas the *fontd* call stack that *spindump* captures at the time of the hang may be for a completely irrelevant request.

Mechanically, to build an event graph, causal tracing tools must infer the beginning and ending boundaries of execution segments and connect a segment with the other segments that it creates or wakes up. Lacking application semantics, existing tools make various assumptions to infer segment boundaries and connections. We select AppInsight and Panappticon, two tools closely related to Argus, and describe how they do so.

AppInsight is designed to help developers understand the performance bottlenecks in mobile applications. To infer execution segment boundaries, it interposes on the interface between the application and framework, and assumes that the application follows mostly the event callback programming idiom. The entry and exit of a callback invocation delimit an execution segment. If this segment installs additional callbacks or signals a waiting thread, then AppInsight connects the corresponding segments.

Panappticon detects performance issues in Android applications. It traces low-level events in the Android system and libraries. To infer execution segment boundaries and connections, it assumes two programming idioms: work queue and thread pooling. For work queue, it marks the handling of a work item as an execution segment, and connects this segment to the segment that dispatches the work item. For thread pooling, it marks from the wakeup of a worker thread to the wait as an execution segment, and connects this segment to the segment waking up the worker thread.

3 Overview

In this section, we describe the design choices in Argus's event graphs (§3.1), its work flow (§3.2), its diagnosis algorithm (§3.3), an example diagnosis session on a real Chromium spinning-cursor issue (§3.4), and its limitations (§3.5).

3.1 Argus Event Graph

Compared to prior tools, Argus differs in three key design choices of its event graphs. First, since event graphs are inherently inaccurate, Argus embraces inaccuracies and mitigates them. For instance, instead of assuming a small fixed set of communication idioms, Argus is tracing general wake-up and wait operations inside the kernel to ensure coverage across diverse user-level custom synchronization primitives because their implementations almost always use kernel wake-up and wait. This approach necessarily includes spurious edges in the graph, including those due to mutual exclusion and context switch by interrupts. Argus filters some of these edges automatically such as those added due to interrupts because they do not reflect application intents. It applies several effective heuristics to further refine them

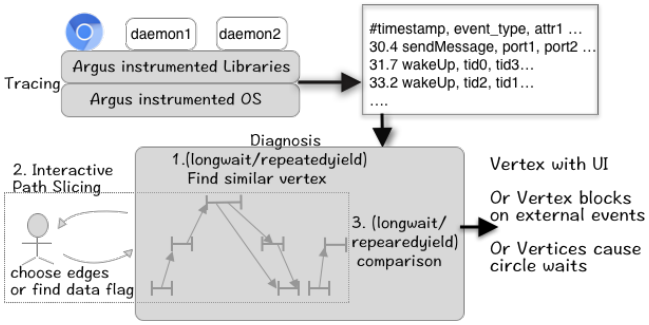


Figure 2. Argus Work Flow

(see §4), and queries the user interactively when needed to resolve ambiguity.

Second, Argus differentiates strong and weak causal edges in its graph. When there is strong evidence of causality, such as callback dispatch, it adds strong edges. When it adds edges heuristically to mitigate inaccuracies (see §4), it marks them weak. Its diagnosis algorithm prefers strong over weak edges, and follows weak edges only when no strong edges present. In case of ambiguity, it queries the user for resolution.

Third, Argus stores extra information in the execution segments of the graphs. Most of this information is semantical, including system calls, call stacks when certain operations such as *mach_msg* are running, and user actions such as key presses. These events enable Argus to find potential normal vs buggy executions for comparative diagnosis, and also provide users necessary information for interactive debugging. In addition, with users' help, Argus also captures key data flag reads and writes which often entail crucial causality.

3.2 Argus Work Flow

Argus operates in two stages as shown in Figure 2. In the first stage, its system-wide tracing tool collects events from Argus-instrumented libraries and kernel using a ring buffer and periodically flushes them to disk. Its overhead is small, so a user may run this tool continuously during normal usage of her computer and invoke Argus to diagnose when a performance issue occurs. Alternatively, she can run this tool reactively once she experiences an issue, and reproduce the issue to collect the events for diagnosis.

In the second stage, the user runs Argus to diagnose in a semi-automated manner. Argus's algorithm constructs an event graph from the traced events and traverses the graph to search for the root cause of the performance issue. Depending on its finding, it queries the user interactively to resolve ambiguity such as determining which edge Argus should follow when a vertex has multiple incoming edges. The result from this stage is either the root cause of the performance issue or that Argus needs more data such as user annotation of a data flag or call stacks for the system calls that the application issues. Such fine-grained data is

not captured by default to reduce tracing overhead. The user can always go back to the first stage to gather such data and diagnose again.

3.3 Diagnosis Algorithm

To initiate diagnosis, Argus needs a problem vertex in the event graph to begin with. While its algorithms and techniques apply to general performance issues, it currently targets spinning cursors – macOS displays them when the current application has not processed any UI events for over two seconds – because they are quite annoying to users, especially several of the authors. The problem vertex is the one in the application's main thread, the designated UI-handling thread in macOS, concurrent to the spinning cursor.

Given the event graph and the spinning vertex, Argus runs Algorithm 1 to pinpoint the root cause. Specifically, upon examining what the main thread is actually doing, there are three potential cases.

- **LongRunning** (lines 3 - 5). The main thread is busy performing lengthy CPU operations. This case is the simplest, and Argus traverses the event graph backwards to find a slice originating from the offending UI event to the long running CPU operations. This slice is particularly useful for further diagnosing the bug. As shown in Function *InteractiveSlice* in line 26, Argus may encounter vertices with multiple incoming edges or weak edges that do not reflect causality when traversing the graph. It queries the user to resolve them.
- **RepeatedYield** (lines 6 - 11). The main thread is in a yield loop, which is highly indicative it is waiting on a data flag (e.g., "while(!done) thread_switch();"). If Argus cannot find any record of data flags in the spinning vertex, it terminates debugging by prompting the user to identify data flags and re-trace the application. Here we assume that the performance issue reproduces with a reasonable probability because, fortunately, a one-off issue that never reproduces is not as annoying as one that occurs frequently. If Argus finds the data flag the spinning vertex is waiting for, it falls through to the next case.
- **LongWait** (lines 12 - 22). The main thread is in a lengthy blocking wait and the wake-up has been missing. Argus handles this case by finding a normal scenario where the wake-up indeed arrives, and then figures out which wake-up edge is missing in the spinning scenario along the expected wake-up path. Specifically, Argus first finds a similar vertex to the spinning one based on the sequence of events such as system calls in each vertex (more details at the end of this subsection). It then traverses backwards from the similar vertex to find the normal wake-up path. For each thread in the wake-up path, it examines the vertex in the thread right before the time that the spinning vertex waits. If this vertex is also problematic (e.g., another LongWait), Argus appends it to the path of root cause vertices, and applies Function *Diagnose* recursively to diagnose "the culprit of the culprit." For each such vertex, it queries the user to determine whether to proceed or stop. Based on our experience, a few iterations suffice to pinpoint the root cause.

Algorithm 1 Diagnosis algorithm.

Input: g - EventGraph; spinning_vertex- the vertex in the UI thread when the spinning cursor occurs
Output: root_cause_vertices- vertices for user to inspect root cause

```
1:  function DIAGNOSE(g, spinning_vertex)
2:      switch spinning_vertex.block_type do
3:          case LongRunning
4:              slice ← InteractiveSlice(spinning_vertex)
5:              return vertex contains UI event
6:          case RepeatedYield
7:              if DataFlagEvent ∉ {event types in spinning_vertex } then
8:                  Require users to annotate data flag
9:                  abort()
10:             end if
11:             /* Fall through */
12:         case LongWait
13:             similar_vertex ← FindSimilarVertex to spinning_vertex
14:             normal_path ← InteractiveSlice(similar_vertex)
15:             for each t ∈ {threads in normal_path} do
16:                 vertex_t ← vertex in t before spinning_vertex spins
17:                 if vertex_t ∈ {LongRunning, RepeatedYield, LongWait} then
18:                     root_cause_vertices.append(vertex_t)
19:                     root_cause_vertices.append(Diagnose(g, vertex_t)
20:                 end if
21:                 /* if t is normal running, disgnose the next thread */
22:             end for
23:     end switch
24:     return root_cause_vertices
25: end function

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

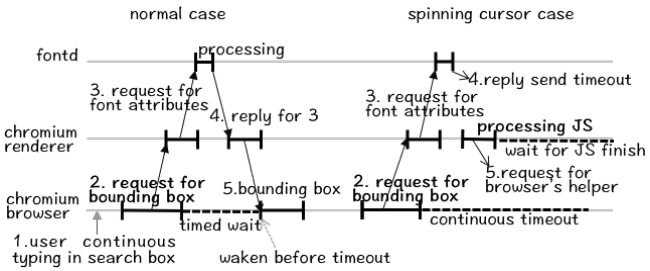


Figure 3. Chromium Spinning Cursor Example. For clarity, only processes are shown, not their threads.

Our experience suggests that the first case is the most common, but the second and third represent more severe bugs. Long-running CPU operations tend to be more straightforward to diagnose with existing tools such as *spindump* except they do not connect CPU operations back to daemons and UI events. Repeated yielding or long waiting cases involve multiple threads and processes, and are extremely hard to understand and fix even for the application's original developers. Such issues may remain unresolved for years, significantly degrading user experience.

Algorithm 1 is semi-automated, integrating user input to leverage hypotheses or expert knowledge as to why a hang may occur. Our results show that user inputs, albeit few, are crucial (§6.3).

Finding similar vertices. Argus finds similar vertices to the spinning one as follows. It computes a signature based on the event sequence in each vertex. For each system call, it extracts the system call name, key parameters, and the call stack. If the same system call repeats, Argus extracts only one avoid treating vertices as dissimilar simply due to nonessential factors (e.g., variable number of *write()* calls due to different data sizes). It extracts return values, too, except for wait operations because the difference between a normal node and a spinning one might just be the return value of a wait operation. If Argus finds multiple similar vertices, Argus usually chooses the most recent one heuristically.

3.4 Chromium Spinning Cursor Example

One of the authors experienced first-hand performance issue in the Chromium browser. She tried to type in the Chromium search box a non-English word with a default Input Method Editor shipped with MacOS. The browser appeared frozen and the spinning cursor showed for a few seconds. Afterwards everything went back to normal. This issue is reproducible and always ruins her experience, but is quite challenging to diagnose because two applications Chromium and IME and many daemons ran and exchanged messages. It was reported by other users for other non-English input methods, too.

To diagnose this issue with Argus, the author follows the steps in Figure 2. She started system-wide tracing, and then reproduced the spinning cursor with a non-English search string while the page was loading. After the very first few characters which the browser handles normally, the remaining characters triggered a spinning cursor. The entire session took roughly five minutes. She then ran Argus to construct the event graph. The graph was highly complex, with 2,749,628 vertices and 3,606,657 edges, almost fully connected. It spanned across 17 applications; 109 daemons including *fontd*, *mdworker*, *nsurlsessiond* and helper tools by applications; 126 processes; 679 threads, and 829,287 messages. Given the scale of the graph and the diverse communication patterns, it would be extremely challenging for prior automated causal tracing tools [12, 13, 18, 34] because they handle a limited set of patterns. Tools that require manual schema [14, 28], would be prohibitive because developers would have to provide schema for all involved applications and daemons.

Next she ran Argus to find the spinning vertex in the main thread of the browser process. Argus returned a *LongWait* event, a *psynch_cv_wait()* with a timeout of 1.5 seconds, and identified a similar vertex in normal scenario where the *wait* was signaled quickly.

Argus then found the normal wake-up path with interactive path slicing, which connects five threads. The *browser* main thread was signaled by a *browser* worker thread, which received IPC from a worker thread of *renderer* where the rendering view and WebKit code run. This worker thread is woken up by the *renderer* main thread, which in turn woken by *fontd*, the font service daemon, as shown in the upper side of Figure 3. Argus further compared the normal case with the spinning case thread by thread as shown in Figure 3, and returned the *LongWait* event on semaphore in the *renderer* main thread, the culprit that delayed waking up the *browser* main thread over four seconds. What caused the wait in the *renderer* main thread though? She thus continued diagnosis and recursively applied Argus to the wait in *renderer*(corresponding normal path is not shown in the figure), and it turned out that *renderer* was actually waiting for the *browser* main thread – a circular wait formed.

Inspect of the reported vertices reveals that the *browser* was waiting for the *renderer* to return the string bounding box for a string, and the *renderer* was waiting for the *browser* to help render JavaScript. This circular wait was broken by a timeout in the browser main thread (the *wait* on *psynch_cv_wait()* timeouts in 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. To verify this diagnosis, we shortened the timeout in the *psynch_cv_wait()* called in the Chromium browser to 150 ms, which proportionally reduced how often this spinning cursor occurs.

```
//worker thread in fontd:
//enqueue a block
block = dispatch_mig_sevice;
dispatch_async(block);
```

```
//main thread in fontd:
//dequeue blocks
block = dequeue();
dispatch_execute(block);
```

```
//implementation of dipatch_mig_server
dispatch_mig_server()
for(;;) //batch processing
    mach_msg(send_reply, recv_request)
    call_back(recv_request)
    set_reply(send_reply)
```

Figure 4. Dispatch message batching

3.5 Limitations

Argus is designed to support technical users or developers for interactive diagnosis of performance issues. It is not for ordinary users without programming expertise.

Argus may require that a performance issue be reproducible so it can capture fine-grained event traces including data flag accesses and call stacks. Fortunately, a performance issue that never reproduces is probably not as annoying as one that occurs now and then. Changing tracing granularity may perturb timing, causing some timing-nondeterministic issues to disappear. However, performance issues tend to last a while for them to be problematic, and we have not experienced such nondeterminism in our evaluation.

To diagnose spinning cursors caused other than long running CPU operations, Argus requires a normal case to compare against. This requirement should generally be fine because if the application always causes a spinning cursor, the issue is most likely already caught during testing.

We implemented Argus in the closed-source macOS which presents a harsh test, but have not ported Argus to other operating systems yet. It is possible that the techniques do not generalize to other operating systems. However, modern operating systems share many similarities, and inspire each others' designs, so we are hopeful that the ideas are generally applicable.

4 Handling Inaccuracies in Causal Tracing

As explained in §2, causal tracing builds a graph to connect execution segments on behalf of a request that spread across separate threads and processes. Based on our experience building a causal tracing system on commercial closed-source macOS, we believe such graphs are inherently inaccurate and contain both *over-connections* which do not really map to causality, and *under-connections*, missing edges between two vertices with one causally influencing the other. In this section, we present several inherently inaccurate patterns (§4.1) and Argus's mitigation strategies (§4.2).

```
//inside a single thread
  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
  }
```

Figure 5. Batching in event processing

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

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

Figure 6. CoreAnimation shared flag

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

```
//Main thread:
{
  ... //pull events from event queue
  Convert1CGEvent(0x1);
  if (sCGEventIsMainThreadSpinning == 0x1){
      CGSConnectionSetSpinning(0x0);
      sCGEventIsMainThreadSpinning = 0x0;
      sCGEventIsDispatchedToMainThread = 0x0;
  }
}
```

Figure 7. Spinning Cursor Shared Flags

4.1 Inherent Inaccuracies

4.1.1 Over Connections

Over connections usually occur when (1) intra-thread boundaries are missing due to unknown batch processing programming paradigms or (2) superfluous wake-ups that do not always imply causality.

Dispatch message batching While traditional causal tracing assumes the entire execution of a callback function is on behalf of one request, we found some daemons implement their service loop inside the callback function and create false

dependencies. In the code snippet in Figure 4 from the *fontd* daemon , function *dispatch_client_callout* is installed as a callback to a work from dispatch queue. It subsequently calls *dispatch_mig_server()* which runs the typical server loop and handles messages from different apps. Any application or daemon can implement its own server loop this way, which makes it fundamentally difficult to automatically infer event handling boundaries.

Batching in event processing Message activities inside a system call are assumed to be related traditionally. However, to presumably save on kernel boundary crossings, WindowServer MacOS system daemon uses a single system call to receive data and send data for an unrelated event from different processed in its event loop in Figure 5. This batch processing artificially makes many events appear dependent.

Mutual exclusion In a typical implementation of mutual exclusion, a thread's unlock operation wakes up a thread waiting in lock. Such a wake-up may be, but is not always, intended as causality. However, without knowing the developer intent, any wake-up is typically treated as causality.

4.1.2 Under Connections

We observe that under connections mostly result from missing data dependencies. This pattern is more general than shared-memory flags in ad hoc synchronization [30] because it occurs even within a single thread.

Data dependency in event processing The code for Batching in event processing above also illustrates a causal linkage caused by data dependency in one thread. WindowServer saves the reply message in variable *_gOutMsg* inside function *CGXPostReplyMessage*. When it calls *CGXRunOneServicePass*, it sends out *_gOutMsg* if there is any pending message.

CoreAnimation shared flags As shown in the code snippet Figure 6, 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.

Spinning cursor shared flag The very fact that spinning cursors are displayed involves data flags. Each application contains an *NSEvent* thread which fetches *CoreGraphics* events from the *WindowServer* daemon, converts them to *NSApp* events to dispatch to the application's main thread, sets flag *sCGEventIsDispatchToMainThread* as shown in Figure 7, and arms a timer to monitor the processing times of the main thread. If any *NSApp* event takes longer than two seconds to process, the main thread will not clear flag *sCGEventIsDispatchToMainThread*. When the timer in *NSEvent* fires, it sets flag *sCGEventIsMainThreadSpinning* and invokes *WindowServer* to display a spinning cursor. An interesting fact is that accesses to these shared-memory flags are controlled via a lock – the lock is used for mutual exclusion, and does not imply a happens before relationship.

4.2 Handling Inaccuracies

In this section, we discuss how Argus leverage heuristics to mitigate over-connections and under-connections.

4.2.1 Mitigating Over-Connections

From a high-level, Argus deals with over-connections by heuristically splitting an execution segment that appears mixing handling of multiple requests.

Specifically, Argus splits based on three criteria. First, Argus recognizes a small set of well-known batch processing patterns such as *dispatch_mig_server()* (§4.1.1) and splits the batch into individual items. Second, when a wait operation such as *recv()* blocks, Argus splits the segment at the entry of the blocking wait. The rationale is that blocking wait is typically done at the last during one step of event processing. Third, if a segment communicates to too many peering processes, Argus splits the segment when the set of peers differs. Specifically, for each message, Argus maintains a set of two peers including (1) the direct sender or receiver of the message and (2) the beneficiary of the message (macOS allows a process to send or receive messages on behalf of a third process). Argus splits when two consecutive message operations have non-overlapping peer sets.

4.2.2 Mitigating Under-Connections

Causal tracing infers boundaries for the background thread with blocking wait. It can break an "atomic" task in the worker thread, e.g.a thread pauses for IO resource. Argus adds weak causal edges to mitigate under-connection. When a weak edge is encountered during diagnosis, it queries users to decide whether to follow the weak edge or stop (§3).

The other primary under-connections are due to data dependencies. Currently Argus queries the user to identify the data flags. It is conceivable to leverage memory protection techniques to infer them automatically, as demonstrated in previous record-replay work [20, 25]. It is out of the scope of this paper and we leave it for future work. Currently, to discover a data flag, the user re-runs the application with Argus to collect instruction traces of the concurrent events in both the normal and spinning cases and detects where the control flow diverges. Argus exposes register values for the basic blocks before the divergence and uncovers the address of the data flag. Once the user identifies a data flag, Argus traces it using either binary instrument, such as the *need_display* flag in CoreAnimation (§4.1.2), or with Argus's watchpoint tool. Argus add a causal edge from a write to a data flag to its corresponding read.

5 Implementation

In this section, we discuss how Argus collects tracing events from both kernel and libraries.

5.1 Event Tracing

Current macOS supports a system-wide tracing infrastructure [1]. It by default stores events in memory and flushes them to screen or disk periodically. Argus extends this infrastructure with a file-backed ring buffer to support larger-scale tests and avoid exhausting the disk. The file size is set to 2GB by default, approximately 19 million events (about 5 minutes of normal operations). The size can be adjust by users to accommodate the desired trace history.

Default tracing points in macOS provide too limited information to enact causal tracing. We therefore instrument both the kernel [2] (at the source level) and key libraries (at the binary level) to gather adequate data. We instrument the kernel with 1,193 lines of code, and binary-instrument the following libraries: *libsystem_kernel.dylib*, *libdispatch.dylib*, *libpthread.dylib*, *CoreFoundation*, *CoreGraphics*, *HIToolbox*, *AppKit* and *QuartzCore* in 57 different places.

5.2 Instrumentation

Most libraries and applications are closed-source in macOS. To hook their functions with techniques such as library preloading are not applicable, as libraries use two-level executable namespace [3]. Hence, we implemented a binary instrumentation mechanism that allows users to add tracing points inside a binary image.

Like Detour [24], we use static analysis to decide where the instrumentation performs and enact it at runtime. The user supplies a sequence of instructions for Argus to search the locations of interest in the image, and a trampoline function which overwrites the sequence of instructions and produces tracing data with API *kdebug_trace* from Apple. Argus generates shell code with the trampoline function to replace the victimzed instructions. All of the trampoline functions are grouped into a new image with an initialization function that triggers the drop-in replacement. Argus configures the image in the following steps to finish the instrumentation: (1) it re-exports all symbols from the original image, so the original code can be called like a shared library; (2) it renames the original image and applies original name to the new one to ensure the modifications are properly loaded; (3) it invokes the initialization function externally with *dispatch_once* when the library loads.

5.3 Tracing Data Flags

As described in (§4), under-connection due to the missing data dependency requires users' interaction. Users specify that reads and writes to a given variable should be considered data dependencies. Data flag tracing is possible through Argus's binary rewriting, but we also provide a simple command line tool which uses watchpoint registers to record *data_flag_write* and *data_flag_read* events. The tool *bp_watch* takes as input the process ID, path to the relevant binary image, and the symbol name of the global variable. Here is a simple example of how a user asks Argus to trace *_gOutMsgPending*:

```
bp_watch Pid/of/WindowServer Path/to/CGs _gOutMsgPending
```

Argus loads the watchpoint handler into the address space of the target application by hooking it to *CoreFoundation*. The handler invokes Apple's API *kdebug_trace* to record the value of the data flag and its operation type (read or write).

5.4 Tracing Instructions and Calls

Users may need to gather more information, such as individual instructions and call stacks, to come up with and verify a binary patch. Argus integrates with *lldb scripts* to capture this information and add it to the corresponding vertices in the event graph. Our *lldb scripts* gather call stacks at relevant locations and parameters tainted by user inputs. To reduce the data collection overhead, only at each beginning of a function call does the script record a full call stack. While *lldb* steps into functions from apps and frameworks to record parameters tainted, it steps over and only records the return value of APIs from low-level libraries (i.e. those with the filename extension *.dylib*).

The combination of instruction-level tracing and occasional call-stacks offers more than enough detail to diagnose even the most arcane issues, and in our experience has been very helpful in multiple steps of an Argus diagnosis.

6 Evaluation

In this section, we describe the methodology of case study, the root causes diagnosed with Argus for performance issues in macOS, and the overhead of Argus tracing system.

6.1 Methodology

Prior work relies on the request graph per transaction to identify bottleneck and speculate possible causes. Panappticon for Android is one and closest to ours in desgin. Their traced events and causality are in a subset of ours. We therefore carry out the study in §6.2 with a microbenchmark to demonstrate the inherent inaccuracy of request graph extracted with Panappticon's causal tracing. We also measure the effectiveness of AppInsight with their methodology.

For real-world case study, we collect performance issues of popular applications, because they likely represent the bugs attractive to tech-savvy users. Among the 26 bugs from the github reports, we reproduced 3 of them successfully. Others are failed either due to the version capacity in ElCapitan or insufficient information in the bug report. 8 performance issues are collected from the daily use applications in the auther's laptop. As a result, we study 11 reproducible cases.

Now we describe how we measure the effects of heuristics which Argus uses to mitigate the graph inaccuracy, and manual efforts required in the diagnosis. We first enable tracing component and reproduce the performance isssues in macOS. The tracing data are collected to construct an event graph, which contains vertices with multi-incoming edges or weak edge. We run Argus diagnosis algorithm, and count how many times we encounter those vertices. In the worst case that we make a wrong decision, before reaching the end of path slicing, Argus allows them to relocate the path to a particular vertex.

6.2 Microbenchmark

In the section, we demonstrate the inaccuracy of request graph extracted with prior causal tracing system. We first describe the study carried out on the application from Apple's sample code *AppList* [4], and then we lists microbenchmarks in Table 1.

The main window of *AppList* shows all running applications. *Hide*, *Unhide*, and *Terminate* buttons are beside the window for users to manipulate a chosen app. In the study, we chose *Sequel Pro* from the list of running applications, and clicked the *Hide*. To find the ground truth for the transaction, we manually check the tracing log with extra instrumentation in event handler. *AppList* feteches user input events from *WindowServer*. The user input event handler invokes *appleeventsd* and *launchserviced*, which communicate with the chosen app *Sequel Pro*. *Sequel Pro* hides itself and *AppList* updates main window. During the process, *AppList* invokes daemons like *cfprefsd*, *distnoted*, *fseventsd*, *syslogd*, *systemstatsd* to communicate with system.

The graph generated with Panappticon contains more information than what we identified. It has 20 processes involved. We identify at least 7 of them are unrelated to the app. 4 of them are connected because of the *WindowServer*'s batching in event processing as shown in Figure 5, and 3 of them are from kernel task's batching in timer processing. In addition, 4 UI events(mouse click, mouse move, system defined, and mouse exit event) passed to *AppList* are present in the same graph because of batching in runloop blocks [5]. All the UI updates are batching in a callout function from the runloop. Without tracking the flags in Figure 6, Panappticon can not seperate them.

AppInsight is effective in tracking the boundary of the event handler, however, it does not track the execution boundaries in daemons. Thus, it can not track the activities in *appleeventsd*, *launchserviced* and *Sequel Pro*.

Without the Apple developers' efforts, Argus's binary instrumentation on runloop is also not enough to identify all under- and over-connections, therefore our event graph exposes 59 vertices with multiple incoming edges.

Microbenchmark	Description
AppList	AppList manipulates running apps in system. ✗Panappticon connects 7 unrelated apps due to lack of boundaries on batching. ✗Argus event graph contains 59 vertices with multi-incoming edges.
AbSearch	AbSearch searches Contacts with first name. ✗Panappticon misses connections to the searching element constuction due to lack of tracing data flag. ✗Argus event graph contains 72 vertices with multi-incoming edges.
AnimationSlider	AnimationSlider moves slide bar with input percentage in textfield. ✗Panappticon connects 5 unrelated apps due to lack of boundaries on batching. ✗Argus event graph contains 9 vertices with multi-incoming edges.

Table 1. Inaccuray of request graph for microbenchmarks.

Bug ID	Application	Bug description
1-Chormium	Chormium	Typing non-english in search box causes webpage freeze.
2-SystemPref	System Preferences	Disabling an online external monitor and rearranging windows causes System Preferences freeze.
3-SequelPro	Sequel Pro	Lost connection freezes the APP.
4-Installer	Installer	Moving cursor out of an authentication window causes freeze.
5-TeXStudio	TeXStudio	Modification on bib file with vim causes its main window hang.
6-TextEdit	TextEdit	Copying text over 30M causes freeze.
7-MSWord	Microsoft Words	Copying a document over 400 pages causes hang.
8-Notes	Notes	Launching Notes where stores a long note before causes freeze.
9-SlText 10-TextMate 11-CotEditor	SublimeText TextMate CotEditor	Copying or pasting in a file with large amount of context causes freeze.

Table 2. Bug Descriptions.We assign each bug in Column **Bug ID** to ease discussion

6.3 Case Studies

In this section, we demonstrate how Argus helps to diagnose 11 spinning-cursor cases in popular applications. Table 2 describes these spinning-cursor cases. In Table 3, we compare Argus with Panappticon and AppInsghit. In the table we list the missing tracing data for their failure. However, Argus event graph does not fully support automatic diagnosis. The user interaction is still required but not overwhelming. As shown in Table 4, up to 3 user queries in most cases suffice to find root cause path accurately. Although complex applications like MicosoftWord and Chromium require more queries, 13 and 22 respectively, many of them result from repeated patterns. They can be easily identified by users.

Bug ID	Panappticon	AppInsight	Argus
2-SystemPref	✗(data flags)	✗(WindowServer)	✓(2 data flags)
3-SequelPro	✗(batching in kernel task)	✗(sshd)	✓(2 ME)*
4-Installer	✗(batching in worker thread)	✗(SecurityAgent, authd)	✓(1 WE, 1 ME)*
5-TeXStudio	✗(appkit event, timer, batching in WindowServer)	✗(fseventd)	✓(3 ME)*

Table 3. Compare Effectiveness of Causal Tracing System. *WE stands for user interaction on Weak Edges, *ME stands for user interaction on Multi-incoming Edges in Argus.

Bug ID	user interactions	size of path with interaction	heuristics	hueristics over interaction
1-Chromium	13	32	303	9.47
2-SystemPref	1	2	30	15.00
3-SequelPro	2	5	264	52.80
4-Installer	2	6	36	6.00
5-TeXStudio	3	6	44	7.33
6-TextEdit	3	21	21	1.00
7-MSWord	22	67	136	2.03
8-Notes	2	10	42	4.20
9-SlText	1	3	3	1.00
10-TextMate	0	3	3	1.00
11-CotEditor	1	4	6	1.50

Table 4. Path slicing for buggy cases. We noticed that Argus's query on incoming edges can be answered with the most recent one, thus we automated the path slicing in Argus with the heuristics. The length of the path is usually much longer due to excessive traverses to daemons and kernel task.

6.3.1 Long Wait and Repeated Yield

In this section, we discuss the cases where the spinning vertex is blocking on wait event or yielding loop, corresponding to LongWait and RepeatedYield. These root causes are mostly can be verified by themselve, as they are manifested by comparing to normal scenarios.

2-SystemPref *System Preferences* provides a central location in macOS to customize system settings, e.g.additional monitors configuration. *DisableMonitor* [11] provides more functionality, enable/disable monitors online. We catch the spinning cursor when we disable an external monitor and rearrange windows in *Display* panel.

The log collected by Argus contains 2 cases: 1) a baseline scenario where the displays are rearranged with the enabled external monitor, and 2) a spinning scenario as we described above. The spinning vertex in the main thread is dominated by system calls, *mach_msg* and *thread_switch*, which falls into the category of Repeated Yield. We discovered data flags, "*_gCGWillReconfigureSeen*" and "*_gCGDidReconfigureSeen*", which signify the configuration status and break the thread-yield loop. Argus reveals that the main thread of *System Preferences*, in the baseline scenario, sets the flags after receiving specific datagrams from *WindowServer*. Conversely, the setting of "*_gCGDidReconfigureSeen*" is missing in the spinning case, and the main thread thus repeatedly sent messages to *WindowServer* for datagram.

In conclusion, we discovered that the bug is inherent in the design of the *CoreGraphics* framework, and would have to be fixed by Apple. We also verified this diagnosis by creating a dynamic binary patch to fix the deadlock. The patched library makes *DisableMonitor* work correctly, while preserving correct behavior for other applications.

3-SequelPro *Sequel Pro* [6] is a fast, easy-to-use Mac database management application for *MySQL*. It allows user to connect to database with socket or ssh. We experienced the non-responsiveness of Sequel Pro when it lost network connection and tried reconnections.

The tracing log contains two cases: 1) a quick network connection during login, and 2) Sequel Pro lost connection for a while. Although Argus identified the spinning vertex and similar vertex with ease, the backward slicing from similar vertex encountered multiple incoming edges, including one from a kernel thread. The kernel thread processes tasks from different applications in a batch. Interaction in the path slicing is helpful to reduce the noise. With comparison to the causal path in normal case, Argus reveals the main thread is blocking on a kernel thread, which waits for a ssh thread. Thus we conclude the root cause is the main thread blocking for network IO.

4-Installer *Installer* [7] is an application that extracts and installs files out of *.pkg* packages in macOS. When *Installer* pops up a window for privileged permission during the installation of *jdk-7u80-macosx-x64*, moving the cursor out of the popup window triggers a spinning cursor.

Argus successfully records the baseline scenario with the following operations. We first type in password in the popup window and then click the back button to reproduce the spinning case by moving cursor. Examining the spinning vertex and its similar vertex, Argus reveals the main thread is waiting for *authd* which blocks on a semaphore. Further diagnosis on *authd* reveals the root cause is the *SecurityAgent*. It processes user input and wakes up *authd* in normal case, but fails to notify *authd* in spinning case. We would suggest an operation for *SecurityAgent* to inform *authd* when the user input box loses cursor, so that the main thread in *Installer* receives proper signal from *authd*.

6.3.2 Long Running

In this section, we discuss the cases where the spinning vertex is busy on the CPU. Most text editing apps fall into this bug category. We studied bugs on TeXstudio, TextEdit,

Microsoft Word, SublimeText, TextMate and CotEditor, to reveal the root causes.

5-TeXStudio *TeXstudio* [8] is an integrated writing environment for creating LaTeX documents. We noticed a user reported spinning cursor on the modification of bibliography (bib) file. Although the issue was closed by the developer for insufficient information, we reproduced it with our bib file around 500 items in a *TeXStudio* tab. When we touch the file in other editors like vim, a spinning cursor appears in *TeXStudio*'s window.

Argus recognizes the spinning vertex belongs to the category of LongRunning. The causal path sliced from the spinning vertex by Argus reveals the long-running function is a callback from daemon *fseventd*, and the long processing segment is busy to CalculateGrowingBlockSize, even without modifications to the file. The advantage of Argus over other debugging tools is it narrows down the root cause with the inter-processes execution path.

6-TextEdit *TextEdit* is a simple word processing and text editing tool from Apple, which often hangs on editing large files. When Argus is used to diagnose this issue, the heuristics of choosing the most recent edge is powerful enough to get the causal path.

The event graph reveals a communication pattern where a kernel thread is woken from I/O by another kernel thread; the woken kernel thread processed a timer callback function armed by *TextEdit* and finally woke a *TextEdit* thread. In the pattern, the kernel thread has two incoming edges. One is from another kernel thread's IO completion, and the other is from *TextEdit*'s timer creation. It not hard to reveal the high level semantics. *TextEdit* arms a timer for IO work, and the kernel thread gets the notification for the completion of IO and processes the timer callback. The success of heuristics is not surprising because the most recent edge in the vertex reflects the purpose of the execution segment. Although the heuristics works for the kind of simple application, it is not general enough to succeed for all patterns.

7-MSWord *Microsoft Word* is a large and complex piece of software. Argus can analyze the event graph, but it identifies multiple possible root causes: the length of path interactively sliced from the spinning vertex is 67, while the slicing with heuristics of choosing the most recent edge generates a path of 136 vertices.

We compared the paths and find they diverge from the third vertex backward from spinning vertex. In the vertex, a *Microsoft Word* thread is woken by another *Microsoft Word* thread, and launches a service *NSServiceController-CopyServiceDictionarie*. The woken thread sends a message to *launchd* to register the new service and waits for a reply message. With the most recent edge heuristics in automatic slicing, Argus chooses the reply from *launchd* as its predecessor. However, a user can more accurately identify that the execution segment is on behalf of another *Microsoft Word* thread. We rely on user interaction in this case to find the true root cause, since Argus is likely identifies all possibilities without priority.

BUG-ID	**costly API**	UI
8-Notes	*1)NSDetectScrollDevicesThe -nInvokeOnMainQueue*	*system defined event*
9-SlText	*1)px_copy_to_clipboard 2)__CFToUTF8Len*	*key c*
10-TextMate	*1)-[OakTextView paste:] 2)CFAttributedStringSet 3)TASCIIEncoder::Encode*	*key v*
11-CotEditor	*1)CFStorageGetValueAtIndex 2)-[NSBigMutableString characterAtIndex:]*	*key Return*

Table 5. Root cause of spinning cursor in editing Apps

	1st	2nd	3rd	4th	5th
without Argus	5.98	6.23	6.18	6.05	6.28
with Argus	6.29	6.01	6.09	6.28	6.01
average overhead	0.13%				

Table 6. Score From iBench

Other Editing Apps Select, copy, paste, delete, insert and save are common operations for text editing. However, these operations on a large context usually trigger spinning cursors. In Table 5, we list the root causes reported by Argus, including the most costly functions in the event handler, and the user input event (derived from path slicing).

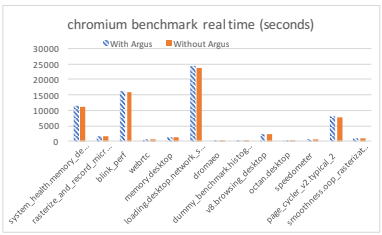
6.3.3 Summary

Overall, in the case of simple text editing applications, Argus can identify the UI event that causes a spinning cursor by merely relying on a few heuristics. However, these heuristics may make the wrong decision in complicated cases, and misidentify the relationships between intra/inter-thread events. It is unlikely that there exists a single graph search method that works in all cases, e.g. when given the choice between multiple incoming edges, the most recent match is sometimes correct, but sometimes not. This is why our system relies on expert knowledge of users to reconstruct a developer's intent and accurately diagnose performance issues.

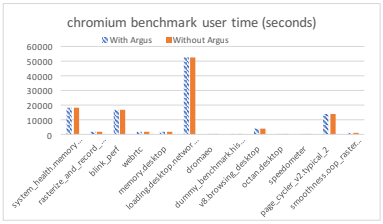
6.4 Performance Evaluation

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

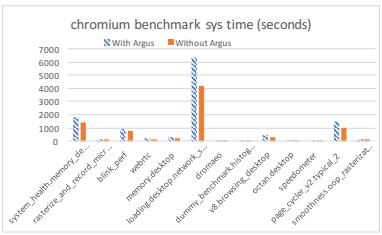
Argus has a very small space overhead with the configuration of its tracing tool. It uses the ring buffer with configured 2G by default to collect tracing events. The memory used to store events is fixed to 512M by Apple, which is pretty low



(a) real time



(b) user time



(c) sys time

Figure 8. Chromium benchmark

	kb/s	With Argus	Without Argus	overhead
bonnie++	read char	21922	22149	0.01
sequential	read block	226931	244089	0.07
	rewrite	246807	267491	0.08
	write char	22924	22936	0.00
	write block	4073361	4396387	0.07
seq	file create	17391	17381	0.00
	file delete	18089	19401	0.07
random	create	17472	17887	0.02
	delete	8849	9567	0.08
iozone	initial write	1199453	1318572	0.09
	rewrite	3663066	4059912	0.10
	average	-	-	0.05

Table 7. IO throughput with bonnie++ and iozone

with regards to the memory usage of modern applications. In the remaining of this section, we measure Argus's overhead overall with iBench scores, IO throughput degradation with bonnie++, iozone and CPU overhead with chromium benchmarks.

iBench We first show the five runs of iBench with and without Argus to evaluate the overall performance. The machine is clean booted for each run, and the higher score means it performs better. As shown in Table6, their performance are almost of no difference, only 0.13% degradation on average.

IO Throughput Next, we evaluate the IO throughput with bonnie++ and iozone. As shown in the Table 7, the throughputs of sequential read and write by characters with and without Argus are almost same. Read and write by block impose less than 10% overhead in both microbenchmarks, bonni++ and iozone. With selected events in our system, the tracing tool integrated in Argus only adds 5% IO overhead on average.

CPU We evaluate Argus's CPU overhead with chromium benchmarks by recording their time usage on real, user and sys. Although the overhead on sys is relatively higher than other two, due to the tracing events usually crossing the kernel boundary, they are not triggered too frequently in our daily software usage, including browsers. The time cost is mostly under 5%, except the *dummy_benchmark.histogram*. As shown in Figure 8, the time overhead for real, user and sys are 7%, 5% and 40% 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.

Paradigm	Panappticon	AppInsight	Argus
Async calls	MessageQueue, ThreadPoolExecutor	Upcall	libdispath runloop timer
IPC calls	✓	✗	✓
Batching	✗	✗	✓
Data flag	✗	✗	✓
Wait-Wakeup	4 primitives	Silverlight methods	system wide

Table 8. Programming paradigms tracked

Event tracing. Panappticon [34] 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 [27] 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. Magpie [14] 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. 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.

Aguilela [12] uses timing analysis to correlate messages to recover their input-output relations while treating the application as a black box. XTrace, Pinpoint and etc [16, 17, 21] 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 comes up violation patterns and does not assume the presence of a unified identifier in closed-source, third-party applications, frameworks, and libraries.

Performance anomaly detection. Several systems detect performance anomalies automatically. [22, 33] leverage the user logs and call stacks to identify the performance anomaly. [18, 19, 29, 31] apply the machine learning method to identify the unusual event sequence as an anomaly. [32] 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 inaccurate. The inaccuracy unlikely gets remedied by heuristics so as to generate a feasible request graph for automatic diagnosis on performance issues. We built Argus, a practical system for effectively debugging performance issues despite inaccurate causal tracing. It lets a user provide domain knowledge on demand to mitigate the inaccuracy. Compared to all upfront knowledge, this method is more general and efficient given the various programing paradigms.

References

[1] https://opensource.apple.com/source/system_cmds/system_cmds-671.10.3/trace.tproj.

[2] https://opensource.apple.com/source/xnu.

[3] http://mirror.informatimago.com/next/developer.apple.com/releasenotes/DeveloperTools/TwoLevelNamespaces.html.

[4] https://developer.apple.com/library/archive/samplecode/AppList/Introduction/Intro.html#//apple_ref/doc/uid/DTS40008859.

[5] https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html.

[6] https://www.sequelpro.com.

[7] https://en.wikipedia.org/wiki/Installer_(macOS).

[8] https://www.texstudio.org.

[9] The Chromium Projects. https://www.chromium.org, 2008.

[10] Issue 115920: Response time can be really long with some IMEs (e.g. Pinyin IME (Apple), Sogou Pinyin IME). https://bugs.chromium.org/p/chromium/issues/detail?id=115920, 2012.

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

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

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

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

[15] Benjamin H. Sigelman and Luiz AndrÃľ Barroso and Mike Burrows and Pat Stephenson and Manoj Plakal and Donald Beaver and Saul Jaspan and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, 2010.

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

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

[18] Cohen, Ira and Chase, Jeffrey S and Goldszmidt, Moises and Kelly, Terence and Symons, Julie. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, 2004.

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

[20] Dunlap, George W and Lucchetti, Dominic G and Fetterman, Michael A and Chen, Peter M. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2008.

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

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

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

[24] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *3rd usenix windows nt symposium*, 1999.

[25] King, Samuel T and Dunlap, George W and Chen, Peter M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

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

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

[28] Reynolds, Patrick and Killian, Charles Edwin and Wiener, Janet L and Mogul, Jeffrey C and Shah, Mehul A and Vahdat, Amin. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

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

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

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

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

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

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

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