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 5 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 [12]. 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, 14, 19, 21, 23]. 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 [22]. 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 [15, 22]. 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 [18] 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, 16], 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 5 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 5 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 3 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

2.1 Argus Work Flow

Figure 1 shows Argus's work flow, which consists of two phases. A user runs command "`Argus start`" to enter the system-wide tracing phase, within which Argus logs events including system calls, inter-process communications (IPCs), and waits and wake-ups from all applications and daemons. Occasionally based on user configurations, it logs data flag accesses leveraging hardware watch-point registers. Each log entry includes a timestamp, the event type, key attributes of the event, and a lightweight call stack obtained by unwinding the stack pointer. Argus implements tracing by instrumenting core system libraries and a small portion of the operating system. The performance impact of this system-wide tracing is low because the logged events themselves are often expensive and require user-kernel crossings, masking the overhead of tracing.

Whenever a user detects a performance issue such as a spinning cursor, she runs "`Argus debug`" to enter the interactive diagnosis phase. Argus initializes this phase by constructing a causal graph from all logged events up to user-specified duration. Rather than requiring a precise application-specific user-written schema, Argus leverages a simple system-wide schema we created to construct approximate graphs (§2.1.1), and relies on interactive user insights for diagnosis (§2.1.2).

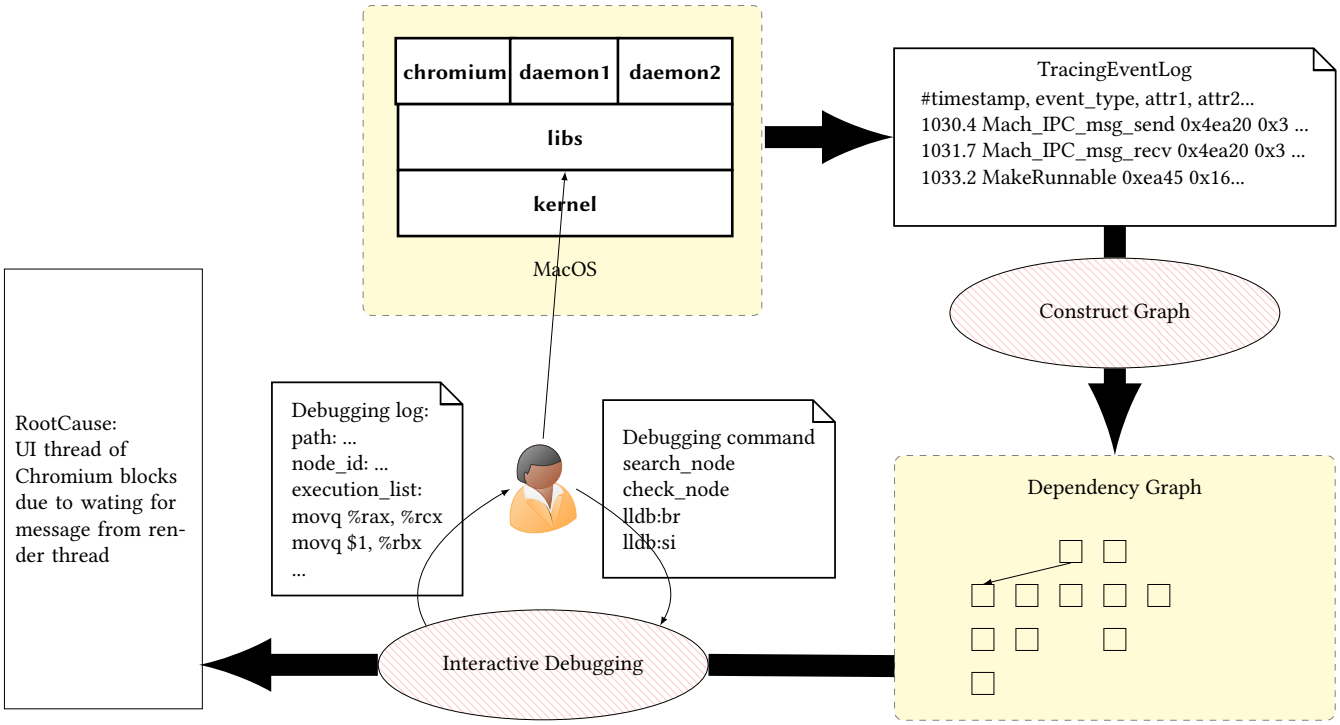

Figure 1. Argus Work Flow

2.1.1 Constructing Event Graphs

Argus defines the beginning of an execution segment as one of the following three types of events (1) the beginning of a thread, (2) the event from a wait operation such as `pthread_cond_wait()` or `mach_msg_receive()`, and (3) the first logged event of a thread because tracing may start mid execution. It similarly defines the end of an execution segment as (1) the exit of a thread, (2) the call to a wait operation, and (3) the last logged event of a thread.

Argus defines the edges as follows. First, the return from a wait operation causally depends on the wake-up operation. Since an application or its libraries may define custom synchronization primitives, Argus traces wait and wake-up operations inside the operating system kernel (the `waitq_assert_wait64_locked` and `thread_unblock` functions in MacOS kernel). This design decision ensures that Argus captures a large set of casual edges at the expense of superfluous edges that do not map to causality. For instance, inside a system call or interrupt handler, the kernel typically checks whether the current process has used up its time slice and, if so, wakes up another process. Argus thus explicitly filters out edges due to kernel maintenance (in `interrupt` and `sched_timeshare_miantenance_continue` functions in MacOS) instead of application intent. Second, the read of a data flag causally depends on the write of the same flag. Edges of this type are few but critical for diagnosis. Third, the timer expiration and cancellation events causally depend on the installation of the timer. Fourth, each execution segment has an incoming edge from the immediately preceding segment in the same thread. Argus considers this type of intra-thread edges weaker than inter-thread edges due to the wait between the two segments. In its analysis, Argus follows intra-thread edges typically only when it cannot find any inter-thread edges.

It is easy for a user to incrementally extend Argus with custom segment boundaries and edges. For instance, to handle batch processing, we created a heuristic that splits a segment if it has outgoing edges to two or more different processes (unless the attributes of the corresponding events indicate otherwise). We also added edges for three data flags and eight custom communication primitives (see §4.3).

2.1.2 Interactive Diagnosis

After Argus builds the event graph, a user can interactively query this graph for diagnosis. For instance, consider a spinning cursor in MacOS which indicates the current application's main thread has not processed any UI events for over two seconds. The user can query Argus to find the ongoing event in the application's main thread concurrent to the display of the spinning cursor. Depending on the type of the event, she can proceed in three directions.

First, if the concurrent event is a busy operation that occupies the CPU, she has found the cause of the spinning

because a busy main thread cannot process UI events. She can examine the event's lightweight call stack. If it does not provide enough details, she can rerun the application and use Argus's fine-grained debugging tool to obtain a more complete call stack, the addresses of the instructions executed, and parameters and return values of call instructions. In general, she can increase debugging details for any event, not just a busy event.

Second, if the concurrent event is a blocking wait, she runs Argus to locate another event in the graph that causes the wake-up to arrive late. Argus does so using the following idea. In the normal case, there must be a path of wake-up edges that leads to the blocking wait, so the main thread starts running again. In the spinning case, somewhere along the path, a thread's wake-up is missing, so this thread is the culprit. Mechanically, Argus first searches the graph to find a similar wait that does not cause a spinning cursor. If there are multiple nodes similar to the wait, Argus asks the user to pick one. It then slices the graph backwards to find the wake-up path. If an event has exactly one inter-thread edge or only an intra-thread edge, it follows the edge. Otherwise, the event must have two or more inter-thread edges, and Argus consults the user to pick one to follow. Given the path, Argus examples the threads in the path one by one, and returns the thread whose wake-up is missing in the spinning case.

Third, if the concurrent event is a thread yield, it is highly indicative that the main thread is waiting on a data flag (e.g., "while(!done) thread_switch();"). To discover a data flag, the user reruns 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, the first type of spinning cursor is more common but the second and third types cause the most harm. The reason is that the first type tends to be straightforward to diagnose, so they are fixed quickly. The second and third types involve multiple threads, so they are extremely hard to understand and fix even for the application's own developers. Therefore, they remain open for years and ruin many users' experiences.

2.2 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, 4, 8, 22] because they handle a fairly limited set of patterns. Tools that require manual schema [5, 16], 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 2) 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 2, 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

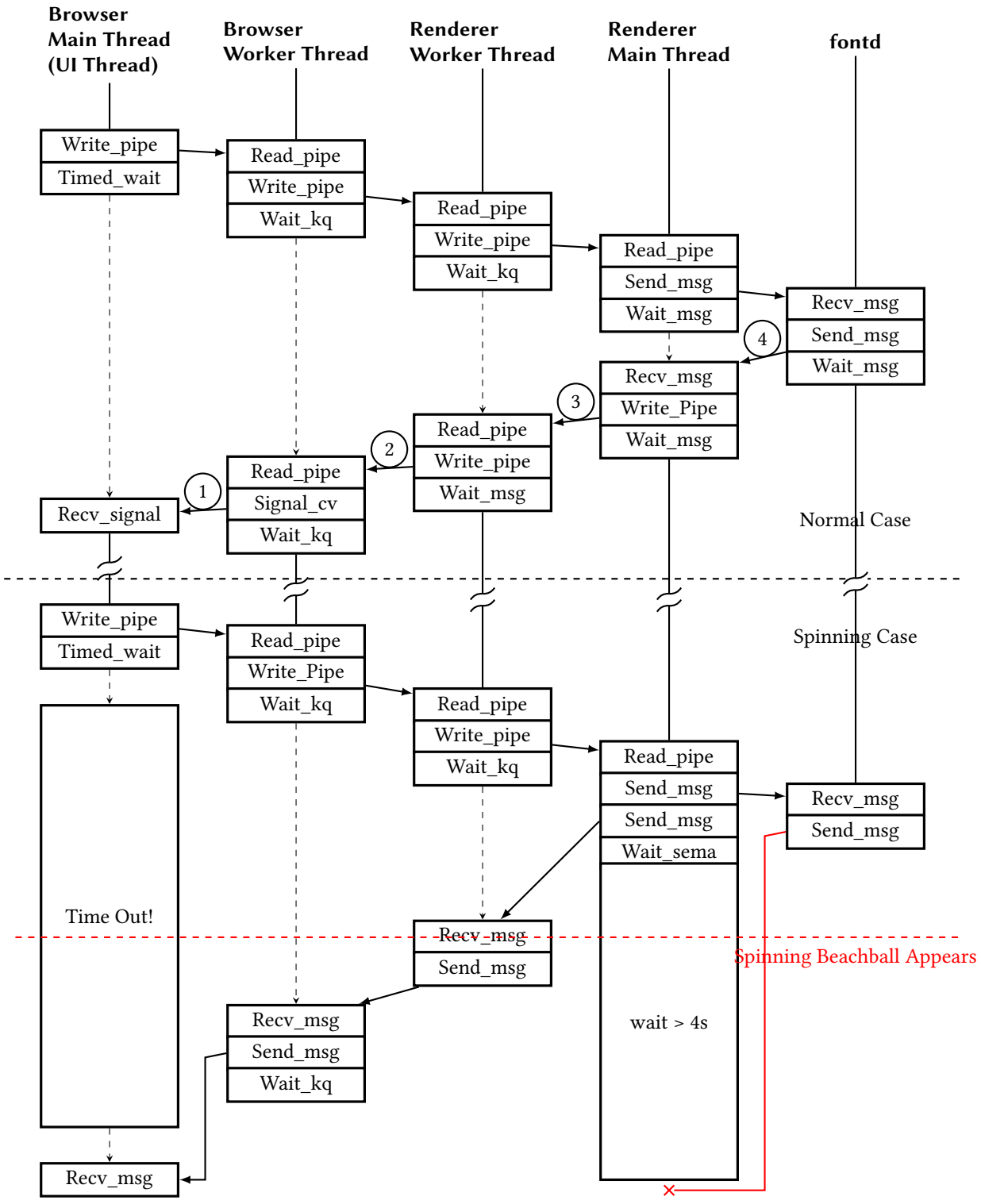

Figure 2. Chromium case study.

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.

2.3 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 Dependency Semantics

3.1 Tracing Events

The log that Argus collects consits of a squence of tracing events, falling into three categories: semantics tracing events, dependency tracing events and boundary tracing events. Semanctics events include system calls, backtraces, and instruction logs. These events help diagnosis and are also used by Argus to find similar events in its analysis. Boundary events are recorded within a single thread where we may eventually place an execution segment boundary. These events tend to be the calls to and returns from wait operations. Dependency events are recorded whenever one thread comunnicates with another,such that we will eventually create causual links—for example, a thread calls *mach_msg_send* and delivers message to anther thread which calls *mach_msg_receive*. These two dependency events will eventually be used to create a link in the graph. These categories are not disjoint. For instance, most events are logged together with the call stacks in which they occur. Similarly, the return from aforementioned `mach_msg_receive` is both a dependency event and a boundary event as it begins a new execution segment.

3.2 Dependency Patterns

We encountered several instances of runtime event dependencies, correct or incorrect, between thread contexts. We present several generalizable cases below.

P1: Signal or interrupt handling Sometimes, a signal or interrupt handler happens to run within a thread's context, for example, a timer interrupt. As discussed in the previous section, Argus logs wait and wake-up events inside the kernel to increase the completeness of tracing, so it may log events that occur in the call to signal or interrupt handler.

We identify the start and end of the signal-handling code to splice it away from the containing context, since it is usually unrelated.

```
Thread in Notes application:
stat()
interrupt() {
  state = save_contex
  lapic_interrupt(intr, state)
}
wait(lock_mutex)
```

P2: Kernel takes over context As part of a thread context switch, an execution context may enter kernel space. As shown below, the code will enter kernel scheduling by calling `sched_timeshare_consider_maintenance`, which in turn wakes up another `kernel_task` thread. Again, such a wake-up does not reflect the application's intent, and should be filtered out from the containing context.

To detect this case, we filter wakeups from the kernel timer, interrupt handler, or kernel shared time maintenance. Such cases represent spurious dependencies. However, sometimes when a worker thread wakes up another worker thread, this can represent a true dependency. The distinguishing feature is whether a synchronization primitive (including shared memory) is used.

```
thread_invoke(self, new_thread, reason) {
// thread switch, kernel space
  sched_timeshare_consider_maintenance() {
    thread_wakeup(sched_timeshare_miantenance_continue);
  }
  ast_context(new_thread);
}
```

P3: 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
  }
}
```

P4: 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)
```

P5: 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;
```

P6: 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()
```

P7: Mach message mismatch When RPC-style interprocess communication is used, most systems would use the same thread to send the call request and receive the return. They would also use one recipient thread to process the call. However, in MacOS, the thread sending the call request may

be different from the one receiving the return, and multiple threads may be used in the recipient thread to handle the request.

Fortunately, the messages involved typically carry metadata such as the `reply_port` shown in the following diagram, so Argus can easily link the corresponding events together.

```
SCIM calls XPC_connection_send_msg
t1_scim   t2_chromium       t3_chromium     t4_scim
|         |                 |               |
mach_msg_send(remote_port, reply_port)      |
          |                 |               |
          dispatch_mach_msg_receive(remote_port, reply_port)
                            |               |
                            mach_msg_send(reply_port)
                                            |
                                 mach_msg_receive(reply_port)
```

P8: 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.
```

3.3 Discussion

These patterns reinforce our insights described in Section 1. First, as illustrated by patterns P3, P6, and P7, batch processing is frequently used to lump work on behalf of different requests together, causing inaccurate edges that do not reflect causality in the event graph. Some amount of manual schema is needed, but annotating all applications and daemons upfront to expose batch processing would be strenuous and error-prone. We believe Argus's interactive approach represents a better design tradeoff. Second, as illustrated by P3, P4, and P5, systems use data flags for causal linkage. They do so in both multithreaded and single-threaded environments. Such data dependencies can be crucial for diagnosing performance issues – the spinning cursor's display itself uses shared memory flags as shown in P5. Third, as P1, P2, and P5 show, wake-ups do not necessarily reflect causal linkage. Lastly, systems may deviate from well-established patterns such as in P7 or create their custom primitives such as in P6.

4 Implementation

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

4.1 Instrumentation

Like Detour [13], 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 illustrated the root causes for two cases which triggered the spinning wait cursor. As we have illustrated the Chromium case in Section2, we will not present it here again.

The spinning wait cursor 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, since spindump shares the same triggering condition. Spindump is launched by the message from WindowServer, after WindowServer receives a message from the NSEvent thread of the targeted application.

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 targeted App fetches CoreGraphics events from WindowServer, converts and creates NSEvents for the main thread. If the main thread is not spinning with the cleared "`is_mainthread_spinning`", "`dispatch_to_mainthread`" is set and a timer is armed. If the main thread processes the next event before the timer fires, nothing happens and the timer gets re-armed. Otherwise, NSEvent thread sends a message to WindowServer from the timer, and WindowServer notifies the CoreGraphics to draw a spinning cursor over the application window.

5.1 System Preferences spin

System Preferences is the application in MacOS for users to modify various settings. Displays in the Panel allows user to rearrange the position of displays, but it does not support the disabling of a montor online. DisableMonitor is an application used to complete the function, easily disabling/enabling monitors withou unplug them. Surprisingly, the operation of DisableMonitor exposed a performance bug in System Preferences. If we disable an external monitor and arrange them afterward, the window System Preferences freezes for seconds.

We enabled Argus on the background and collect the data by normally arranging the displays and reapeating the spinning sequence. With 2 minutes recording, we get the tracing log of size 132MB, which contains 428,785 vertexes and 320,554 edges.

It is not hard to tell the spinning node in the UI thread with our tool. However, to find the normal node corresponding to the spinning node is not straightforward as in the previous case. The execution segment includes two types of events: "`mach_msg`" and "`thread_switch`". Both of them are used to waiting for the data available ping. The semantics of the execution segment is not descriptive enough to identify the normal nodes in the same operation stage.

In the case we detected the intensive timeout in the node, our search algorithm identify the corresponding normal node by searching the similarity of their proceding nodes. We find out the path of the normal path. Our lightweight callstacks were used to verify the correctness of the findings.

The normal node showed it proceeded to *displayReconfigured* after received the message with id 29675. The spinning node fell into the "`thread_switch`" after receiving the message with the same id, and end up to send message for the available datagram ping with *CGSSnarfAndDispatchDatagrams* again. The proceding nodes before them sent message to WindowServer for *activeDisplayNotificationHandler*.

With the result, we initiate the concrete debugging by filling the debugging script with the APIs reported. We set the method *activeDisplayNotificationHandler* as a beakpoint where the script begins debugging. *displayReconfigured* and *CGSSnarfAndDispatchDatagrams* are recorded to indicate the end of debugging for the normal case and spinning case respectively.

Our debugging scripts ran within the confined range for both the noraml and the spinning execution. By diff the two logs, it is easy for the user to notice the differen braches in *display_notify_proc*, which is resulted from its prameter standing for the datagram type.

We make use of the diassembly tool, and reveals the story in the background. Datagrams from the WindowServer makes applications to handle notifications. The datagram causes difference are used to finish display reconfiguration for System Preference. However, in the spinning case the reconfiguration got initiated but not completed. The main thread leveraged thread_switch to wait for the flollowing datagram and resulted in a freeze. As a conclusion, the handler display_notify_proc is not appropriately implemented.

5.2 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 to carry out the root causes analysis, including TextEdit, Microsoft Word, SublimeText, TextMate, CotEditor and TeXStudio.

Argus analyzed the tracing data and reported the spinning node for user to inspect. All the three cases are fell into the non-blocking categories. Our lightweight callstack tells the root cause directly.

For the selection in TextEdit, the size of the tracing log is 144MB. The graph generated from the log contains 170,054 edges and 101,212 vertexes. The spinning node illustrates that 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 generated a graph of 158,066 edges and 126,663 vertexes. It 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:]*.

We collected 214MB data for the paste in TextEdit, and generated 2,312,112 edges and 1,060,273 vertexes. TextEdit is busy with *DDLookupTableRefLookupCurrentWord_block_invoke_2*, *_RTFGetToken* and *platform_memmove*. All of them called by *-[NSTextView(NSPasteboard) _readSelectionFromPasteboard:types:]*

5.3 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*.

5.4 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*.

5.5 TextMate

Spinning cursor appears when a large amount of context get pasted into a file opend 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*.

5.6 CotEditor

As is TextMate, CotEditor does not trigger spinning beachball when a large amount of context is copied in the Application. Any insertion in a large file results in a spinning beachball. We collected 318MB trace data and generated a graph with 374,577 edges and 245,638 vertexes. The main thread is busy processing *-[NSBigMutableString characterAtIndex:]* and *CFStorageGetValueAtIndex*, The processing happens once the return key is pressed.

5.7 TeXStudio

TeXStudio is another case that causes beachball only when a large amount of context is pasted. The size of the trace is 287MB. The graph contains 349,557 edges and 259,918 vertexes. The spinning node is busy with *QEditor::insertFromMimeData(QMimeData const*)*, which repeatedly calls *match(QNFAMatchContext*, QChar const*, int, QNFAMatchNotifier)*.

5.8 Notes

The Notes spinning cursor is quite annoying in the fact that one wrote an note and stored it, and later when she tried to lauch the Notes app, the spinning beachball appeared for a long while. It prevents the user from accessing the note for over one hour in our experiment.

With Argus we stop the recording in 3 minutes after the spinning cursor shows up, and collect the trace of 229M. The generated graph has 198228 edges and 144051 notes. It reports within one second by searching the graph that the node which peforms the block submited by *NSDetectScrollDevicesThenInvokeOnMainQueue*.

5.9 Installer

Installer throws out spinning wait cursor in a situation that confuses the users. While we were installing `jdk-7u80-macosx-x64`, the installer asked for privileged permissions. Once we move the cursor, it starts spinning whenever the original application is selected, potentially confusing the user.

Argus collected tracing events in log of 550M for Installer, The generated graph contains 392199 edges and 294856 vertexes. Argus generate the graph in 168 seconds, and find the spinning node is blocked in the *[IFRunnerProxy requestKeyForRights:askUser:]*, which sent a synchronous XPC to daemon *authd* with *xpc_connection_send_messag_with_reply_sync*.

5.10 Sequel Pro

Sequel Pro is a graphical App for database client end. It can be connect to database server via different ways, socket, ssh and direct connection. When one of the window in Sequel Pro tries to reconnect with ssh, a spinning cursor appears and blocks the whole application, which results in the user can not save work in other windows if the reconnection failed.

We got 2G tracing data and generated graph containing 3,650,832 edges and 2,412,236 vertexes. The main thread is blocked on rw_lock. With the path slicing, we discovered the blocking is eventually caused by waiting on ssh connection, as is shown in Figure **??**.

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 1. Score From iBench

We show the five runs of both cases in Table1. 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 2.

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

Table 2. Chromium benchmark: telemetry

As shown in Table 2, 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 [22] 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 [15] 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 [6, 7, 10] 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. [11, 21] leverage the user logs and call stacks to identify the performance anomaly. [8, 9, 17, 19]apply the machine learning method to identify the unusual event sequence as an anomaly. [20] generates the wait and waken graph from sampled call stacks to stidy 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. (2008). https://www.chromium.org/

[2] 2012. (2012). https://bugs.chromium.org/p/chromium/issues/detail?id=121917

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

[4] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. 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*. 307–320.

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

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

[7] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. 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)*. 217–231.

[8] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control.. In *OSDI*, Vol. 4. 16–16.

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

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

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

[12] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2012. A file is not a file: understanding the I/O behavior of Apple desktop applications. *ACM Transactions on Computer Systems (TOCS)* 30, 3 (2012), 10.

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

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

[15] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. 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*. 107–120.

[16] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. 2006. Pip: Detecting the Unexpected in Distributed Systems.. In *NSDI*, Vol. 6. 9–9.

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

[18] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad Hoc Synchronization Considered Harmful.. In *OSDI*, Vol. 10. 163–176.

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

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

[21] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. 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)*. 293–306.

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

[23] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. 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)*. 603–618.