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

Abstract

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

1. Introduction

Today's web and desktop applications are predominantly parallel or distributed, making performance issues in them extremely difficult to diagnose because the handling of an external request is often spread across many threads, processes, and asynchronous contexts instead of in one sequential execution segment [14]. 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 [8, 25, 21, 16, 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 [4]. 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 [24]. 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 [24, 17]. 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 [20] 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.

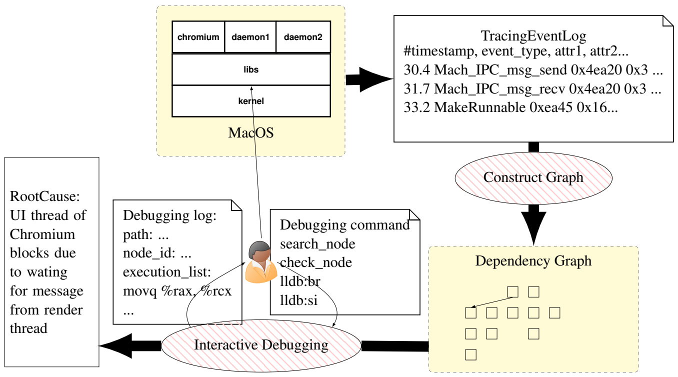
We believe that, without detailed understanding of application semantics, request graphs computed by causal tracing are *inherently* imprecise and both over- and under-approximate reality. Although developer annotations can help improve precision [7, 18], modern applications use more and more third-party 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 [7].

Moreover, Argus enables users to dynamically control the granularity of tracing using a number of intuitive primitives. The system begins by using always-on, lightweight, system-wide tracing. When a user observes a performance issue (e.g., a spinning cursor), she can inspect the current graph Argus computes, configure Argus to perform finer-grained tracing (e.g., logging call stacks and instruction streams) for events she deems relevant, and trigger the issue again to construct a more detailed graph for diagnosis. Argus also supports interactive search over the event graphs that contain both normal and buggy executions for diagnosis.

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

We evaluated Argus on 11 real-world, open spinning-cursor issues in widely used applications such as the Chromium browser engine and MacOS System Preferences, Installer, and Notes. The root causes of all 11 issues were previously unknown to us and, to a large extent, the public. Our results show that Argus is effective: it helped us find all root causes



```
TracingEventLog
#timestamp, event_type, attr1, attr2...
30.4 Mach_IPC_msg_send 0x4ea20 0x3 ...
31.7 Mach_IPC_msg_recv 0x4ea20 0x3 ...
33.2 MakeRunnable 0xea45 0x16...
```

```
Debugging log:
path: ...
node_id: ...
execution_list:
movq %rax, %rcx
...
```

```
Debugging command
search_node
check_node
lldb:br
lldb:si
```

RootCause: UI thread of Chromium blocks due to wating for message from render thread

Figure 1: Argus Work Flow

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 describe our event graph from causual tracing, and Section 4 describes our tracing implementation and tools for user interaction. 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

In this section, we describe the steps a user takes to investigate a performance anomaly with Argus. 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 as listed in Table 1 (§3.1). Whenever a user detects a performance issue such as a spinning cursor, she runs "`Argus debug`" to enter the diagnosis phase.

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

§3.1. Next, we describe the graph operations Argus performs leading to a diagnosis.

2.2. Diagnosis with Graph

Algorithm 1 Main Argus algorithm.

Require: Program to run + buggy test case + (optional) similar baseline test case
Ensure: Sequence of UI events triggered the performance problem + Nodes from Event Graph

```
 1: function ARGUS MAIN
 2:     $TracingEvents \leftarrow$ Run program under Argus, trigger baseline case
        (if possible) and buggy test case
 3:     $HeuristicsSet \leftarrow \{default\ \ heuristics\}$
 4:     $EventGraph \leftarrow$ ComputeGraph($HeuristicsSet$, $TracingEvents$)
 5:     $SpinningNode \leftarrow$ search $EventGraph$
 6:     if $SpinningNode == CPUbusyexecution(livelock)$ then
 7:         run backward traversal of main thread until UI event found
 8:     else
 9:         $N_0 \leftarrow SpinningNode$
10:         $N_1 \leftarrow$ FindSimilarNode($EventGraph$, $N_0$)
11:         call AssistedGraphDiff($EventGraph$, $N_0$, $N_1$)
12:         if new_hueristics were added then
13:             go to 4
14:         end if
15:     end if
16:     Return set of UI events and root cause related Nodes
17: end function
```

Algorithm 1 shows how Argus investigates a performance anomaly for users, given a buggy test case and (potentially) a baseline test case for comparison purposes. We assume the user has a program to execute and a buggy test case that can be reproduced eventually; the tracing log collected by Argus would contain a similar event processing normally, or the user can provide a nomarlly executed test case for comparison purpose, listed in line 2. Argus initializes this debugging phase by constructing an event graph from all logged events up to user-specified duration in line 4. The detail of algorithm ComputeGraph is presented in Algorithm 4(§3.2).

The exact searches and queries performed on the graph depend on the bug under investigation. Consider a common performance bug on MacOS, the *spinning cursor*, which indicates the current application's main thread has not processed any UI events for over two seconds. Argus queries the event graph to find the ongoing event in the application's main thread concurrent to the display of the spinning cursor shown in line 5. Upon examing what the main thread is actually doing, the user may encounter three potential cases. First, the thread may be busy performing lengthy CPU operations (which take longer than two seconds). Second, the thread may be in a blocking wait. Third, the main thread may be in a yield loop, which is highly indicative it is waiting on a data flag (e.g., "while(!done) thread_switch();"). Line 7 to 14 is to track events and synchronization primitives throughout the system to figure out the root cause, in terms of user input events and bug in code, for the three cases.

For the first case, shown in line 7, Argus will examine the stack trace and find a sequence of events that led to the current CPU processing. If more specific tracing is required, the user can rerun the program with more heavyweight instrumentation enabled for any portion of the code's execution, gathering a precise sequence of calls or even instructions executed.

On the contrary, Argus locates the node N_1 for the rest cases, either a blocking wait or a yield loop, in our graph when the spinning beachball is triggered, as is shown between line 9 and line 14. Then Argus applies Algorithm 2 to locate a node N_0 in our graph that corresponds to the baseline version of the hanging node N_1.

For the second case of block waiting, Argus takes the assumption there usually exits a chain of blocking thread and the initial blocking thread is the most valuable for debugging. Consequentally, Argus diffs the subgraphs between the blocking version and the baseline version to determine their difference with Algorithm 3.

In the third case, 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();"). To discover a data flag, the user re-runs the application with Argus to collect instruction traces of the concurrent event in both the normal and spinning cases and detects where the control flow diverges. She then reruns the application with Argus to collect register values for the basic blocks before the divergence and uncovers the address of the data flag. She then configures Argus to log accesses to the flag during system-wide tracing. Finally, she can recursively apply Argus to further diagnose "the culprit of the culprit".

Based on our results and experience, the first case is the most common, but the second and third represent more severe bugs. Long-running CPU operations tend to be more straightforward to diagnose with existing tools. Blocking or yielding cases involve multiple threads and processes, and are extremely hard to understand and fix even for the application's original developers. Therefore, issues remain unaddressed for years and significantly impact the user experience. Algorithm 1 is semi-automated but can integrate user input at each stage to leverage hypotheses or expert knowledge as to why a hang may occur.

2.3. Argus Assisting Algorithm

We mentioned previously in §2.2 that Argus explores the event graph semi-automatically to narrow in on relevant nodes in event graph for diagnosis. Algorithms that require no user intervention are described in this section.

Find Similar Node Algorithm FindSimilarNode in Algorithm 2 is meant to find nodes which have the same high level semantics but different execution results compared to the spinning node. For example, both of them are waiting on locks, but one returns successfully and the other not. Argus defines a subset of events that peserve semantics, which contains System_call, Back_trace, NSApp_event, Mach_message,

Algorithm 2 Argus Find similar node algorithm.

```
Require: EventTypesSet + SpinningNode + Graph
Ensure: SimilarNodeSet

function FindSimilarNode
    $thread \leftarrow$ thread of $Node_blocking$
    for $Node_i$ in $thread$ other than $Node_blocking$ do
        $iter_i \leftarrow$ iterator of first event in $Node_i$
        $iter_b \leftarrow$ iterator of first event in $Node_blocking$
        while $iter_b \neq$ end and $iter_i \neq$ end do
            while $iter_i \neq$ end and $Event(iter_i) \notin EventTypesSet$ do
                $iter_i$++
            end while
            while $iter_b \neq$ end and $Event(iter_b) \notin EventTypesSet$ do
                $iter_b$++
            end while
            if $iter_b \neq$ end and $iter_i \neq$ end then
                $if_eq \leftarrow$ CompareEvent($Event(iter_b)$, $Event(iter_i)$)
                if $if_eq ==$ False then
                    Break
                end if
            end if
        end while
        if $iter_b ==$ end and $iter_i ==$ end then
            if wresult of $Node_i) \neq$ wresult of $Node_blocking$ or timespan of $Node_blocking$ - timespan of $Node_i \geq$ threshhold then
                Put Node into $SimilarNodeSet$
            end if
        end if
    end for
end function
```

Algorithm 3 Argus Assisted graph diff algorithm.

```
Require: SpinningNode + BaselineNode + Graph
Ensure: Possible root cause of thread blocking

function AssistedGraphDiff
    $ResultSet \leftarrow \{\}$
    $SlicingPath \leftarrow$ BackwardSlicing on $BaseLineNode$
    $T_spinning \leftarrow$ timestamp of spinning beachball for $SpinningNode$
    for $thread \leftarrow$ thread of $Node_i$ in $SlicingPath$ do
        $T_normal \leftarrow$ timestamp of last event in $Node_i$
        $Node_blocking \leftarrow thread$ blocks during ($T_normal$, $T_spinning$)
        if $Nodeblocking \neq NULL$ then
            put $Node_blocking$ into $ResultSet$
        end if
        if $UIEvents$ in Main UI thread then
            put $UIEvent$ into $ResultSet$
        end if
    end for
    Return $ResultSet$
end function
```

Wait, Dispatch_enqueue, Dispatch_invoke, Runloop_submit, Runloop_invoke, Share_flag_read and Share_flag_write. The comparison of events depends on event type and its category in Table 1. For connection events, Argus compares their peers. For semantics Events, Argus compares the content of the event attributes, for example, the system call number. The wait results of wait event usually differentiate the normal execution and buggy case. Argus differentiate the cases with wait results by default. Depending on the report detail of the spinning node, user can change the comparing metrics or request Argus to compare proceeding nodes to narrow down the resultset reported.

Graph Diff Algorithm Argus tries to figure out the root cause for the second case(§2.2) with Algorithm 3. Argus assumes that the blocking in the main thread usually caused by the unresponsive of another thread. In the algorithm, with the input of the blocking node in main thread, Argus carries out a backward path slicing from the node from the baseline case in the graph. The path consists of nodes from mutiple threads. Argus traverses every thread and confines the search within the time interval from the base line node in the thread and the timestamp when the blocking node appears in main thread. Among them Argus expects to find a potential blocking in some threads and reports them. At the same time Argus also collects the UIEvents into return set.

2.4. 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 [3]. 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 [5, 24, 6, 10] because they handle a fairly limited set of patterns. Tools that require manual schema [7, 18], would be prohibitive because developers would have to provide schema for all involved applications and daemons.

Next she ran Argus to find the spinning node in the main thread of the browser process. Argus returned a `Wait` event on condition variable with timeout that blocked the main thread for a few seconds. Thus Argus compares the spinning node to a similar one in normal case where the `Wait` was signaled quickly with Algorithm 2. Argus reported three, and confirmed with the user which one she wanted.

Argus then found the normal-case wake-up path 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. The worker thread is woken up by the `renderer` main thread, which in turn woken by fontd, the font service daemon. Argus further compared the wake-up path with the spinning case with the Algorithm 3, and returned the `wait` event on semaphore in the `renderer` main thread, the culprit that delayed waking up the browser main thread over 4 seconds.

What caused the wait in the `renderer` main thread though? She thus continued diagnosis and recursively applied Argus to the wait in `renderer`, and got the wake-up path. The culprit that delayed the semaphore was the timeouts in the browser's main thread. At this point, a circular wait formed. To understand what exactly happens in the situation, she inspected the full call stacks by Argus scripts, taking the reported nodes from the renderer and the browser as input. Inspection reveals that the `renderer` requested the browser's help to render Javascript and waited for reply with semaphore. 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 `wait` on cv 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.

The finding was verified with chromium source code. Shortening the timeout interval in the main browser thread proportionally shortens the waiting of the main render thread on processsing the javascript. Skipping certain javascripts processing in the renderer thread cuts down the success rate of spinning case reproducing.

3. Argus Graph Computing

3.1. Event Graph

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

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

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

Event Type	Event Categories
System_call	Semantics
Back_trace	Semantics
NSApp_event[1]	Semantics
Wait	Semantics, Boundary
Interrupts	Boundary
Sharetime_maintenance[2]	Boundary
Dispatch_invoke[3]	Boundary
Runloop_invoke	Boundary
Mach_message	Boundary, Connection
Wake_up	Connection
Timer	Connection
Dispatch_enqueue	Connection
Runloop_submit	Connection
Share_flag_read	Connection
Share_flag_write	Connection

1. User input events dispatched to the Application.
2. Kernel invoked a routine to update timeshare quota.
3. Invoke callback function for works from dispatch queue.

Table 1: Event Type Categories.

3.2. Graph Computing Algorithm

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

A node is a sequence of events derived from the execution of a task in a thread. As shown in Algorithm 4, Argus checks events per thread and applies heuristics when a boundary event is checked. Argus provides five default heuristics. Four of them share the idea of general cuasual tracing, one is used to work around unknown programming paradigms, and more are expected from users to improve graphs. We discuss them in below.

As listed in (§3.3), event sequences for interrupt processing and kernel maintainance are removed from threads in line 7. The second heuristics in line 9 treats a wait event as an end of a node. An wait event usually indicates a thread switches to other tasks, but it is not always true considering a thread may park due to low thread priority. One of the examples in MacOS is the pause of worker threads draining a low priority diapatch queue. Dispatch queues are FIFO queues to which an application can submit tasks in the form of block objects. Wait events should not divide the block objects, otherwise it may result in a missing connection. To save the intergrity of block objects, Argus keeps a counter `callout_level` to mark if the current event is inside a block object. Only the begin and end of Dispatch_invoke are served as the boundary for the node, as is shown in line 14. The hueristics also applies to runloop. Runloop is an event processing loop that used to schedule work and coordinate the receipt of incoming events. The begin and end of the work invocation are served as boundaries. The last default heuristics in line 22 is used to work around the difficulty of exploiting batch processing programming paradigms completely, as listed in follows (§3.3). Argus makes use of mach messages to avoid clustering multiple tasks due to unknown batch processing. It defines `IPC_peer_set` to compute the set of mach message receivers/senders for every node. For every mach message, the algorithm checks whether its peer process exists in `IPC_peer_set`, and adds the message into the current node if the condition is true or the set is empty. Otherwise it adds an ending boundary for current node and constructs a new node beginning with the mach_message event.

Edges connect nodes, both intra-thread and inter-thread, with connection events. Argus walks through event type in connection category to apply heuritics as follows. First, the return from a wait operation causally depends on the wake-up operation. An edge is defined from the wake-up event to the first event after the wait returns. Argus also add a weak edges from the wait event to the wake-up event in line 26. Mach message is the core of ipc mechanism implemented in kernel, upon which higher level RPC are built. Argus connects the sender and receiver of a mach message. For messages that expect a reply, Argus also connects the receiver of the original message and the sender of the reply message in line 29. As discussed above, dispatch queue and runloop are popular batch processing programming paradigms in MacOS, Argus connects submissions of a task and executions of the task, which are listed in line 34 and line 36 respectively. Similarly, Argus adds edge from a timer armed event to its triggered event in line 38. Shared flag can either be traced with binary instrument, such as `need_display` flag for CoreAnimation in (§3.3), or with breakpoint watcher command line tool provided by Argus. Edges from the flag written to its read are added from line 40. Since share variables are hard to completely exploit, and the causality can be more complicated than the writer-reader pattern, user interaction is expected.

Finally, a event graph returns and is subject to the improvement with more user input heuristics.

3.3. Inherent Inaccuracy

However, to construct an accurate and complete dependency graph is difficult, if not impossible. The graph is inherently inaccurate. That one thread wakes up the other thread does not always stand for a causility between them. In implementation,

Algorithm 4 Argus Compute Graph algorithm.

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

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

Argus filters out some of the definitive noise in the following types.

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

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

3.3.1. Over Connections

Over connections usually occur when intra-thread boundaries are missing due to unknown batch processing programming paradigms. We list the example patterns we found below.

Dispatch message batching While traditional causual tracing assumes the entire execution of a callback function is on behalf of one request, we found some daemon implements its service loop inside the callback function and creates false dependencies. In the code snippet in Figure 2 from the `fontd` daemon, function `dispatch_execute` 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.

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. Any capplication 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:
//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 2: Dispatch message batching

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 differnt processed in its event loop, as shown in Figure 3. This batch processing artificially makes many events appear dependent. We split the execution segments to maintain the independence of the events.

3.3.2. Under Connections

On the other hand, under connections mostly result from missing data dependencies. Data dependencies inter/intra threads are usually hard to fully exploit in the initial pass of graph computing.

Data dependency in event processing The code in Figure 3also 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. Argus uses watch point registers to capture events on `_gOutMsg` and establish the causal link between the handling of the previous request and the send of the reply.

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

Figure 3: Batching in event processing

CoreAnimation shared flags As shown in Figure 4, 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.

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

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

Figure 4: CoreAnimation shared flag

Spinning cursor shared flag As shown in Figure 5, 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 the 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.

3.4. User Interactions

In addition to batching processing and data dependency, the spurious edge introduced by mutex lock is also a challenge in debugging. The synchronization on mutex lock reflects one thread wakes up the other thread. The scenario can either be a producer-comsumer problem or merely mutualy exclusion. Making the graph completely sound without user interaction is almost impossible given the essential attribite of commericial operating system as a grey box.

As we mentioned above, to figure out all over connections and under connections before hand is almost impossible. Instead, users can find out the descrepency in event graph after the initial graph computing. For example, we noticed that two unrelated applications connects to one node in the graph,

```
//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 5: Spinning Cursor Shared Flags

which leads to the manifestation of kernel thread batch processing on timers. Like other causality tracing approaches, Argus is a general framework for MacOS and tested on limited programing paradigms. Allowing user interaction makes its graph more practical and useful case by case.

Users can gradually inject their knowledge to improve the graph in two ways. They can either add add heuristics to the graph computing algorithm in Algorithm 4, or binary instrument images with the APIs provided by Argus to expand the boundary or connection event categories.

4. Implementation

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

4.1. Event Tracing

Current MacOS systems support a system-wide tracing infrastructure built by Apple [1]. 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 and avoid filling up the disk with a file-backed ring buffer. Subject to configurarion, it allows at most 2GB of data per log, which corresponds to approximately 18,560,187 events (about 5 minute with normal operations).

The default tracing points in MacOS provide too limited information to apply causual tracing. As a result, we both patch source code of kernel [2] and binary instrument libraries to gether more tracing data. In Argus, we patched the kernel with 1193 lines of code, and instrumented the libraries including: libsystem_kernel.dylib, libdispatch.dylib, libpthread.dylib, CoreFoundation, CoreGraphics, HIToolbox, AppKit and QuartzCore, with our binary instrumentation.

4.2. Instrumentation

Most libraries as well as many of the applications used day-to-day are closed-source in MacOS. To add tracing points to such code, 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.

Like Detour [15], we use static analysis to decide which instrumentation to perform, and then enact this instrumentation at runtime. Firstly, users find a location of interest in the image related to a specific event by searching a sequence of instructions. Then the users replace a call instruction to invokes a trampoline target function, in which we overwrite the victimed instructions and produce tracing data with API from Apple. All of the trampoline functions are grouped into a new image, as well as an initialization function which carries out the drop-in replacement. Then command tools from Argus helps to configure the image with the following steps: (1)re-export all symbols from the original image so that the original code can be called Like an shared library; (2)replace the original image with the new one by renaming them to ensure the modifications are properly loaded; (3)invoke the initialization function externally through `dispatch_once` during the loading.

4.3. User Interaction

Tracing Custom Primitives As described in (§3.4), under-connection due to the missing share data dependency requires users'interaction. Argus provides a command line tool which sets the watch point registers to record share_flag_write and share_flag_read events in ad-hoc manner. This is one way users can easily collect the tracing events, and the more tech-savvy users can also instrument the binary. The tool takes the process id, path to image where the variable is defined and the symbol of the variable as input. We show the simple example how a user ask Argus to trace _gOutMsgPending in the following command.

```
./bp_watch pidofWindowServer Path/to/CoreGraphics\
  _gOutMsgPending
```

Argus hooks the watch point break handler in CoreFoundation to make sure that it is loaded correctly into the address space of our target application. The handler invokes the event tracing API from Apple to record the value of the shared variable and the operation type: read or write.

Capturing Instructions for Diagnosis Since the output of Argus is the nodes and user input events suspected to cause the busy spinning, more detail may required to verify and fix the bug. After the offline analysis on the graph, Argus provides tools for user to exact the backtrace events from the output and generate a script for conditional debugging.

The debugging scripts go through the instructions of apps and frameworks step by step to capture the parameters tained by user inputs. At each beginning of a function call, the script records a full callstack for it. Considering the overhead and usefullness, it steps over and only record the return value of APIs from libraries with a filename extension .dylib.

The supplementary information are subject to the users review to pinpoint the root cause of spinning beachball on MacOS.

4.4. Limitations

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

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

5. Case Studies

In this section, we presents how we apply Argus to examine spinning beachballs in 11 popular applications. We begin with how we found out the design of spinning beachball in Apple with event graph. Then we describe the cases in XXXXXX order or in XXXXcategories. As discussed before in (§2), the chromium case will not be present in this section .

5.1. Spinning Node detection

Spinning beachball is a painful sight for Mac users, signifying that the application is non-responsive. It usually remains for minutes at a time, leaving the user at a loss and unable to do anything productive. Under this situation, a hang reporting tool, spindump, usually invoked by root to sample for debugging.

To figure out the spinning node in the main thread, we turned to the event graph, and sliced path backward from the launch of spindump, considering it is another indicator of non-responsive in app. The path showed it was launched after receiving a message from WindowServer, which in turn received a message from the NSEvent thread of the freezing app. The callstacks attached to the messages further revealed NSEvent thread per process fetches CoreGraphics events from WindowServer, converts and creates NSAppEvent for the main thread. If the main thread is not spinning ,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 via the API "`CGSConnectionSetSpinning`"from the timer handler, and WindowServer notifies the CoreGraphics to draw a spinning wait cursor over the application window.

Moreover, with the callstack symbols found above as input, our detail debugging script detected two share variables "`is_mainthread_spinning`" and "`dispatch_to_mainthread`", are used to note the main thread status. With these discovery, we can make use of either the API or the shared variable to identiy the spinning node in the mian thread and the UI event cause the hanging.

5.2. Blocking Cases

In this section, we discuss the cases where the spinning node is blocking on Wait event.

- sequelpro <- ssh connection
- textstudio <- synchronization once the file get touched
- installer <- user input blocked

5.3. Yielding Loop Cases

- systempreferences <- similar node, branches on a shared variable, add monitor on the variable
  - find similar node: compare proceeding nodes, differential metrics
  - shared variable and script to find out why they have different branches

5.4. CPU Busy Cases

- a table on case description, corresponding root cause reported by argus
  - notes
  - textedit
  - mswords
  - sublimetext
  - textmate
  - coteditor

5.5. Verifications

6. Performance Evaluation

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

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

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

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

Table 2: Score From iBench

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

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

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

Table 3: Chromium benchmark: telemetry

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

7. Related Work

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

Event tracing. Magpie [7] 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 [24] 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 [17] 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 [12, 8, 9] 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 [5] 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. [13, 23] leverage the user logs and call stacks to identify the performance anomaly. [10, 19, 21, 11] apply the machine learning method to identify the unusual event sequence as an anomaly. [22] generates the wait and waken graph from sampled call stacks to study a case of performance anomaly.

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

8. Conclusion

Our key insight in this paper is that causal tracing is inherently imprecise. We have reported patterns we observed that pose big precision challenges to causal tracing, and built Argus, a practical system for effectively debugging performance issues in MacOS applications despite the imprecision of causal tracing. To do so, it lets a user provide domain knowledge interactively on demand. Our results show that Argus effectively helped us locate all root causes of the issues, including a bug in Chromium, and incurred only 1% CPU overhead in its system-wide tracing.

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.