Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing

Lingmei Weng[†] Peng Huang[‡] Jason Nieh[†] Junfeng Yang[†]

Columbia University[†] *Johns Hopkins University*[‡]

Abstract

Modern desktop applications involve many asynchronous, concurrent interactions that make performance issues difficult to diagnose. Although prior work has used causal tracing for debugging performance issues in distributed systems, we find that these techniques suffer from high inaccuracies for desktop applications. We present Argus, a fast, effective causal tracing tool for debugging performance anomalies in desktop applications. Argus introduces a novel notion of strong and weak edges to explicitly model and annotate trace graph ambiguities, a new beam-search-based diagnosis algorithm to select the most likely causal paths in the presence of ambiguities, and a new way to compare causal paths across normal and abnormal executions. We have implemented Argus across multiple versions of macOS and evaluated it on 12 infamous spinning pinwheel issues in popular macOS applications. Argus diagnosed the root causes for all issues, 10 of which were previously unknown, some of which have been open for several years. Argus incurs less than 5% CPU overhead when its system-wide tracing is enabled, making always-on tracing feasible.

1 Introduction

Diagnosing performance anomalies is an essential need for all kinds of software. For modern desktop applications, performance diagnosis can be very difficult. Such applications are often built with assorted frameworks and libraries. For responsiveness, they divide handling of user interface (UI) events into many small execution segments [30] that run concurrently on multi-core hardware. For instance, macOS applications handle UI events by sending messages to delegate objects that contain code to react to these events asynchronously. The messages are generated by the closed-source Cocoa framework [11], which in turn interacts with the operating system (OS), daemons, and other libraries. The asynchronous, predominantly concurrent interactions obscure the true cause of a performance anomaly.

Traditional debugging and profiling tools are not well suited to troubleshoot performance issues in desktop applications. macOS tools such as `spindump` [10] and `lldb` [45] allow users to analyze a buggy process' stack traces. Profilers like Gprof [28], `perf` [5], and macOS Instruments [12] mainly analyze what functions take the most time. None of these tools provide insights regarding the sequence of events that span across the many frameworks, libraries, system daemons, kernel, application processes/threads, and result in the performance issue. Traditional tools excel at analyzing system state at a specific point in time in an individual component. They are not amenable to analyzing concurrent execution flows over time whose interactions may cause performance issues.

To debug cross-component performance issues, causal tracing has been proposed [14, 20, 27, 32, 37, 38, 40, 41, 43, 44, 46, 56], especially for distributed systems. Causal tracing utilizes a trace graph to help developers understand performance issues that involve complex interactions. A trace graph consists of vertices and edges, where vertices are *execution segments*, such as an operation, system event, message, etc., and edges indicate causal relationships between vertices. To diagnose a performance issue, these solutions usually run a critical path analysis on the constructed trace graph that finds the sequence of vertices and edges which start from the vertex where the problem occurs and take the greatest amount of time for completion.

Unfortunately, we observe that previous causal tracing approaches are ineffective for desktop applications because they cannot accurately identify the boundaries of execution segments and their causality relationships. For example, a long-standing Google Chrome web browser performance anomaly [2] on macOS occurs when a user enters non-English words in the search box, causing Chrome to hang with the infamous macOS spinning pinwheel, which appears when an application is not responsive to user input. Using previous approaches to construct trace graphs for the multi-threaded, multi-process browser results in many missing execution segments and many additional irrelevant execution segments. Attempting to diagnose the problem using these incomplete and inaccurate graphs would incorrectly pinpoint no events or wrong events as the culprit. In theory, these tracing inaccuracies could be fixed by adding instrumentation, such as adding constraints in noisy trace points to filter irrelevant events. However, frameworks and libraries used by desktop applications have diverse programming idioms and are often closed-source, making deep instrumentation difficult. Extensive instrumentation would also incur prohibitive overhead, resulting in unacceptable performance.

To address these problems, we have created Argus, a causal tracing tool specially designed to help users diagnose performance anomalies in desktop applications. Argus is

based on the insight that tracing inaccuracies are inherently unavoidable in real desktop systems, so instead of trying to eliminate all inaccuracies, we should design tracing solutions that can accommodate some inaccuracies. Argus introduces a new notion of annotated trace graphs, in which edges are explicitly annotated as *strong* and *weak* edges. Strong edges represent connections among segments based on typical programming paradigms that must be causal, such as sending and receiving an IPC message. Weak edges represent ambiguous relationships among segments. For example, when one thread wakes up another thread, it could be a causal relation, e.g., `lock`/`unlock`, or just an artifact of regular OS scheduling. Argus further boosts or prunes unnecessary weak edges by leveraging operation semantics and call stacks.

Argus introduces a new beam search diagnosis algorithm based on edge strength and a novel method of comparing trace subgraphs across normal and abnormal executions of an application. The algorithm is motivated by our observation that critical path analysis used in prior work is ineffective due to inaccuracies inherent in trace graphs. Beam search embraces more possibilities while exploring the annotated noisy trace graph. Our algorithm efficiently selects likely causal paths in the massive trace graph and tolerates noises. Comparing trace subgraphs across normal and abnormal executions also helps with diagnosis when the problem is due to missing operations in the abnormal execution.

Argus provides system-wide tracing by extending existing tracing support in the OS kernel and applying binary patching for low-level libraries. This allows Argus to easily track objects across process boundaries, account for kernel threads involved in communications among processes, and cover customized programming paradigms by operating in a common low-level substrate used by higher-level synchronization methods and APIs that may be introduced and evolve over time. Argus does not require any application modifications.

We have implemented and evaluated a prototype of Argus across multiple versions of macOS. This presents a harsh test for Argus given the many complex, closed-source frameworks, libraries, and applications in the macOS software stack. We evaluated Argus on 12 real-world spinning pinwheel issues in widely-used macOS applications, such as Chrome, Inkscape, and VLC. Argus successfully pinpoints the root cause and sequence of culprit events for all cases. This result is particularly notable given that 10 of the 12 cases are open issues whose root causes were previously unknown to developers. Argus incurs runtime overhead low enough such that users can leave Argus tracing always-on in production without experiencing any noticeable performance degradation. Source code for Argus is available at `https://github.com/columbia/ArgusDebugger`.

2 Motivation and Observations

We experienced first-hand the Chrome web browser performance issue on macOS. Typing non-English words in a search box while a web page is loading causes Chrome to freeze and trigger a spinning pinwheel. The spinning pinwheel appears when an application is not responsive to user input for more than two seconds. Others have also experienced this issue with the Chromium web browser and reported it to Chromium developers [2]; Chrome is based on Chromium.

We study the bug in Chromium since it is open-source, so we can verify its ground truth. Chromium is a multi-process macOS application involving a browser process and several renderer processes, each process having dozens of threads. When a user types a string in the browser search box, a thread in the browser process sends an IPC message to a thread in the renderer process, where the rendering view code runs to calculate the bounding box of the string, which in turn queries `fontd`, the font service daemon, for font dimensions.

To diagnose the bug, we first tried using `spindump` [10], a widely-used macOS debugging tool, which shows the main thread of the browser process is blocking on a condition variable. However, `spindump` provides no clue as to why the condition variable is not signaled. Using macOS Instruments [12] was also ineffective, as it simply analyzes what functions take the most time, which are not the root cause in this case. These traditional debugging and profiling tools are fundamentally not well suited to analyzing causality in highly concurrent execution flows across multiple components over time.

We next tried state-of-the-art causal tracing techniques. Specifically, we use Panappticon [56], a system-wide tracing tool originally built for Android. We reimplemented a version for macOS with more complete tracing of asynchronous tasks, using non-intrusive interposition to trace asynchronous tasks, IPCs, and thread synchronizations from the system and libraries. We use the tool when running Chromium and reproduce the anomaly by typing non-English search strings. After the browser handles the first few characters normally, the remaining characters trigger a spinning pinwheel. We then stop the tracing. The entire session took around five minutes.

Dividing up the trace graph into separate graphs each beginning from a user input event results in 359 trace graphs; user input events are dispatched from the macOS `WindowServer` process to Chromium. The trace graphs are highly complex, with 888,236 vertices and 751,332 edges in total. They span across 11 applications, 79 daemons including `fontd`, `mdworker`, `nsurlsessiond`, and various helper tools started by the applications. They cover 90 processes, 1177 threads, and 644K IPC messages.

Studying the trace graphs, we observe: (i) connections exist between graphs from different UI events; (ii) some long execution segments have no boundaries; (iii) there are orphaned vertices with no edges; (iv) the trace graph that contains the anomalous event sequence triggering the spinning pinwheel contains 12 processes—3 are clearly unrelated to the transaction, and 6 are daemons whose relationships are unclear without further investigation. Based on further analysis of these graphs with call stacks and reverse

```
// worker thread in fontd:
block = dispatch_mig_server;
dispatch_async(block);
```

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

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

Figure 1: Dispatch message batching. `dispatch_mig_server` can serve unrelated applications together.

engineering techniques, we conclude that they have significant inaccuracies. Running diagnosis on them leads to a wild goose chase, investigating components such as `fontd`, as it sends out messages after a long execution, which turn out to be completely unrelated to the root cause. We observe two general inaccuracies: *over-connections* and *under-connections*.

Over-connections usually occur when intra-thread execution segment boundaries are missing. We summarize three common programming patterns responsible for this—dispatch message batching, piggyback optimization, and superfluous wake-ups.

Dispatch message batching. Frameworks and daemons often implement event loops for handling multiple events inside callback functions. For example, Figure 1 shows two threads from the *fontd* daemon in macOS; the worker thread installs a callback function `dispatch_mig_server()` in a dispatch queue and the main thread dequeues and calls the function via `dispatch_client_callout`. `dispatch_mig_server()` has an event loop which batch processes requests from different applications, presumably for performance. It invokes `call_back` to process a message and `set_reply` to post a reply. However, previous causal tracing tools like Panappticon assume the execution of a callback function is entirely on behalf of one request. `dispatch_mig_server` is thus treated as a single execution segment and edges are added between the vertex representing `dispatch_mig_server` and the many unrelated applications for which it handles requests. These edges incorrectly indicate causal relationships that would result in misleading diagnoses.

Piggyback optimization. Frameworks and daemons may piggyback multiple tasks in a system call to reduce kernel boundary crossings. For example, Figure 2 shows the macOS system daemon `WindowServer` uses a single system call `mach_msg_overwrite` to receive data and piggyback the reply for an unrelated event. However, previous causal tracing tools like Panappticon treat the execution of a system call as a single execution segment for one event, artificially making many events appear causally related.

Non-causal wake-up. Desktop applications typically have multiple threads synchronized via mutual exclusion, such that a thread's unlock operation wakes up another waiting thread. Such a wake-up may be, but is not always, intended as causality. For example, in Chromium, a wake-up is commonly followed by a batch processing block, but it is unclear whether

```
//a thread in WindowServer
while (true){
  //postpone a reply
  CGXPostReplyMessage(msg);
  //receive requests
  CGXRunOneServicePass();
}
```

```
CGXRunOneServicePass(){
  if (_gOutMsgPending)
    mach_msg_overwrite(
     SEND|RECV,
     _gOutMsg, RecvMsg)
  else
    mach_msg(RECV,RecvMsg)
}
```

Figure 2: Piggyback optimization and intra-thread data dependency. `mach_msg_overwrite` combines the reply of a previous event. Operations inside a thread have dependencies on `_gOutMsg`.

```
// worker thread needs
// UI update
obj->need_display = 1
```

```
//main thread
if (obj->need_display == 1)
  render(obj)
```

Figure 3: Shared data flag across threads.

the following events being batch processed depend on the wake-up event. Previous causal tracing tools assume any wake-up is causal, which may artificially make events appear causally related when they are not.

Under-connections usually occur due to missing intra-thread data dependencies and inter-thread shared flags.

Data dependency. Frameworks and daemons may have internal state that causally link different execution segments of a thread. For example, Figure 2 shows that a `WindowServer` thread calls the function `CGXPostReplyMessage` to save the reply message, which it internally stores in a variable `_gOutMsg`. When the thread later calls `CGXRunOneServicePass`, it sends out `_gOutMsg` if there is any pending message.

Shared data flags. Frameworks and daemons may use shared flags that causally link different threads. Figure 3 shows a worker thread sets 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. Existing tools do not track these kinds of shared-memory communication.

3 Overview of Argus

We have designed Argus to diagnose performance issues in desktop applications. Argus satisfies four key requirements not met by previous causal tracing tools: (1) use minimal instrumentation, (2) support closed-source components, (3) extract rich information from heterogeneous components with minimal manual effort, and (4) incur low runtime overhead.

Central to its design is the construction of *annotated trace graphs* from low-level trace events. Argus introduces the notion of *strong* and *weak* edges in trace graphs to mitigate inherent inaccuracies in tracing. When there is strong evidence of causality, such as an IPC message event, Argus adds a *strong edge* between vertices. When an execution segment is created by events that may not necessarily represent causality, such as non-causal wake-ups, Argus adds a *weak edge*. During diagnosis, Argus prefers traversing through strong edges when possible. Argus also stores extra semantic information in the graph vertices, including user input events, system calls, and

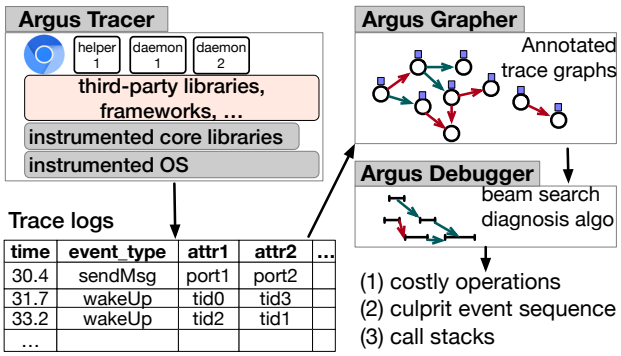

Figure 4: Overview of Argus.

sampled call stacks. This extra information is used to improve weak edge annotation and align and compare trace graphs for normal and abnormal execution to aid diagnosis.

Figure 4 shows an overview of Argus. It consists of three main components—a tracer, a grapher, and a debugger. The tracer runs continuously in the background on a user's machine, transparently logging events from low-level system libraries and the kernel, without any need to modify applications. When a user encounters some performance anomaly, she reports the issue about the problematic application, along with the timestamp of the anomaly occurrence. The reported issue and trace logs are sent to the developer, the logs containing events for both normal execution and abnormal execution when the performance anomaly occurs. The developer feeds the logs into the grapher to construct the annotated trace graphs for both normal and abnormal execution, and runs the debugger on the graphs to output the diagnosis results.

4 Argus Tracer

Argus traces events inside the kernel and low-level libraries, with minimal instrumentation. This provides three advantages over tracing in user applications. First, tracing in the kernel and libraries ensures coverage of custom programming paradigms. For instance, Argus traces general thread scheduling events and wake-up and wait to ensure coverage of a variety of custom synchronization primitives in desktop applications, because their implementations almost always use kernel wake-up and wait. Second, tracing in the kernel helps connect tracing events across process boundaries, because the addresses of the traced objects in kernel space are usually unique, while tracing in user programs requires maintaining and propagating unique identifiers. Third, tracing kernel threads helps bridge communications among processes. For instance, a kernel thread sends out a message to a process when the process needs to execute a delayed function.

In the macOS XNU kernel, Argus traces system calls, thread scheduling, interrupts, time-delayed calls, and Mach messages. Argus leverages existing macOS kernel tracing support [13], but adds enhancements to log more information and enable always-on tracing using a ring buffer to avoid exhausting storage. The enhancements require roughly 500 lines of code (LOC) in the XNU kernel, which are straightforward to add given that the kernel is open source. Trace events are asynchronously flushed to a file with a size limit. The limit is by default 2 GB, which can store roughly 20 million trace events; this is about 5 minutes of tracing when running large applications like Chrome. It can be easily adjusted to accommodate longer execution times. We used the default limit for all experiments in Section 7.

Argus logs kernel events to identify when threads are executing and their causal relationships. All system calls are traced to provide high-level semantics that can be used to identify causal relationships. Argus simply records return values for most system calls, but call stacks are also logged for a small set of system calls, namely those pertaining to Mach messages and synchronization using conditional variables and semaphores. Call stack information is later used by the Argus debugger to provide debugging information for developers. Thread scheduling is traced to track when a thread becomes idle and which thread wakes it up. Argus logs three types of thread scheduling events: *wait* to indicate when a thread becomes idle, *wake-up* to indicate when the current thread wakes up another thread, and *preempt* to indicate when a thread is preempted due to its timeslice being used up or priority policies. Interrupts are logged to indicate when threads are preempted by interrupts, with call stacks also logged for interprocessor interrupts (IPIs). Argus traces the internal kernel implementation of time-delayed calls, which are used to implement asynchronous calls in libraries such as Grand Central Dispatch (GCD). Finally, Argus traces the internal kernel implementation of Mach messages, not just their invocation via system calls, to enabling tracing of all use of Mach messages, including use within the kernel among kernel threads.

To aid developers in interpreting the virtual addresses in call stacks via `lldb`, Argus also logs in userspace the virtual memory layout of images for all processes. The tracer records the virtual memory maps for all running processes when tracing is enabled or terminated; processes launched during tracing are also recorded. The memory layout information is also fed to the Argus debugger.

In addition to kernel tracing, Argus traces four closed-source macOS frameworks, `AppKit`, `libdispatch.dylib`, `CoreFoundation`, and `CoreGraphics`, to track UI events and batch processing paradigms used by applications. Because these frameworks are closed source, the trace events are added via binary instrumentation using a mechanism similar to Detour [31]. `AppKit` is used to dispatch UI events to handlers. Argus traces where a UI event is fetched from the `WindowServer` and dispatched to an event handler. `libdispatch.dylib` implements GCD, managing dispatch queues to balance work across the entire system. Argus adds trace events to track when objects are pushed into a dispatch queue and popped off of the dispatch queue and executed. `CoreFoundation` supports event loops for GUI applications, which are widely used to process requests

from timers, customized observers, and sources such as sockets, ports, and files. Argus adds trace events so the handling of different requests inside event loops can be tracked separately.

To deal with the under-connection issues (Section 2), we annotate a handful of data flags in `CoreGraphics`. Given the shared flag variable names, Argus monitors the respective virtual addresses with watchpoint registers. Reads or writes to the addresses will invoke a signal handler that records trace events with the values stored in those addresses. Argus adds code to `CoreFoundation` to install this signal handler.

Argus can use the same watchpoint mechanism to trace shared data flags in applications. To assist developers in finding these shared data flags, Argus provides a lightweight tool that uses `lldb` to record the operand values of each instruction and finds ones that lead to divergence in control flow, which are likely data flags. The shared flag variable names are recorded in an Argus tracer configuration file, which are then traced using the same signal handler installed by `CoreFoundation`. Since `CoreFoundation` is imported by all GUI applications, Argus can trace these shared data flag accesses without any application modifications.

Note that the annotation effort for shared data flags is in general small. This is because execution segments that access shared variables are usually connected already by some types of causality, e.g., wait/signal events; developers mainly need to provide Argus with shared flags that are accessed through ad-hoc synchronization [49]. In our experience, only a few shared flags need to be monitored. Also for this reason, although hardware watchpoint registers are limited, Argus is unlikely to exhaust them. In fact, none of the applications we evaluated in Section 7 needed shared flags to be identified or traced in the applications themselves. Mechanisms such as Kprobe [3] could potentially be used to extend Argus to support monitoring more shared flags.

5 Argus Grapher

Argus uses the trace logs to build an annotated trace graph by first identifying the boundaries of execution segments in each thread to determine the graph vertices, then adding annotated edges between vertices. The annotated edges contain type metadata to indicate *strong* versus *weak* edges, which is used during diagnosis to mitigate inaccuracies due to over-connections and under-connections, as discussed in Section 2.

Argus first determines the execution segments that will form the graph vertices. Using various trace events as boundaries, Argus splits the execution of each thread is into separate execution segments. First, Argus splits nesting of tasks executed from dispatch queues. If an execution of `dispatch_callout` invokes several other `dispatch_callout`, each dispatched task is separated. Second, Argus recognizes batch processing patterns such as `dispatch_mig_server()` in Figure 1 and splits the batch into separate execution segments. Third, when a wait operation blocks a thread execution, Argus splits the execution into separate segments at the entry of the blocking wait. The rationale is that blocking wait is typically done as the last step in event processing. Finally, Argus uses Mach messages to split execution when the set of communicating peers differs. Argus maintains a set of peers, including the direct sender or receiver of the message and the beneficiary of the message; macOS allows a process to send or receive messages on behalf of a third process. Argus splits execution when two consecutive messages have non-overlapping peer sets. By splitting thread execution using these four criteria, Argus avoids potential over-connections due to batching and piggyback optimizations.

Edge	Rules for Edge Annotation
Strong	1. IPC message send and receive; 2. Asynchronous calls (work queue, delayed call); 3. Direct wake-up of a thread on purpose; 4. Data dependency.
Weak	1. Non-causal wake-up; 2. Execution segments divided between a wait event and a wake-up event, excluding following cases: wait or wakeup are introduced by system call `workq_kern_return`, or they are in `kern_task`; 3. Split suspicious batching execution segments, except known batching APIs: `RunLoopDoObservers`, `CGXServer`, etc.
Boosted Weak	Continuous execution segments matching weak edge rules but are on behalf of the same task.

Table 1: Edge annotation rules.

Argus next determines the edges that should be added between vertices. Edges are introduced to reveal the causality of two execution segments and thus guide the causal path exploration. Based on the rules in Table 1, Argus annotates three types of edges: *strong*, *weak*, and *boosted weak*.

First, Argus adds strong edges by identifying Mach message, dispatch queue, time-delayed call, and data flag trace events associated with a vertex and finding the corresponding peer events and peer vertices. For Mach message events, Argus adds a strong edge from the vertex with the message send event to the vertex with its associated receive event. If a message requires a reply, the received message can produce a reply message, which can be sent by a third thread, in which case Argus adds a strong edge from the vertex with the received message event to the one with the send event for the reply message. For dispatch queue events, Argus adds a strong edge from the vertex where the callback function is pushed to a dispatch queue to the vertex where the callback function is invoked. For time-delayed calls, Argus adds a strong edge from the vertex where the timer is armed to the vertex where the callback function is fired. For shared data flags, Argus adds a strong edge from the vertex with a data flag write event to the vertex with its corresponding read event, avoiding potential under-connections.

Second, Argus adds edges by identifying thread scheduling trace events and finding the events and vertices corresponding to the pair operations. Argus adds strong edges only when the context clearly indicates causality, such as the signal and wait operations of a condition variable. Otherwise, Argus adds only weak edges. One hint Argus takes from macOS is

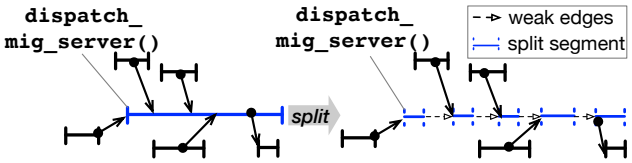

Figure 5: The segment for batch processing in `dispatch_mig_server` is split into multiple segments to distinguish different items. Weak edges are added among the split segments.

that, if a wake-up is not followed by a specific communication operation (*e.g.*, message receive), and does not target a specific thread but all threads on the wait queue, then it is likely not causal, in which case a weak edge is added.

Third, because Argus splits the execution of a thread into segments (graph vertices) based on heuristics that may not always be valid, Argus adds weak edges between these adjacent execution segments, as shown in Figure 5. Argus converts a weak edge into a *boosted weak edge* if two continuous execution segments are on behalf of the same task. It infers whether the segments are for the same task by leveraging call stack symbols. We calculate frequencies for all symbols across the whole tracing and notice a low-frequency (bottom 10%) symbol usually only appears in a task from a specific application, compared to high-frequency symbols from system routines or framework APIs. Thus, if the two segments share the same low-frequency symbols, Argus infers they are collaborating on the same task and sets a boost flag for the weak edge between them.

However, abuse of weak edges could generate excessive false positives during diagnosis, so Argus takes advantage of high-level semantics to avoid adding unnecessary weak edges between adjacent execution segments. First, if the call stacks of two segments of a thread share no common symbols or share a recognized system library batching API, Argus does not add a weak edge between them. Second, because wait and wake-up events are mostly from system calls, Argus leverages system call semantics to determine the necessity of weak edges. For example, we find the wait event from system call `workq_kern_return` indicates an end of a task in the thread, while the wake-up event formed in `workq_kern_return` intends to acquire more worker threads for concurrent tasks in the dispatch queue. Execution segments containing such event sequences do not need bridging with weak edges. Finally, the kernel task in macOS acts as a delegate to provide service for many applications, such as I/O processing and timed delayed invocations. The kernel task threads contain execution segments beginning with a wake-up event and ending with a wait event. Each segment serves different requests and they are not causally related, so weak edges are not added between those kernel task execution segments.

6 Argus Debugger

Argus uses the constructed trace graphs to diagnose performance issues by starting with the vertex that contains the

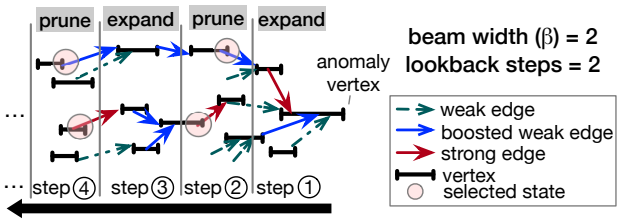

Figure 6: Beam search diagnosis algorithm. Search backwards from the anomaly vertex; choose the best β states to expand next. For every lookback steps, prune the existing states to at most β paths.

performance anomaly and traversing the graphs to identify the causal paths including the root cause vertices. The typical critical path analysis used in existing causal tracing solutions cannot effectively handle the noises in the trace graphs. Argus introduces a new diagnosis algorithm based on *beam search* to efficiently explore the causal paths likely related to the performance anomalies. It also introduces a novel subgraph comparison mechanism to find missing vertices not present in the trace graph for abnormal execution that are present in the graph for normal execution. This comparison is helpful to identify the root cause that would be otherwise unknown.

6.1 Causal Path Search—Beam Search

From a given vertex that contains the anomaly, such as the spinning cursor, Argus finds what path "caused" the anomaly by using beam search based on a cost function for annotated edges. Beam search is similar to breadth-first-search, but at each search step, it sorts the next level of graph vertices based on a cost function and only stores β—the beam width—best vertices to consider next. Argus customizes its beam search with a *lookback* scheme such that the algorithm evaluates the cost function for multiple levels of edges before pruning. Argus evaluates the vertices and prunes them with β only after the search advances the configured *lookback* steps to avoiding pruning paths with weak edges too early.

Argus's beam search algorithm provides two key advantages. First, compared to brute-force search, beam search only explores the most promising vertices, which is essential given that trace graphs are highly complex with millions of edges; searching all paths would be too inefficient and, given graph inaccuracies, result in an overwhelming number of options to consider. Second compared to local search methods such as hill-climbing, beam search embraces more possible causal paths because it ranks partial solutions and the ranking changes during the exploration. For example, assuming strong edges are preferred to weak ones, a path with a weak edge followed by a series of strong edges is likely to get a higher ranking and be returned by beam search, but will be missed by a hill-climbing search algorithm.

Figure 6 illustrates the algorithm. It searches for causal paths *backwards* from the anomaly vertex. For each incoming edge of the current vertex, the algorithm computes the *penalty score* for the new path. At every lookback step, the search branches

Algorithm 1: Causal Path Search Algorithm (Beam Search).

```
   Data: g - event graphs, curVertex - vertex inspected in current search
         state, beamWidth - search branches at most, lookbackSteps -
         searching steps taken before pruning current search branches
   Result: paths
1  Function BeamSearch(g, curVertex, beamWidth, lookbackSteps):
2      curStates.init(curVertex);
3      curSteps ← 0;
4      while curStates.incoming_edges() > 0 && beamWidth > 0 do
5          ++curSteps;
6          newStates.clear();
7          for each state ∈ curStates do
8              if beamWidth <= 0 then
9                  break;
10             end
11             if state.path.reach(UI) || state.path.incoming_edges = ∅
                 then
12                 paths.add(state.path);
13                 −−beamWidth;
14             end
15             for each edge ∈ state.path.incoming_edges do
16                 newState.path ← state.path + edge;
17                 newState.score ← state.score + penalty(edge.val);
18                 newStates.add(newState);
19             end
20         end
21         curStates ← newStates;
22         if curSteps = lookbackSteps then
23             pruneStates(curStates, beamWidth);
24             curSteps ← 0;
25         end
26     end
27     pruneStates(curStates, beamWidth);
28     paths.append(curStates.paths);
29     return SortIncPenaltyScore(paths);
30 Function pruneStates(newStates, beamWidth):
31     SortIncPenaltyScore(newStates.paths);
32     while newStates.size() > beamWidth do
33         newStates.pop_back();
34     end
35     return;
```

Algorithm 2: Subgraph Comparison Algorithm.

```
   Data: anomVertex – problematic vertex, anomGraph – trace graph
         for anomaly case, normGraph – trace graph for normal case
   Result: ret- potential culprits of anomaly
1  Function SubGraphCompare(anomVertex, anomGraph,
     normGraph):
2      ret.clear();
3      similarVertices ← FindSimilarVertices(normGraph,
         anomVertex);
4      baselineVertex ← GetBaseLine(similarVertices, anomVertex);
5      targetVertex ← woken(normGraph, baselineVertex);
6      causalPaths ← BeamSearch(normGraph, targetVertex,
         beamWidth, lookbackStep);
7      // sub-graph is constituted with paths;
8      for each causalPath ∈ causalPaths do
9          for each vertex ∈ causalPath do
10             expectVertex ← SimilarVertex(anomGraph, vertex);
11             if expectVertex = ∅ then
12                 // missing similarity to vertex;
13                 anomThr← SearchThread(anomGraph,
                     vertex.thread);
14                 // get the vertex that causes the dissimilar;
15                 suspVertex← VertexInThread(anomGraph,
                     anomThr);
16             else if DifferentVertices(expectVertex, vertex) then
17                 // vertex acts different from normal case;
18                 suspVertex← expectVertex;
19             else
20                 countinue;
21             end
22             ret.push_back(suspVertex);
23         end
24         if !ret.empty() then
25             return ret;
26         end
27     end
28     return ret;
```

are pruned: it sorts the paths by their penalty scores and only retains at most β paths with low penalties. A path is added to the result if a vertex is reached containing a UI event or has no incoming edges, and the beam width decreases by one. Using such vertices as for path termination helps developers understand causality in an end-to-end request handling transaction.

Algorithm 1 lists the pseudo-code of the search algorithm. Lines 16 – 18 compute penalty scores for new paths after incoming edges are added to the path. Lines 22 – 25 prune the searched branches every L lookback steps. Paths are sorted by their penalty scores and paths with high penalties are discarded. Penalty scores are calculated with a linear function on edge values, where a strong edge is -1, a weak edge is 1, and a boosted weak edge is 0. A path with n edges has a penalty $p = \sum_{i=1}^{n}(a \times E_i + b)$, where E_i is the ith edge value. This approach guides search towards paths with stronger causality. While more complex non-linear functions may be feasible, this simple function works well for many diagnosis cases.

The beam width setting affects the search efficiency and diagnosis accuracy. A setting too large would cause path explosion and noisy paths to be returned. A setting too small may easily miss the true causal path. We set $\beta = 5$ to strike a good balance. Tuning this parameter is relatively easy in practice. The lookback step setting is set based on observing that traversal of most graphs encounters a weak edge within five steps. We set $L = 5$ to tolerate weak edges. Given this setting, a path of x strong edges, y weak edges, and z boosted weak edges has a penalty of $p = -a \times (x-y) + 5 \times b$. If all edges are strong, the penalty is negative only when $b < a$. If there are weak edges, the penalty is positive only when $(x-y) \times a < 5 \times b, where -3 < x-y < 3$. Therefore, we set the default penalty function coefficients $a = 3$ and $b = 2$.

6.2 Subgraph Comparison

If we run causality analysis only on the trace graph constructed with the anomalous performance issue, the root cause may not be exposed in some cases. For example, a blocked function could be caused by a missing wake-up from one of the background threads. If the thread does not perform the wake-up during abnormal execution, there will be no execution segment with the wake-up, and therefore no vertex in the anomalous trace graph that can be identified correctly as the root cause. Argus addresses this problem by first constructing the trace graphs for both normal and abnormal execution. It then uses its beam search method on the normal trace graph to identify the causal paths in that graph that corresponds to the desired normal behavior that does not occur during abnormal execution. We refer to those causal paths a *subgraph*.Argus then uses the vertices in the subgraph to identify the missing root cause in the abnormal execution. This is done by introducing a novel sub-

graph comparison method between the trace graphs for both normal and abnormal execution, which is listed in Algorithm 2.

Argus first determines a baseline vertex in the normal graph that is comparable to the anomaly vertex in the anomalous graph. Argus computes a signature for each vertex based on the trace event sequence in its execution segment. The signature is composed of two parts, one that encodes the types corresponding to the event sequence e.g. 0 for IPC event, 1 for syscall event, etc., and another that is a hash of the event parameters, e.g., process names of IPC events. Argus calculates the similarity of two vertex signatures using string edit distance. Among the vertices in the normal graph that are similar to the anomaly vertex, Argus chooses one that behaves differently from the anomaly vertex, based on return values of system calls and execution times. For example, a vertex whose last event is a blocking system call with a timed wait may behave in two different ways, timing out or quickly woken up.

After Argus identifies a baseline vertex, it obtains its causal paths using Algorithm 1. The result is a subgraph of the normal trace graph rooted from the baseline vertex to some ending vertex. Argus examines the subgraph from the most related causal path. Starting with the ending vertex V, whose execution segment was executed by some thread T, Argus identifies vertices in the abnormal trace graph that were also executed by T. For each identified vertex, Argus checks whether it behaves differently from V, in which case it is flagged as a suspicious vertex. If no such vertices are found, Argus repeats this procedure with the next vertex in the subgraph. Otherwise, for each suspicious vertex that has incoming edges, Argus recursively repeats the subgraph comparison by treating the suspicious vertex as the initial anomaly vertex. The recursive procedure effectively keeps working backwards through vertices to eventually find a set of root cause candidate vertices in the anomalous trace graph with no incoming edges. Argus then returns the vertex whose path to the original anomalous vertex has the lowest penalty score, identifying that vertex as the root cause.

Figure 7 shows a simplified example of the subgraph comparison method applied to the Chromium performance issue discussed in Section 2. Vertex E' in the anomalous graph is the initial anomaly vertex. Argus identifies vertex E in the normal graph as having a similar signature but behaving differently, and treats it as a baseline vertex. Argus applies beam search to the normal graph starting with vertex E, resulting in the subgraph $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E$. Argus starts with A, identifies its browser thread, and determines that A cannot be the root cause since the same browser thread contains the performance anomaly E' in the anomalous trace graph. Argus then considers B, identifies its renderer thread, and finds all vertices in the anomalous trace graph executed by the renderer thread. F' is similar to F, so it is not considered a suspicious vertex, but J' is not similar to any vertex in the normal trace graph, so it is considered suspicious. J' has no incoming edges and is identified as a root cause candidate. If there are no other candidates identified, J' is returned as the root cause.

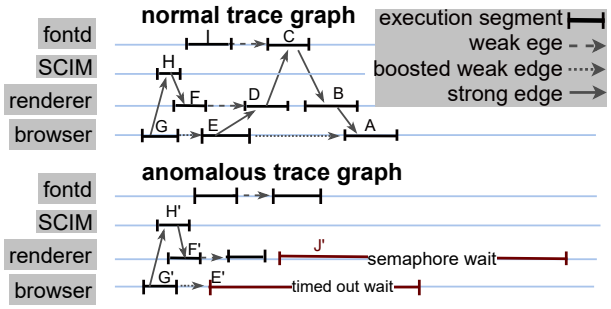

Figure 7: Chromium normal and anomalous trace graphs after user typed in a search box (vertex G/G'). Vertex E' (requesting a bounding box for input) is the anomaly vertex. Sub-graph in normal trace graph is extracted from baseline vertex E. Vertex J'(javascript processing blocks on semaphore) is the root cause Argus reported. Trace graphs are simplified for clarity; only processes are shown and communications with processes such as `imklaunchagent` are omitted.

6.3 Debug Information

Argus further provides the calling contexts of the anomaly vertex and the root cause vertex to help developers localize the bug in code. To do so, Argus examines the call stacks it attaches in the graph vertices. If the anomaly or root cause vertex has a blocking call, the call stack Argus tracer collects would reveal the context of the blocking call directly. If the vertex has a long runtime cost, the problematic vertex usually contains periodic IPIs, where the Argus tracer collects call stacks. In this case, the Argus debugger calculates the longest common sequence of frames from those call stacks. The top frame in the sequence reflects the costly function call.

For instance, in Figure 7, Argus reports the following information: (i) the calling context of problematic vertex E' and its causal path $E' \leftarrow G'$; (ii) the calling context of root cause vertex *J'* along with its *unmatched causal path* in baseline trace graph: $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E \leftarrow G$, and vertex B is marked because its thread should have waken up the blocking thread in the anomaly case.

6.4 Diagnosis for Spinning Pinwheel in macOS

Argus's debugger can be used to effectively diagnose spinning pinwheel performance issues in macOS applications. Recall that a spinning pinwheel appears when the UI thread of an application can not process any user inputs for over two seconds. During normal execution, the two-second interval may cover many vertices, but when the spinning pinwheel appears, the main thread of the application is stalled and the two-second interval covers only a single vertex. Leveraging this timing information, Argus identifies the anomaly vertex in the main thread of the targeted application and classifies the issue as either a *LongRunning* and *LongWait* anomaly.

LongRunning. The main thread is busy performing lengthy CPU operations and therefore its execution segment is in the anomalous trace graph. Argus uses its beam search method

to identify the causal path between the anomaly vertex and the vertex with the UI event resulting in the issue. Argus reports the costly API, event handler, and causal path to the developer.

LongWait. A UI thread is blocked, but it is hard to tell why. Argus uses its subgraph comparison method together with its beam search method to deduce which vertex is missing from the anomalous trace graph. A long-wait event could be caused by another long-wait event. Argus supports recursively diagnosing "the culprit of the culprit." Therefore, it can reveal deep root causes. At the end of each iteration of diagnosis, the calling context of problematic vertex, root cause vertices in the anomalous trace graph, and causal paths are ranked and reported to users.

Some LongRunning issues may be diagnosed with existing tools such as `spindump` if the profiling is accurate and complete. However, Argus is better in that a call stack is usually not enough to connect the busy processing to the event handler, due to the prevalence of asynchronous calls. Also, call stack profiles after the anomaly may miss the real costly operations. LongWait issues usually involve multiple components and are extremely hard to understand and fix with current tools. Those issues may remain unresolved for years and significantly hurt user experience and developer productivity.

7 Evaluation

We have implemented Argus across multiple versions of macOS, ranging from El Capitan to Catalina. We evaluate Argus to answer several key research questions: (1) Can Argus effectively diagnose real-world performance anomalies for modern desktop applications? (2) How does Argus compare to other performance debugging tools? (3) How useful are Argus's weak edges and their optimizations in mitigating tracing inaccuracies? (4) How much overhead does Argus's tracing tool incur? Unless otherwise indicated, all applications and tools were run on a MacBookPro12,1 with an Intel Core i7 CPU, 16 GB RAM, and an APPLE SM0512G SSD.

7.1 Diagnosis Effectiveness

We evaluated Argus on 12 real-world user-reported performance issues in 11 popular desktop applications, which we collected and reproduced, as listed in Table 2. We are especially interested in evaluating performance issues that have been hard to troubleshoot. Except for B11, all of these are open issues, meaning their root causes were previously unknown to developers. For B2, the reported issue was "fixed" in the latest version (due to refactoring or platform upgrade) but the root cause remained unknown. Nine applications, or some of their components, have source code available, whereas two applications are closed-source. Source code was used to validate whether the correct root cause was diagnosed for the performance issues, but all evaluation was performed on the released application binaries. We have also used

ID	App	Performance Issue	Age
B1	Chromium	Typing non-English in searchbox, page freezes.	7 yr
B2	TeXstudio	Modifying Bib file in other app gets pinwheel.	2 yr
B3	BiglyBT	Launching BiglyBT installer gets pinwheel.	1 yr
B4	Sequel Pro	Reconnection via ssh causes freeze.	4 yr
B5	Quiver	Pasting a section from webpage as a list freezes.	5 yr
B6	Firefox	Connection to printer takes a long time.	1 mo
B7	Firefox	Some website triggers pinwheel in the DevTool.	3 yr
B8	Alacrity	Unresponsive after a long line rendering.	6 mo
B9	Inkscape	Zoom in/out shapes causes intermittent freeze.	1 yr
B10	VLC	Quick quit after playlist click causes freeze.	7 mo
B11	QEMU	Unable to launch on macOS Catalina.	1 mo
B12	Octave	Script editing in GUI gets pinwheel.	2 yr

Table 2: Real-world performance issues in macOS applications.

Argus with proprietary applications like Microsoft Word for macOS, but without source code, we need to wait for vendors' confirmation and responses; in our experience, vendors are reluctant to communicate issues with an external party.

Table 3 shows that Argus was able to diagnose all 12 performance issues, including all longstanding open issues. As listed in Table 4, we checked the correctness of Argus's diagnosed root causes in three ways: (1) inspecting the corresponding source code if available, (2) dynamic patching with `lldb` based on the diagnosed root cause to fix the problem, and (3) confirmation by developers. The last one is ideal, but not always feasible; we reported our findings to developers for seven issues, but only received two responses. Only the root cause of B11 was previously known, which Argus returned correctly (Grd). For B1, B7, and B10, we validated the diagnosed root causes by analyzing the source code (Src). For B2 and B4, we received confirmation from the respective application developers that Argus correctly diagnosed the root cause for these open issues [8, 9] (Dev). For example, for B4, the Sequel Pro developers suspected a particular Cocoa Framework API does not work as expected, but could not pinpoint the exact place to fix it. Argus determined the defect was in their installed callback function, and we submitted a pull request [8] to fix the issue. B8 was fixed in an official developer patch after we reported the root cause (Fix). For the remaining issues, we confirmed the issue was resolved by dynamically patching the application based on the root cause (Dyn). We describe a few of the performance issues in further detail, but omit others due to space constraints.

B1-Chromium: This is the Chromium performance issue discussed in Section 2. Argus analyzes the trace graph, pinpoints the circular waits between renderer main thread and browser main thread with the interactions of daemon processes like `fontd`. Argus not only localizes the problematic execution segment (waiting on a condition variable), but also the sequence of events leading to this issue. The same issue occurs in Chrome. We also reported our findings to Chrome developers, but received no reply.

B2-TeXstudio: TeXstudio [55] is an IDE for creating LaTeX documents. Users reported when they modified a bibliography

ID	Root Cause Identified
B1	circular wait between renderer and browser main threads.
B2	long running function calculating line indices in document.
B3	recursive invocations of accessible objects in GUI.
B4	UI event loop mishandling input causes deadlock with ssh.
B5	paragraph value never equals last paragraph inside web view.
B6	sleep waiting on chain of deamons, the last being nsurlsessiond.
B7	excessive garbage collection on the main thread.
B8	excessive copy of rendering cells when searching potential URL.
B9	excessive memory operations for trimming and compositing.
B10	termination signal before displaying thread ready; deadlocks.
B11	window adjustment before it finishes launching; deadlocks.
B12	readline thread writing tty repeatedly, main thread waiting.

Table 3: Root causes identified by Argus.

	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12
`spind.@top1`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
`spind.@top3`	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗
`spind.@top5`	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✗
`spind.@top10`	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✓	✗
`Instr.@top1`	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
`Instr.@top3`	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
`Instr.@top5`	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
`Instr.@top10`	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗
AppInsight	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
Panappticon	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗
Argus	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
no weak edges	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
w/critical path	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗	✗	✗
Argus result validation	Src	**Dev**	Dyn	**Dev**	Dyn	Dyn	Src	Fix	Dyn	Src	Grd	Dyn

Table 4: Comparing Argus with other debugging tools.

file with another application, TeXstudio froze with a spinning pinwheel. We reproduced this case by running `touch` from a terminal on a 500 entry bibliography file, which immediately caused a spinning pinwheel to appear in *TeXstudio*'s window. Argus analyzes the trace graph and identifies five causal paths, ordered by likelihood of causality. The first path connects multiple entities: Terminal→`WindowServer`→`bash`→`kernel_task`→`fseventd`→TeXstudio—and suggests the following root cause chain. `touch` triggers a change in the file metadata. `fseventd` notifies TeXstudio and invokes a callback handler. TeXstudio executes a function `QDocument::startChunkLoading`, and causes busy processing in TeXstudio's main thread. Argus also outputs the call stack with the busy APIs, `startChunkLoading` and `QDocumentPrivate::indexOf()`. We reported our findings to the developers and received confirmation that the diagnosis is correct.

B5-Quiver: *Quiver* [7] is a closed-source notebook application for mixing text, code, Markdown, LaTeX, etc. Users report that applying bullet points to a text cell without an empty line at bottom causes a spinning pinwheel [6]. Based on the Argus trace graph, there is a hanging vertex in the WebKit component used by Quiver. In particular, WebKit hangs in executing `InsertListCommand::doApply` when applying the list command to the Webview context from Quiver. The hang occurs because of an infinite loop bug in WebKit rather than Quiver. We verified the root cause by changing the comparison result of the loop with `lldb`, which enables Quiver to display the bulletin points without a spinning pinwheel. We reported our findings to the developers, but received no reply.

7.2 Comparing with Other Approaches

We compared Argus versus other state-of-the-art tools for diagnosing the performance issues in Table 2. We used two widely-used traditional debugging and profiling tools from Apple, `spindump` [10] and `Instruments` [12]. For `spindump`, we enable it once the performance issue appears, and repeat the process five times to eliminate bias on the start timing. `spindump` separately ranks the symbols from all sampled call stacks and only the top of call stacks. We examined the top$N$ symbols and their corresponding call stack information. For `Instruments`, we enable its time profiler in the background when reproducing the bugs, and analyze its data from two seconds before the performance issue occurs to three seconds after. We rank APIs in the reported call trees with CPU time percentage and filter out system routines. Then, we select the top$N$ APIs for investigation. We used values from $N = 1$ to $N = 10$. We also used two causal tracing tools, the macOS version of Panappticon, as discussed in Section 2, and AppInsight [40]. Since AppInsight was originally built for Windows, we reimplemented a version for macOS which captures trace events, constructs trace graphs, and follows the path analysing rules for diagnosis according to AppInsight's design.

Table 4 shows the results for using the different tools, including the results for Argus discussed in Section 7.1; checks indicate correct root cause diagnosis. All of the other tools diagnosed much fewer performance issues than Argus. `spindump` diagnosed at most five issues. It captures the state near the symptom point but cannot deduce how the execution reaches a problematic point, especially in the presence of highly concurrent and asynchronous execution across different entities. `Instruments` diagnosed at most four issues. It only outputs the most costly functions, which are helpful for performance optimizations but may not be for troubleshooting specific performance issues. Neither of the causal tracing tools did any better because the constructed trace graphs are highly inaccurate. AppInsight only diagnosed two issues while Panappticon diagnosed four issues.

7.3 Mitigation of Trace Graph Inaccuracies

We evaluated the effectiveness of Argus in mitigating trace graph inaccuracies in diagnosing the performance issues in Table 2. Table 4 shows the benefits of weak edges and Beam

	Events	Vertices	Edges		
			Total	Strong	Weak
Max	12.3M	1.68M	1.62M	751.3K	864.6K
Min	260.8K	15.1K	25.5K	17.5K	8.01K
Mean	3.31M	349.5K	358.4K	188.8K	169.6K
Med	1.02M	97.3K	172.6K	111.9K	60.71K

Table 5: Argus trace graph statistics.

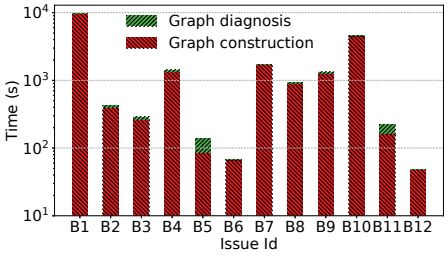

Figure 8: Argus diagnosis time.

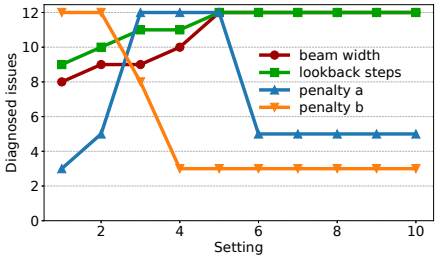

Figure 9: Sensitivity of beam search settings.

search. Argus diagnoses eight issues if it discards weak edges (no weak edges), and seven issues if it uses traditional critical path analysis instead of Beam search (w/critical path). In both cases, Argus still performs better than other tools.

Table 5 shows that the Argus trace graphs include hundreds of thousands to millions of events, and on average have 350K vertices and up to 1.68M vertices. Graphs are in general dense, with an average of 358K edges. A significant percentage, 40% on average, of the edges are tagged as weak edges. To avoid abusing weak edges and overwhelming the diagnosis, Argus applies the optimizations discussed in Section 5. Figure 10 shows the percentages of potential weak edges that Argus excludes from the trace graph for different techniques: call stack similarity, wait on end of task in a thread, acquire worker threads, and kernel task delegate. Call stack similarity was most effective in pruning potential weak edges.

We evaluated the sensitivity of Argus's beam search settings: beam width, lookback steps, and penalty function coefficients a and b. Figure 9 shows the number of diagnosed issues when changing one setting and leaving the rest at their defaults. The settings for beam width and lookback steps are robust. Larger settings increase the diagnosis effectiveness, but if they are too large, the Argus debugger could run out of memory or time out for large trace graphs. Changing penalty function coefficients can significantly change the number of diagnosed issues. In general, small coefficients from two to four are better. Overall, the results indicate that Argus is practical, and developers do not need to spend much effort to tune search settings.

7.4 Performance

We measured the time to run the Argus grapher and debugger for diagnosing each of the performance issues in Table 2. Figure 8 shows the time varies for different issues, ranging from 49 s (B12) to 9870 s (B1). Constructing the trace graph is the dominant cost. Running the beam search diagnosis algorithm on the graph is fast, taking at most 144 s (B10).

We also measured the overhead of the Argus tracer using various CPU, memory, and I/O benchmarks running on a live deployment of Argus on a MacBookPro9,2 with an Intel Core i5-3210M CPU, 10 GB RAM, and a 1 TB SSD. We first measured five runs of the iBench Cocoa benchmark [35], with and without Argus, to measure overall performance. The reported scores were 6.14 with 0.027 standard error without Argus tracing and 6.13 with 0.025 standard error with Argus tracing enabled. Argus only has a 0.16% performance degradation on average. In comparison, with `Instruments`, the reported score was 6.04, showing a 1.6% performance degradation. We next ran the Chromium Catapult benchmarks [1] to evaluate CPU performance, with and without Argus tracing. Figure 11 shows that Argus overhead is less than 5%. The average overhead for `real` and `user` time was 3.36% and 2.15%, respectively. `sys` overhead was higher because Argus tracing in libraries involves crossing the user-kernel boundary. Finally, we ran Bonnie++ [22] and IOzone [19] I/O benchmarks to evaluate I/O performance, with and without Argus tracing. Figure 12 shows the I/O throughput measurements. Argus tracing has almost no overhead for sequential character read and write operations and less than 10% overhead for block read and write operations.

8 Discussion and Limitations

Diagnosis in Argus may require the anomalous execution trace as well as the normal one for comparison. Obtaining the latter is not difficult. Persistent performance problems are typically eliminated before release, so the remaining issues are often non-deterministic, only occur with specific input events (e.g., typing special characters), and disappear with other events.

The quality of the Argus diagnosis results is affected by edge annotation accuracy. Beam search helps tolerate errors by inspecting multiple paths, but its settings can affect diagnosis effectiveness, as discussed in Section 6.

Argus addresses performance issues that are reflected in the underlying execution sequences and CPU time. It does not handle performance issues due to contentions among userspace threads or incorrect settings of UI elements.

Argus supports closed-source applications and libraries, but its tracing infrastructure requires slight source-level kernel modifications. System libraries such as CoreFoundation are patched at the binary level. Binary instrumentation could also be used to implement kernel changes, but is more cumbersome. Vendors of proprietary OSes have incentives to enhance their existing tracing mechanisms, and may conceivably adopt Argus kernel modifications.

We have not yet ported Argus to other OSes, but modern OSes share many similarities and provide tracing facilities that can support Argus, such as ETW [39] in Windows and LTTng in Linux [4]. Therefore, we are hopeful that our ideas are generally applicable to other OSes.

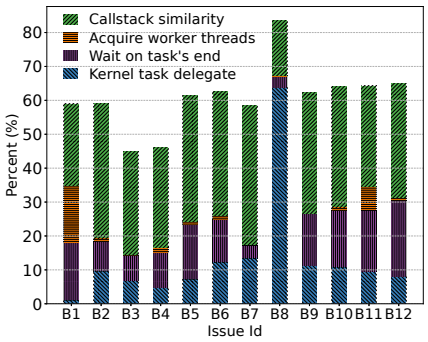

Figure 10: Potential weak edges pruned.

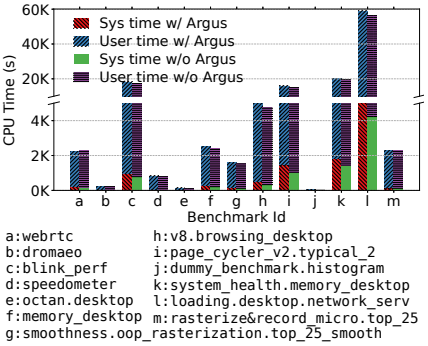

Figure 11: CPU overhead.

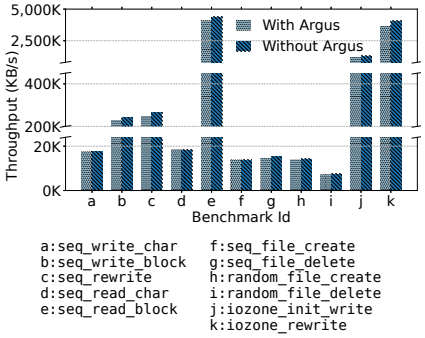

Figure 12: I/O overhead.

9 Related Work

Many causal tracing solutions have been proposed for networked and distributed systems, including Magpie [14], XTrace [27], Dapper [44], and Pivot Tracing [38]. These systems typically attach metadata to each request, propagate the metadata to all components, and stitch the traces. This approach assumes (1) the system is composed of white-box components that can be easily modified; (2) these components communicate in uniform interfaces. Neither assumption is true for desktop systems. Magpie [14] does not use metadata propagation but assumes a manual schema to extract and join events from different components' logs. The extracted traces are limited by what each component chooses to log. However, desktop components typically are packaged as release builds that only log critical events, and logging practices among components vary greatly, which makes writing uniform schema difficult, time-consuming, and fragile.

Some causal tracing tools have been developed for mobile applications. AppInsight [40] interposes on the interface between applications and Windows Mobile frameworks and assumes that applications follow the event callback programming idiom. Panappticon [56] traces low-level events in Android and assumes two asynchronous programming idioms, message queue and thread pooling. Neither of these approaches is effective for desktop applications such as those in macOS.

Profiling or static code analysis are typically ineffective for detecting performance issues [23, 34]. Several solutions [29,54] detect performance anomalies by leveraging logs and call stacks. Other works [21, 24, 42, 50] apply machine learning methods to identify anomalous events. Yu *et al.* [52] study the performance impact of Windows device drivers in real-world execution traces and propose to extract wait graphs from the execution traces. Several solutions [15–17] infer models from logs for distributed and concurrent systems, and use them to automate the detection of anomalous behavior when systems are exposed to new workloads and environments. These systems are orthogonal to Argus, as Argus's goal is to diagnose an already-detected performance anomaly.

Argus is complementary to the work on concurrency bugs and race detection [18,25,26,33,36,47,48,51,53]. The former typically checks one (server) program, while Argus targets desktop applications where the defect often involves user interaction events, daemons, external frameworks or other applications. The latter usually focuses on testing and eliminating bugs before software is released, while Argus focuses on helping developers diagnose performance issues in the wild. Argus also addresses performance issues caused by other types of bugs.

10 Conclusions and Future Work

Argus is the first comprehensive causal tracing system to diagnose performance anomalies in complex desktop applications. We observe that although causal tracing is powerful and extensively studied in distributed systems, it is brittle when applied to desktop systems due to inherent tracing inaccuracies. Argus addresses this problem by introducing annotated trace graphs with strong and weak edges to account for these inaccuracies. Argus pairs annotated trace graphs with a novel beam search diagnosis algorithm and subgraph comparison mechanism to determine causal paths in the presence of these inaccuracies. We have implemented Argus across multiple versions of macOS and evaluated its effectiveness on complex desktop applications. Argus successfully pinpoints the root causes for 12 real-world performance issues in these applications, many of which had remained open for several years. Argus imposes less than 5% CPU overhead, making it fast enough for regular use.

We believe Argus's strong and weak edge notions and inaccuracy-tolerant diagnosis algorithm may extend beyond the scope of desktop systems. In causal tracing of distributed systems, many solutions assume systems are perfectly instrumented, but in practice this is not the case. We plan to explore using Argus's techniques in the context of distributed systems as an area of future work.

Acknowledgments

We thank our shepherd, Pedro Fonseca, and the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CCF-1918400, CNS-1563555, CNS-1564055, CNS-1942794, CNS-1910133, and CCF-1918757, ONR grants N00014-16-1-2263 and N00014-17-1-2788, a JP Morgan Faculty Research Award, and a DiDi Faculty Research Award.

References

[1] Catapult : Chromium benchmark. https://chromium.googlesource.com/catapult.

[2] Chromium 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.

[3] Kernel probes (Kprobes). https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[4] LTTng: Linux tracing toolkit - next generation. https://lttng.org.

[5] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[6] Quiver: Crash when applying bullet points on multiple lines of text. https://github.com/HappenApps/Quiver/issues/21.

[7] Quiver: The programmer's notebook. https://happenapps.com.

[8] Sequel-Ace fix reconnect timeout - accept SSH password after network connection reset. https://github.com/Sequel-Ace/Sequel-Ace/pull/772.

[9] TeXstudio freezes when bib file is updated in the background. https://github.com/texstudio-org/texstudio/issues/288.

[10] Apple. Activity monitor user guide: Run system diagnostics in activity monitor on mac. https://support.apple.com/guide/activity-monitor/run-system-diagnostics-actmntr2225/mac.

[11] Apple. Cocoa fundamentals guide. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/Introduction.html.

[12] Apple. Instruments overview. https://help.apple.com/instruments/mac/current/#/dev7b09c84f5.

[13] Apple. trace: configure, record, and display kernel trace events. https://opensource.apple.com/source/system_cmds/system_cmds-671.10.3/trace.tproj.

[14] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 259 – 272, San Francisco, CA, USA, December 2004.

[15] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, page 468–479, Hyderabad, India, May 2014.

[16] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, Arvind Krishnamurthy, and Thomas E Anderson. Mining temporal invariants from partially ordered logs. In *Workshop on Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML'11)*, Cascais, Portugal, October 2011.

[17] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'11)*, pages 267–277, Szeged, Hungary, September 2011.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, page 209–224, San Diego, CA, USA, December 2008.

[19] Don Capps, Carol Capps, Darren Sawyer, Jerry Lohr, George Dowding, Gary Little, Capps Capps, Robin Miller, Sorin Faibish, Raymond Wang, Tanmay Waghmare, Yansheng Zhang, Vernon Miller, Nick Principe, Zach Jones, Udayan Bapat, William Norcott, Isom Crawford, Kirby Collins, Al Slater, Scott Rhine, Mike Wisner, Ken Goss, Steve Landherr, Brad Smith, Mark Kelly, Alain Dr. CYR, Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner, Jean-Marc Zucconi, Jeff Blomberg, Halevy. Benny, Dave Boone, Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root, Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer, Vangel Bojaxhi, Ben England, Lapa. Vikentsi, and Alexey Skidanoy. IOzone filesystem benchmark. https://www.iozone.org/.

[20] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'02)*, pages 595–604, Bethesda, MD, USA, June 2002.

[21] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design*

and Implementation (OSDI'04), pages 231–244, San Francisco, CA, USA, December 2004.

[22] Russell Coker. Bonnie++ benchmarking. https://www.coker.com.au/bonnie++/.

[23] Charlie Curtsinger and Emery D. Berger. COZ: Finding code that counts with causal profiling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*, pages 184–197, Monterey, CA, USA, October 2015.

[24] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, page 1285–1298, Dallas, TX, USA, October 2017.

[25] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*, pages 215–228, Salzburg, Austria, April 2011.

[26] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*, pages 221–230, Chicago, IL, USA, June 2010.

[27] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*, pages 271–284, Cambridge, MA, USA, April 2007.

[28] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN'82)*, page 120–126, Boston, MA, USA, June 1982.

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

[30] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, page 71–83, Cascais, Portugal, October 2011.

[31] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, USA, July 1999.

[32] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 34–50, Shanghai, China, October 2017.

[33] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 353–367, Cascais, Portugal, October 2011.

[34] Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*, pages 359–373, Philadelphia, PA, USA, June 2018.

[35] Ramón Medrano Llamas. iBench: The Cocoa Benchmark. https://ibench.sourceforge.io.

[36] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 329–339, Seattle, WA, USA, March 2008.

[37] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*, pages 1–18, Porto, Portugal, April 2018.

[38] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*, pages 378–393, Monterey, CA, USA, October 2015.

[39] Microsoft. Event tracing for windows. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-, 2002.

[40] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI'12), pages 107–120, Hollywood, CA, USA, October 2012.

[41] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, pages 115–128, San Jose, CA, USA, May 2006.

[42] Ali G. Saidi, Nathan L. Binkert, Steven K. Reinhardt, and Trevor Mudge. Full-system critical path analysis. In *Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and software (ISPASS'08)*, pages 63–74, Austin, TX, USA, April 2008.

[43] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. Principled workflow-centric tracing of distributed systems. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*, pages 401–414, Santa Clara, CA, USA, October 2016.

[44] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, April 2010.

[45] The LLDB Team. The LLDB Debugger. https://lldb.llvm.org/.

[46] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'06/Performance'06)*, pages 3–14, Saint Malo, France, June 2006.

[47] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 369–384, Cascais, Portugal, October 2011.

[48] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pages 135—149, Vancouver, BC, Canada, October 2010.

[49] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad Hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pages 163–176, Vancouver, BC, Canada, October 2010.

[50] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, pages 117–132, Big Sky, MT, USA, October 2009.

[51] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, pages 485–502, Tucson, AZ, USA, October 2012.

[52] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, pages 193–206, Salt Lake City, UT, USA, February 2014.

[53] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 221–234, Brighton United Kingdom, October 2005.

[54] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 293–306, Hollywood, CA, USA, October 2012.

[55] Benito van der Zander, Jan Sundermeyer, Danel Braun, and Tim Hoffmann. TeXstudio: LaTeX made comfortable. https://www.texstudio.org.

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