Title for Submission to Eurosys 2018

Paper #XX

NN pages

Abstract

MacOS is well known for its user experience. However, performance bugs still exist, as such, spinning cursors appear from time to time. Although the reason why it appears is straightforward based on its mechanism that the main UI thread is too busy to process user input, its root cause is still hard to diagnose. Tracking the performance bug where the lagging or unresponsiveness of applications stem, is hard due to the massive interactions among threads. Without instruments in the source code, it is challenging to detangle the user transactions running concurrently in the system, as well as the background heartbeat threads.

Fortunately, instruments in low level libraries and system could help developers to recognize interactions of threads by generating a dependency graph over the tracing data. A tracing point records a thread activity which we call it event. It contains an event name and attributes correlated to the event. The selected event either represents the execution boundary inside a thread on behalf of particular user input or reflects the potential connections with other events. The dependacy graph is dynamically generated with the execution segments divided by the boundary event as nodes and connections as edges. It is expected to help developers to figure out the critical paths of user transactions and enlight them on the inefficient code causing performance bugs.

However, without the in-depth understanding of a system, especially when it is not open source, the integrity and completeness of the dependency graph is hard to guarantee. The incorrect graph becomes less useful, even misleading, for the developers. In this paper, we focus on Apple's MacOS and address the challenge by inspecting the over connections and under connections in the dependency graph. The recognized wrong connection will give the developers guidance to further explore special programming paradigms in a less familiar system from various levels, including libraries, frameworks in the middleware and kernel. By adding tracing points in the proper locations to indicate boundaries and connections for the programming paradigms, the dependency graph gets improved accumulatively on correctness and hence usefulness in performance bug diagnosis.

1. Introduction

Performance bugs, usually cause significant performance degration, have vital impact on user experience. However, it is hard to diagnose for a knowledge gap between the symptom and the inefficient code for app developers. To identify causes of performance degreadation, the general idea shared by multiple previous work is to track the system activity and extract dependancy graphs for individual user transaction for further analysis. However, multithread programming are widely applied in modern systems. Under these circumstances, massive thread activities emerge in the system, from kernel, daemons, various applications and even the interleaved user transactions from the same application. Those activities in interleaved manner make it difficult to extract a correct and complete dependancy graph for a corresponding user input.

To address the challenge, some previous works in the area depend on the input from developers. Magpie is designed for distributed system to extract control path and resource demand of workloads from system wide traced events. It adds necessary tracing points in application, middleware and kernel by extending the tracing tool (Event Trace for Windows). A user provided event schema is used to correlate events from the same request. While Pip is an infrastructure developed to compare the expectation of an application with the actual behavior so as to reveal bugs of applications in distributed system. It designs a language for developers to describe their expectations of application behavior. An annotation library and a set of tools for gathering and checking the traced events. However, either the schema or the source code abstraction can be error prone. In that different components in a system, for example libraries, are usually developed by different developers, and even the worse, some of them can be closed source. Without the help from system authors, it is hard for the app developers to construct a complete and sound schema or expected application behavior.

There also exist works that do not need developers' inputs. AppInsight minimize the noise by only concentrating on the activities from the Application. It tracks an user transaction from the begin of user input to the end of UI update to diagnose performance bottlenecks and failures of apps on Windows Phone, including UI manipulation, Layout update, Asynchronous function call, begin and end of callback functions from framework into the app code (upcalls), thread synchronization and exception data. Without tracking daemons or system level activity, the dependancy graph may be less useful in pinpoint bugs residing in daemons or underlying frameworks. Panappticon instruments system wide with fine-grained tracing points and captures the correlations and causality of events across thread/process boundaries. Its usefulness highly depends upon the knowledge of Android property and the understanding of the whole system to define the correlations of temperally ordered events and the causality of events between threads. Hueristics that the locking primitive in the background thread indicated the producer/consumer of a task queue and no unrelated work will be performed while processing a particular task from a queue may be not true for other systems. The causality, deduced from asychronous call of MessageQueue and ThreadPoolExecutor, interprocess communication via Binder, and sychronization mechanisms, may be not complete either. To sum up, the integrity and completeness of dependancy graph is hard to guarantee without in-depth understanding of the whole system. Previous work from various platforms, such as Windows, Android and etc, could hardly directly applied to a new system, especially one with different design of programming paradigms.

Our work is a complementary part to the previous work. We proposed XXX to help the developer to detect the under-connection and over-connection in a dynamically generated dependancy graph, discover and explore the potentially missing vital tracing points to rectify the errors. To justify the integrity of the dependancy graph, we need to rule out the over connections. One simple way is to audit the path between every two processes in the graph. If two irrelevant applications appear in the same graph, there must be some over connections in their connecting path. Further exploring and fixing can be done by checking the nodes in the path. To improve the completeness of the dependancy graph, we need to inspect the under connections.

To prevent the dependancy graph from over-connection and under-connection, tracing points can be added either by hooking dynamic libraries or by setting hardware watch pointers.

We apply the method on the application running on MacOS and figure out both false negtive(under-connection) and false positive(over-connection) with the tranditional tracking technique.

There are missing thread asynchronization mechanisms. False negatives we detected:

- Timers can be armed to delay the event processing.
- Grand Central Dispatcher is frequently referred by the high level frameworks to schedule tasks.
- Daemons like WindowServer may postpone event processing via shared variable.
- Rendering with layer can be batched for later with flags in layer objects.

False positive we detected:

- Runloop is a more complicated programing model than MessageQueue or ThreadExecutor. Dividing it merely depending on the user event would result in false connection of thread activities in temporal order.
- Some kernel_task threads in MacOS running as heartbeat thread, can always introduce over connections.
- One block from dispatch queue, invoked with dispatch_mig_server, could serve for multiple unrelated work.
- Daemons like WindowServer will also serve for different work in interleave manner to support postponable requests.

2. Design and Implementation

The application running in MacOS system is compromised of various components. For propagating events to target application, WindowServer will be touched. An user input originated from the attached device will go through the I/O Kit to the WindowServer's event queue first. The WindowServer dispatches the event to the appropriate thread in target process. After that, the event will be passed the main UI thread and invok the corresponding event-handling routin. The transition of event from Windowser to the main UI thread requires additional preprocessing in the Cocoa Framworks. The event-handling ususally act in asynchronous manner to prevent the block in the main thread. In this procedure, daemons such as fontd, md, mdstore, prefersd, notifyd and services like SandboxHelper and XPCService will get involved. As a result, an user input will trigger system wide thread activities.

Similar scenarios happen in Windows and Android. Previous work on WindowServer from Microsoft Research use its own ETW(event tracing for windows) to record flow of control transfer between components in application and middleware system wide, leveraging their advantage of Windows design knowledge as well as developer input schema. AppInsight rewrites the app bytecode and instruments it leveraging the upcalls from the high-level framework into the app code for various reason like to handle user input, spawning of worker threads, sensor trigger and so on. It captures events including UI manipulation, thread execution, asynchronous calls, thread synchronizaiton, UI updates and unhandled exception, but limited to the app wide. The work on Android also collects limited event categories, with the trade off between the coverage and overhead, including user input, asynchronous calls (via its well known MessageQueue and ThreadPoolexecutor mechenisms), IPC (Binder RPC), selected synchronization mechnism, display update via view Class and recource counting system wide.

We share the goals of the previuos work on helping developers understand the performance bugs in the wild, but we also help the developer to rectify the framework for special programming paradigms that they are not aware of.

The archtecture of XXX is shown in figure XXX. Three components are included: Instrumenter, Analyzer and Inspectors. The instrumenter is responsible to add tracing points in kernel, dynamic libraries and middleware frameworks. Tracing points in the kernel can be added by modifying the kernel directly, while as the dynamic libraries and middleware frameworks are only partially opensource, we need to leverage the reverse engine tools and instrument the binary code. We replace the library by hooking one exposed symbol which will call a detour function and reexporting all the rest symbols. Indise the detour function, tracing points can be added to any address by replacing instrutions with tempoline function. Analyzer will collecte attributes from tracing points for events, which help events to connect or to differentiate execution segments. With the identified execution boundary and connections, it generates dependancy graphs for user inputs. Inspectors are used to check the dynamic generatged dependancy graph with respect to the correctness and completeness. Although it is hare to achieve the goal of completeness, we are expected to achieve the certain degree where the dependancy graph becomes useful for disganosis purpose.

3. Instrumentation

Considering the trade off between userfulness and overhead, we first choose the well known events that previous work chosen. We instrument the Cocoa Unbrella Framework for user input and display update. Ueser input: user inputs from the I/O kit to the target application will refer the Umbrealla Frameworks HIToolbox and AppKit. User inputs like key stroke and mouse move may get collapse in queues. We trace the user inputs in the AppKit framework as it was done in previous work. Display update: we not only record the updates but also trace the code locations where the need_display flags are set. With this method, we track both the display updates with View class and CALayer class. Asynchrouns call: asynchrous calls in MacOS can be implemented via RunLoop object, Grand Central Dispatcher, timer and wait queue. IPC: Mach message is the main mechanism for IPC. Thread synchronization: We also trace the thread synchronization by tracking the thread scheduling event, including to wait and to be made runnable, which has better coverage. To provide hints for developers to exploring programming paradigms, we also add backtrace after every mach message and dispatched function execution. We also trace certain events to exclude the obvious background noise introduced by hearbeat threads. The heartbeat threads in the system are identified with the interruption the running thread itermittently. We address the interrupts and time share maintanence by isolate the event and the thread activites that it triggers from the current running thread activities. Other potential heartbeat theread activities may relate to programing paradigms are left later for inspections.

4. Parse and Analysis

Unlike AppInsight which tracks the begin and end of function calls, like edge triggered tracing, we only track particular events and attributes in the states, which is level triggered tracing. As worker threads can be reused on behalf of different user inputs or processsing different requests from various applications in the system concurrently, identifying the execution boundaries is important. In addition to the known events that can indicated the boundaries, some potential boundaries may be covered by the unknown programming paradigms.

5. Dependancy Graph Inspect

6. Case Study

7. Related Work