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, for example, a piece of inefficient code, is still hard to diagnose. Tracking the performance bug where the lagging or unresponsiveness of applications stem, is complicated due to the massive interactions among threads, such as the user transactions running concurrently in the system and background daemons. Lack of source code makes the tracking even more challenging.

Fortunately, the knowledge of the implementation of programming paradigms lessens the difficulty to some degree. Instrumentation in middle-ware and kenel with the knowledge could help developers to recognize the relationships among threads temporally. Thread activies and relationships for one user transaction provides the developer a dependency graph for further disgnosis. 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.

In this paper, we focus on Apple's MacOS and address the challenge from a system level. As is in other systems, the thread activity is traced with a name that stands for its functionality and atrributes related to it in some execution point, which is called an event. The events traced either represent the execution boundary inside a thread for a request, or record potential connections with other events. With these events,the dependency graph can be generated with the execution segments divided by the boundary event as nodes and connections of events in execution segmenst as edges. As it is not straightforward to achieve the goal of integrity and completeness, inspectors are provided for developers to discover the over connections and under connections. The recognized wrong connections give the developers guidance to further explore special programming paradigms in a less familiar system. We further provides a method for binary instrumentaion to add tracing events in proper binary code to uncover the missing boundaries and connections. As a result, developers can rectify the dependency graph with knowledge accumulatively and use it to figure out the inefficient code causing performance bugs.

1. Introduction

Performance bugs, usually cause significant performance degradation, have vital impact on user experience. However, it is hard to diagnose because of the 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 thread activities and extract dependency graphs for individual user transaction for further analysis. However, with multithread programming widely applied in modern systems, massive thread activities emerge in the system, from kernel, daemons, various applications and even interleaved user transactions from the same application. Those activities in interleaved manner make it difficult to extract a correct and complete dependency graph for an individual 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. Pip is an infrastructure developed to compare the expectation of an application with the actual behavior collected in system 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 are provided 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 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 minimizes 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 activities, the dependency graph may be less useful on pinpointing bugs related to 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 temporally ordered events and the causality of events between threads. Heuristics 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 asynchronous call of MessageQueue and ThreadPoolExecutor, inter-process communication via Binder, and synchronization mechanisms, may be not complete either. To sum up, the integrity and completeness of dependency 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.

Base on previous work, we proposed XXX to help the developer to detect the over-connection and under-connection in a dynamically generated dependency graph, discover and explore the potentially missing vital tracing points to rectify the errors. To justify the integrity of the dependency 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 dependency graph, we need to inspect the under connections.

To fix the over-connection and under-connection in dependency graphs, tracing points can be added by hooking dynamic libraries or setting hardware watch pointers.

We apply the method on the application running on MacOS and figure out both false positive(over-connection) and false negative(under-connection) compared to the traditional tracking techniques. Most of them are caused by programming paradigms from the middleware.
False positive cases we detected:

- Loops are widely used when kernel or daemons process requests from different applications for the same service without transition events traced. For example, kernel thread will continuously wake up all the armed timers if they get fired at the same time. Another example is on MIG(Mach Interface Gnereator), which is used to compile procedural interfaces to the message-based APIs. The API dispatch_mig_service is designed to processing all the requests for the same service in a loop.
- Runloop is a more complicated programing model than MessageQueue or ThreadExecutor. Sources, observers, timers, dispatch queue and blocks are all checked inside one loop iteration. Dividing it into execution segments merely based on the user event would result in over connection.
- Mach messages send and receive from different applications can be packed inside one mach_msg_trap system call. For example, WindowServer will send out the pending message from one applicationa and try to receive message from another application in one mach_msg_trap system call.
- Some kernel_task threads in MacOS running as heartbeat thread, can always introduce over connections.

False negative cases we detected:

- Timers can be armed to delay the event processing. In previous work, a timer is treated as an individual transaction.
- Data dependency is hard and rarely tracked before, such as:

  WindowServer may postpone message sending via shared variable.

On screen rendering, changes from multiple layers can be commited independently and rendered in a batch later with need_display flag.

User inputs pulled from the WindowServer are pushed into the event queue of the main UI thread for later processing.

When a spining beachball should be drawn also depend on shared variable.

2. Design and Implementation

The application running in Mac OS system is comprised of various components. As an example, consider what happens when a user action is routed to target application. An user input originated from the attached device will go through the I/O Kit to the WindowServer event queue first. I/O Kit creates a low-level event and puts it in the WindowServer event queue. WindowServe processes it in various ways, time-stamps it, annotates it with the associated window and process port, and so on. After that the event manager thread from the target process will pull the event and pass it to the main thread's event queue. The main thread fetches the topmost event from the queue and dispatches it forward to the corresponding event-handling routine. The event-handling usually act in asynchronous manner to prevent the block in the main thread. In this procedure, daemons such as fontd, md, mdstore, cfprefsd, syslogd, notifyd and services like SandboxHelper and XPCService will get involved. As such, a user action will trigger system wide thread activities.

Similar scenarios happen in Windows and Android. Tools upon these system are developed to help developers to diagnose bugs by analyzing dependency graphs or critical paths. Magpie from Microsoft Research uses ETW(event tracing for windows) to record flow of control transfer system wide, between components in application and middleware, leveraging their advantages of Windows design knowledge as well as developer input schema. AppInsight rewrites the app bytecode for instrumentation leveraging the upcalls from the high-level framework into the app code for various reason, such as to handle user input, spawning of worker threads, sensor trigger inside the app. Panappticon on Android collects limited event categories based on its knowledge on Android property. It divides thread activities into execution segments based on the java thread pattern and heuristics for background thread to generate dependancy graphs. It is not trival to convert a tracking method in one system to the other though. We share the goals of these works on helping developers understand the performance bugs in the wild. What is more, our wok will help the developer to rectify the dependency graph generating framework for special programming paradigms that they are not even aware of when developing an application.

The architecture of XXX is shown in figure XXX. It contains instrumenter, analyzer, inspector and diagnosis tools. Instrumenter is responsible for adding tracing points in the correction location in kernel, dynamic libraries and middleware frameworks. For kernel, as it is open source, tracing points can be added into the source code directly, while dynamic libraries and middleware frameworks requires the help of reverse engineering tools and instrumenting the binary code because of close source. We reexport the symbols in the original library and hook the location where intrumentation is necessary by replacing the original instruction with a tempoline function, where tracing points reside and original instruction is sumilated.

Three steps are taken by Analyzer. First it will collect attributes for events as some event requires multiple tracing points to record its attributes. Second, the event list for a thread will be divided into execution segments with identified events as boundary. Third, execution segments from the second step are connected into a cluster with particular event pairs as connectors. So far, connectors in our frameworks includes Mach message send/receipt, timer armed/fired, dispatche blocks enqueue/dequeue, CALayer set_need_display/display, and selected wake_up and the first traced event in the woken thread. The generated cluster is expected to represent the dependancy graph for a user transaction and used for diagnosis. As is it hard to guarantee the integrity and completeness of dynamically genrated dependancy graphs, inspector is added for checking and improvement. The graph is expected to be correct and complete to some degree and becomes useful for disganosis purpose.

Finally, utils can be built to assist the developers for diagnosis purpose.

3. Instrumentation

Considering the trade off between userfulness and overhead, we first instrument and collect the well known events that previous work chosen.

- User input: user inputs to the target application will access the Umbreialla Frameworks includeing HIToolbox and AppKit. The data transfering from low-level device, I/O kit to WindowServer are ignored with the consideration of userfulness and overhead. Instead, how the user input events are fetched by the target application and dispatched to main thread are tracked in our framework.
- Display update: screen rendering has two ways in Mac OS, One is managed by the NSView object and the other is through the CALayer. The common part

the two methods share is to set the flag need_display in object, and call the method display in the object later. We track the object when its need_display flag is set and when its display method is called.

- 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. Mach message send and receive are tracked in the kernel.
- Share variables: shared variables are hard to explore. We only track the detected ones in our framework with hardware breakpoint registers. Inspector helps further to add more if necessary.
- Thread synchronization: We trace thread synchronization by tracking the thread scheduling events, wait event and wake-up event. Wait event is recorded before the thread blocking, and a wake-up event is added when a thread tries to make another thread runnable. Compared to previous work that tracks only semaphores and condition variables, this method has a better coverage.
- Heartbeat thread related noise: We find timeshare maintainance and certain interrupts happens peoriodically in the system. To exclude those noise, we need to trace them, isolate the event as well as the thread activites that it triggers and discard them. Other potential heartbeat thread activities that may be related to programing paradigms are left later for inspections.
- Other information: To provide hints for developers to explore programming paradigms, we also add backtrace tracing points, recording necessay memory mapping infomation once and making use of lldb for offline symbolization.

All the tracing points are recorded by extending the usage of Kernel Event Tracing (kdebug) facility provided by Apple. Tracing points can be added into the library, as mentioned in previous section, by replacing the original library and reexporting its symbols. They can also be added via hardware breakpointer register. Once the breakpoint traps in the particular memory location, the tracing point will be record inside the signal processing handler. We install the handler for hardware breakpoint in CoreFoundation which will be linked by all applications. Tracing data from the tracing points is written to memory buffer first and then get dumped to file system in kernel, and a trace tool in the user space is shipped to control the begin and end of tracing. By limiting the file size and treating file as a ring buffer for flushing data, we are able to run the trace tool in the background 24X7. Also, we record memory mapping of images for all running processes in the trace tool for later symbolication. Once the performance problem rises, we can disable the trace and analyze the tracing data collected in the file for problem diagnosis.

4. Analysis

We expect to generate a dependency graph, containing the process from user input, event handling, to screen rendering with the analyzer. Traced events in a thread are divided into execution segments which is an atomic unit in a thread belonging to one user request. Execution segments can be connected by the event pairs, for instance, mach message send and corresponding receive. The dependancy graph is defined with the execution segments as nodes, and the connections of the segments as edges. To achieve the goal of sound dependency graphs for user transactions, we have to identify the boundaries of execution segments and recognize connections of them if they are on the behalf of the same user transaction. However, both steps are knotty.

It is triky to define events that represent the end of a user request in a thread. Although the execution segment looks straightforward if we track every entry and exit of function calls like what AppInsight did, it is too expensive in system wide and impossible if system components are closed source. One may argue to divie the thread execution with as few events as possible. However, dividing the thread execution into too fine granularity probihits following connections and therefore hurts the completeness of dependency graph. Identifying execution segments with thread synchronization(wait event) is misleading in some situation. We notice that the execution of a function can be interrupted by the wait event, and one thread can be resused by different requests without any wait event. One concrete example is from kernel thread. It will wake up all threads whose timers get fired at the same time continuouly without any transitional event. As a result, not only the thread synchronization but also programming paradigms should be considered when identifying the boundary of execution segments.

Connection recognition also relies on the knowledge of thread synchronization and programming paradigms, but it is more complex. It becomes ambiguous for similar event sequences given different circumstance. Consider the scenario that thread A blocks on certain resource, thread B wakes up thread A later, and thread A resumes execution. Three execution segments are gnerated because of thread synchronization. We call them segment1, segment2 and segment3. If we assume thread A is from a daemon and it blocks to wait for more requests, segment1 and segment3 should not be connected in that they may be from different processes, while segment2 and segment3 should be connected. On the other hand, if thread A is waiting for I/O, we expect segment1 and segment3 from thread A should be connected while segment2 should be excluded from the connection.

There is no uniform rules to divide and connect execution segments considering different roles a thread may play. To make things practical, we need a general algorithm to identify the thread segements and their connections, and also a mechanism to figure out what is not reasonable and how to improve it, which we will disscuss in next section. Studying thread patterns provides hints for us to figure out the general algorithm. We recognize that some threads in the system are associated with RunLoop which is an event processing loop used to schedule work and cooordinate the receipt of incoming events. It will keep the thread busy when there is work to do and put it to sleep when there is none. The order how events get processed is very specific in the runloop according to the description on Apple official site.

- Notify observers that the run loop has been entered.
- Notify observers that any ready timers are about to fire.
- Notify observers that any input sources that are not port based are about to fire.
- Fire any non-port-based input sources that are ready to fire.
- If a port-based input source is ready and waiting to fire, process the event immediately. Go to step 9.
- Notify observers that the thread is about to sleep.
- Put the thread to sleep until one of the following events occurs:

  An event arrives for a port-based input source.

  A timer fires.

  The timeout value set for the run loop expires.

  The run loop is explicitly woken up.
- Notify observers that the thread just woke up.
- Process the pending event.

  If a user-defined timer fired, process the timer event and restart the loop. Go to step 2.

  If an input source fired, deliver the event.

  If the run loop was explicitly woken up but has not yet timed out, restart the loop. Go to step 2.
- Notify observers that the run loop has exited.

RunLoop is driven by an exteranl while or for loop provided by developers. The event processing handlers for them are also installed by developers. The well defined pattern helps us to identify execution boundary in threads.

To identify execution segment boundaries, we first classify thread into two categories: thread with RunLoop object and others. As thread with runloop infrastructure processes pending events and generates notifications for any attached observers, we can trace the thread execution stage by tracking the observers, including RunLoopEntry, RunLoopBeforeTimers, RunLoopBeforeSources, RunLoopBeforeWaiting, RunLoopAfterWaiting and RunLoopExit. Every RunLoopEntry is the beginning of an execution segment and others helps us to identifying what adhoc requests are going to process. For threads which are not equipped with runloop, we first apply rules extracted from the traditional programming paradigms. There are six rules generalized:

- dispatcher: GCD(grand central dispatcher) are widely applied in Apple, threads activity can be divided every time it dequeues a block from dispatch queue and executes it.
- thread synchronization: wait event means blocking for resource. We recoganize it as the end of execution segments in most case, except that the execution of a dispatched block has not yet finished.
- heartbeat thread isolation: thread execution can be interruptted by the heartbeat thread intermittently. We will isolate the interrupt and the activity it triggers into a new execution segment and come back to the original thread. The triggered activity is identified via the make_runnable event inside the interrupt event. Another periodically executed thread is time share maintainance. We apply the similar method to isolate it from the original thread.
- kernel thread for timers: kernel thread could interact with multiple user processes continuously. Timer management is one example. If a timer is fired, the kernel thread will send mach message to the processes who aremed the timer. As a result, every timer will be in a new group.
- IPC and vouchers: mach message is the main IPC mechanism in MacOS. Some message carries voucher, which means the message is on the behalf of a third party. For example, process A sends a message to Daemon B, and Daemon B sends a message on behalf of the received message from Process A with the voucher that records infomation on A. Mach messges that send/receive in the one thread do not always serve for the same request. Some daemons like WindowServer will pack the pending out message and currently message receipt into one system call. We check every mach message for its receiver or sender. If two continuous messages are sent/received to/from different processes, we first check if they carried vouchers that connect them together. If so, they go into the same execution segments, otherwise, a new segment is create for the later one.

- other special programming paradigms: If errors detect in the later process, we will examine the execution segments with callstacks, and add tracing poings to identify the new boundary introduced by the unrealized programming paradigms.

Part of those rules are also applied to thread with runLoop object as we also realize that the runloop is not dedicated to UI event processing. The draining of the main dispatch queue in runloop utilizes CGD to process requests submitted by other threads. Observers also provide a general means to receive callbacks, which developers can install if needed, at different points within a running run loop. Besides, scheduled asynchronous blocks on other threads is implemented by inserting blocks into an array. These blocks are processed in FIFO the next time the runloop calls out to blocks.

The second step towards the dependency graph generation is to discover the connections between execution segments. Execution segments that contain pairs of events representing known programming paradigms will be connected. We check

- Mach IPC, segments with messages that send and receive respectively will get connected. Besides, if a mach message requires a reply, it will carry a port in its message header. However, the reply message is not always sent by the thread that received the message. As a result, we also connect the message receipt and the following reply message sent. Messages are paired mainly on the slided kernel addresses of the ports.
- GCD, for asynchronous function calls, we observe blocks get enqueued, dequeued and executed by different threads in most cases. They are scatted in different execution segments. We connected them with the block address captured when they are enqueued, dequeued and executed.
- Timer, timer is another asynchronous means to delay function execution. We will record the address of timer item in kernel when it get push to a queue, fired and cancelled. The segment that push the timer to q queue and segment that get woken because it get fired are connected.
- Screen updates, we will record the object address when the view get touched and set need_display and when the display update happens, and connecte them together.
- Shared variables, variables can be used to synchronize the threads. The event thread that pulls events from WindowServer and the main thread that dispatches event to event handler coordinate with variable. By recording every time the variable written, we can pair when the event is put by the event thread and when it is fetched by the main thread. Besides, we also track the variable related to the postponed message ready and sent in WindowServer and variable related to the spinning beachcall mechanism.
- Thread schedule, we record the relationship between two threads that one thread wakes up the other threads. However, the two threads are not always get connected depending on the wake up reason. For example, if two threads are all worker threads, and the former finished a task from dispatch queue and wake up the later thread to process the next item, we will not connected them together because the two tasks in the same dispatch queue are not neccesarily correlated. We picked a conservertive way to connect two segments that have the wake-up relationship and put it after all other connections are finished.

All of the necessary connections are recorded in the data structures of traced event, so that the peers of the connection can be easily reached from one to the other.

The final step is to pick a root segment and augment it into a dependency graph. The dependency graph begins from the user input according to our goal and definition. We first check the main UI thread, and initiate the graph with the execution segment containing user input as a root. Then we check all of the connections except wake-up connector in the execution segment, and add execution segments where the peers resides until there is no more can be added. After this process, we will check all the segments that contain wait event. If a segment has following segments in time order from the same thread in the cluster, we will include the execution segment from another thread that contains the wake-up event to make the former thread runnable. Compared with a more aggresive method to augment the cluster with the wake-up events, this method will exclude the case that a thread try to wake up multiple processes that are not related to each other for timeout reason. However, the risk is if a daemon is used to process a request but no reply is need to send back, the execution segment from the daemon may be ignored. Althougt it will affect the completeness of the dependency graph, it may not interrupt the diagnosis of performance bug as the current request will not block by the daemon.

5. Dependancy Graph Inspect

We implemented the inspector with two concerns. The first one is the over-connection. To eliminate the over connection, the inspector will traverse the dependency graph to find the possible paths that connects any two processes. If any path of two processes only connects via daemons like WindowServer, they are reported as the suspicious over-connection. In most cases, at least one of the processes is noise introduced by WindowServer. In our studies, we found this is because WinsowServer will broadcast certain events and wake up a bunch of processes in the system inside one execution segment. The daemon fontd will also connects two unrelated processes because of dispatch_mig_server, which serves multiple processes continously in a while loop by listensing to a particular port.

The second concern is on the under-connection. Base on the general sequences of user input processing, we will check the dependency graph to see if the path includes nodes stands for user input, event processing and screen updates complete. As we use user input as the root of the dependency graph, it is always present. Updates can be checked via the traced events which represent the display of CALayer objects.

6. Case Study

We set up the case studies in the virtual machine running El Captain. The kernel version is 10.11.6. Kernel is recompiled with tracing points in addition to the ones provided by Apple. Frameworks and Libraries are modified by detouring an instruction and reexporting the original symbols. User space tool are provided to set hardware breakpointer. We studied XXX cases of performance bugs.

6.1 Spinning Beachballs

The applications that throws spinning beachball is one main category of performance bugs. The indicator is clear for user that the main ui thread fails to response, but without analysis tool, it is still hard to tell what cause it unresponsive. Base on our understanding of the spinning beachball mechanism, a shared variable "isCGEventIsMainThreadSpinning" is referenced by main thread and event thread in an application to indicate the state of the main thread. Another variable "isDispatchedToMainThread" is set every time the event thread passes an event, user input or system defined, to main thread, while the main thread will clear it every time it fetches an event from its queue. If the event thread find the last time dispatched event is cleared, it will pass the event and re-arm a timer to monitor the waiting time of current event. Once the timer fired, it means that the main thread has not fetched new event. It will set "isCGEventIsMainThreadSpinning" and notify the WindowServer to change the appearance of the cursor to a spinning beachball.

6.1.1 Google Chrome

Google Chrome throws spinning beachball in a sequence of user inputs. We can reproduce it in the following steps. Switch the input method to Simplified Chinese to get prepared. Open a new tab in the Chrome and go to yahoo.com. Before the page get fully loaded, type any Chinese characters rapidly. Usually the spinning cursor appears in the webpage if it gets the focus.

To firgure out the cause of beachball cursor, we've tracked the shared variable "isCGEventIsMainThreadSpinning" with hardware breakpoint register. In our generated dependency graph, we first use the search engine to position where the main thread is set spinning by checking the value of "isCGEventIsMainThreadSpinning". It narrows down the end boundary of root cause. Then the search engine will search up towards the timeline to get the last time "isDispatchedToMainThread" get cleared, from which we will get the beginning boundary of root cause. With the boundary, we can analyze time span of each execution segments and the time cost of wait event in the main thread. In this particular case, we find that the wait event in the main thread, from BSD system call psynch_cvwait with a cost of one second, repeats twice. The return value of psynch_cvwait indicates timeout happened. This is the straitforward reason of the spinning. We further search threads that touch the condition variable with corresponding system calls, psynch_cvbroad or psynch_cvsignal. The result shows that a worker thread that invokes psynch_cvsignal intermittently does not call it any more, while the main thread still tries to wait for condition variable.

6.1.2 Notes

A second case is quite common in text editing softwares, including TextEdit, the serial softwares of Microsoft Office, Latex and so on. In those softwares when you edit a large text, you will encounter the beachball. Examples like copying, pasting and changing the font size or style will trigger the unresponsiveness of the applications. Sometimes, only a small amount edit can also trigger.

We show the study of Notes app from Apple here. Our experiments on other softwares gives the similar results. We apply the similar serching method as the Google Chrome case to narrow down the search range of execution segments in the main thread. In this case, we could not find the outstanding time spent on wait, but we found an execution segment spans around 8 seconds. Inside the segment, we discoverred remarkable frequency of interrupts, including timer interrupts and inter processor interrupts. With the symbolication of intruction pointer recorded in traced interrupt event, the main thread is busy on memory operations when the interrupts happen. Besides, from the backtrace, the main thred is also busy preparing the changes in the layouts and setting the need_dispaly flags in NSView objects.

6.1.3 System Preference

A beachball will appear in the System preference setting panel when the application DisableMonitor is running in the system. DisableMonitor is an application used to manage external monitors for pc. The beachball appears when we use the DisableMonitor to close the external connected monitor first, and then try to arrange the displayers in System Preferences. Applying the tool for diagnosis, we find the most frequently appears API is thread_switch in the long execution segment of main thread.

6.1.4 Installer

Spinning beachball appears during the process of application installation. When there is a small input box waiting for password and you move the cursor over the underlying installer box, the spinning beachball comes out on the screen. However the installer is not an UI application, and the main thread and event thread do not exist. Cases like this are beyond out ability to diagnosis, in that who triggers the beachball is unknown. The variables used to indicated beachball for the unknown process is not monitored, so the search engine can not be applied. One solution is to symbolize all of the processes running in current system and make use of the call stack to find the process that causes beachball. After that, the case should be reproduced for study.

6.2 Comparative performance degradation

6.3 Discussion

The limitation of the diagnosis are confined inside the Application. However, the case like Google Chome must be related to the input method because the English input can not trigger beachball.

7. Related Work

Works on performance bug diagnosis usually fall into the following categories. The most naive way is profiling. The souce code should be accesible and it is of high ovehead and less practical especially when the performance bug that appears intermittentely. The second category makes us of the statistic technique or machine learning. It is useful in identify the possible performance bug. However, it is not hard to tell the root cause of the performance bugs. The second category is on the analysis of the dynamically collected data and examine the dependancy graph. The limitation is the method may work well in the system that are fully understanded by the tool developers. Our work complements this category of works in unfamiliar systems.