Dynamic Detection of Event Handlers

Steven P. Reiss

Department of Computer Science
Brown University
Providence, RI. 02912

spr@cs.brown.edu

ABSTRACT

This paper considers the problem of dynamically finding event handlers in a running application using information obtained from periodic stack samples. Knowing the set of event handlers in an application is a prerequisite to building a model of the event behavior of the application which is in turn needed to do performance analysis, program visualization, or program understanding in terms of events. We show that a trie-based statistical technique can effectively and accurately find event handlers.

Categories and Subject Descriptors

D2.5 [**Testing and Debugging**]: Monitors, debugging aids.

General Terms

Performance, Experimentation.

Keywords

Event handler, dynamic instrumentation, trace analysis, monitoring.

1. INTRODUCTION

Many of today's applications are event-driven. Standard desktop applications wait for a user action and then process the corresponding event. Servers listen on multiple client connections, waiting for input requests. When these requests arrive, they process them as events and generate the appropriate output. Inside the servers, requests are broken up into tasks that are serviced as events by thread pools and task queues.

Attempting to analyze such applications requires understanding their basic structure in terms of event processing and then doing the analysis using that structure. For example, the interesting performance problems in a modern server more often relate not to its overall behavior but to the resources used in processing particular requests.

Moreover, the programmer is often not interested in the overall processing done by the application, but rather in specific behaviors. For example, it is sometimes the case that while most events are processed quickly, occasionally some event will take significantly longer. Here the programmer is interested in identifying this event and understanding the resources that its processing required. Without analyzing the performance in terms of events, such behavior is very difficult to ascertain because the more common event behavior will dominate the global statistics.

In order to enable event-based analysis and understanding of applications, one needs to understand the structure of event processing in such applications. Because of the widely varying nature of the types of events and the ways that events are handled, current and prior work in this area has required that the programmer define this structure for the application. This can be a significant amount of work.

We are currently working on a performance analysis framework for long-running applications. As part of this framework we want to enable event-based analysis. Similarly, we are working a framework for application-specific visualizations that requires understanding event behavior. In both cases we need to work with arbitrary applications while requiring a minimum of work on the part of the programmer. Given this, we set out to determine if we can dynamically detect the event structure of an application. While this might be possible to ascertain through static analysis, such analysis would assume that one has all the available libraries and relevant code to analyze, that one understood the behavior of native library routines (such as Java's X11 event handler), and would not necessarily be precise. We felt that dynamic analysis would be more reliable and accurate and would better fit the type of dynamic monitoring we were doing.

In this paper we show that it is possible to accurately detect event handling routines on the fly through appropriate dynamic analysis. In the next section we look at related work and provide the details of the performance analysis framework that are relevant to our scheme for determining the event structure of an application. Section 3 provides a more complete definition of event handling and what we are looking for in terms of a model. Section 4 and Section 5 then describe the methodology we use, while Section 6 describes the results we have obtained.

2. PRIOR WORK

Traditional methods of performance analysis look at where the overall system spends resources such as execution time, real time, or memory. They assign resources to particular routines or lines of source code. For example, the UNIX *prof* tool uses sampling techniques to estimate how much time is spent in each routine over the full run of the program. More sophisti-

WODA -- Workshop on Dynamic Analysis, July 21, 2008.
Copyright 2008 ACM 978-1-60558-054-8/08/07...$5.00.

cated methods, such as *gprof* [7], assign resources based on two-level calls, that is, instead of looking at all instances of routine R, they separate these instances based on the different callers of R. Other tools such as HP's *CxPerf* provide a per-thread analysis of resource utilization. Still other tools look at specific items in a complex system. For example, *Tmon* [8] and *theadmon* [3] look at the behavior and interaction of threads in a multithreaded system. There have also been a variety of frameworks for customizing performance analysis [1,6], but these have only been used to do selective overall evaluation rather than looking at the behavior of individual actions.

Some of the more recent performance tools provide a foundation for event-based analysis but do so by putting the burden on the user. These tools let the user tell the program when to start collecting information and when to stop collecting it. They then let the user browse over the detailed performance information that was collected. For example, *Jinsight* lets the user specify a method as a trigger [9,10]. Trace collection occurs while the method is being executed. The user can then use *Jinsight* to browse over the dynamic call tree, objects allocated, etc. *JProbe* lets the programmer uses triggers to determine performance differences between releases and lets the user specify what parts of the program should be analyzed in detail. Borland's *OptimizeIt* lets you see performance by thread, and then examine the dynamic call graph to examine the behavior of individual events. Wily's *Interscope* again collects enough data to let the programmer specify filters and browse the dynamic call graph down to the method level so that an individual event can be understood.

Our prior work on event-based trace analysis required the user to specify the event model by providing an XML file that defined the event handling routines [12]. This model was automatically extended to track events throughout the execution by tracking objects within the trace. Our experience here was that this required a significant amount of work both in understanding the system libraries and their behavior and in recalling and understanding the application's behavior. The result was error-prone, requiring the programmer to iteratively add to or modify the XML file based on the output to attempt to get the correct set of events. Moreover, the set of system events was both large and incomplete, requiring updates for each new version of the system, each new system library, and each new platform since event handling is sometimes handled in machine-dependent code.

More recently, the X-Trace facility traces events through multiple layers of applications and allows the recording of associated behaviors such as time spent [4]. This system has the disadvantage that the programmer has to document the event model used by the application by inserting code into the application. The understanding and documentation of this model is a significant obstacle to using the facility.

We have developed a general framework, DYMON, for performance analysis of long-running applications [14]. The initial focus of this system is doing detailed performance analysis within a fixed, user-settable overhead. The system is organized using a set of agents, each of which looks at a particular aspect of performance, for example CPU time, I/O behavior, or heap utilization. The monitor takes periodic stack samples (using a rate based on the allowable overhead) and passes these to each active agent for analysis. Based on their analysis, agents can then request more detailed monitoring for a fixed amount of time. For example, the CPU agent will request a fifteen second sample of basic block counts for code that appears to be heavily used based on the stack samples.

We want to use this framework to provide an event-based analysis of performance. However, doing so requires that the system understand how events are handled in the application so that resources could be allocated to events. This in turn requires that we first identify how events are processed in the application. The DYMON framework works directly from the running binary of the application, so we needed a dynamic technique. Moreover, to fit in the DYMON framework, we needed to structure our solution as an agent that can build the event-handling model from the information gleamed from occasional stack samples.

3. PROBLEM DEFINITION

The first specific problem we needed to address was how to identify event handlers dynamically using only the information from periodic stack samples. In order to do this, one must first understand exactly what an event handler is.

An event handler is defined in Wikipedia as "an asynchronous callback subroutine that handles inputs received in a program". Similarly, an event is defined as "an action which can be initiated either by the user, a device such as a timer or keyboard, or even by the operating system". To find event handlers in the code then, we need to find routines that are called when an event occurs where an event is characterized as something being read, something being waited for, or something derived from the operating system.

In practical terms, to find event handlers dynamically one needs to understand how they appear in the code. In general, an event based application has code that looks something like:

```
LOOP
    Wait for the next event
    Process that event
END
```

The problem is that this code can occur in a multitude of different guises, with each of the lines of pseudo code being replaced by large blocks of code, multiple, possibly nested routines, additional processing, and hidden (low-level or operating system) code.

For example, I/O events might be handled by a routine that looks like:

```
LOOP
    r = ReadNextMessage()
    Process(r);
END
```

where the actual waiting and reading of the next message-based event is handled by the routine *ReadNextMessage* while the *Process* routine is actually the event handler.

Alternatively, there might be multiple routines processing routines called.

A work queue might be maintained with code that looks like:

```
LOOP
    SYNCHRONIZED
        while (Queue is empty) {
            wait();
        }
        e = Remove front of queue;
    END
    Process(e);
END
```

Here, the processing might not be as simple as a function call; it could involve in-line code that determines the type of event and then branches accordingly.

Another type of event handler is based on callbacks. For example most user interfaces work by having the user register a call back routine during the set up of the user interface. Then, when the user interface code detects an appropriate event, it calls the previously registered routines. This is generalized in the publish-subscribe pattern [5], and some applications use it for handling messages [11] and other program events.

Event handling is further complicated by the fact that it typically involves a mixture of user and system code. The I/O processing, waiting for events to be placed on a queue, or analyzing inputs to generate appropriate events are all sometimes done in user code, sometimes in system code or system libraries, and sometimes in a combination of the two.

The first step in building the event model for an application is determining which application routines correspond to the processing of an event. These are the various *Process* calls in the above code and their generalizations as well as routines that are invoked using callbacks. Here we want to restrict ourselves to application routines and ignore any system processing of events.

We find these routines in two ways. First, callbacks are relatively straightforward to detect since they represent user code fragments that are called from the operating system or library code rather than vice versa. Non-callback routines are more difficult. Here we need to identify locations in the code corresponding to waiting for an event to happen, either by doing a blocking I/O operation, by suitable polling, or using an appropriate wait loop. Then we have to identify the processing routines that correspond to the associated events.

Both of these detection mechanisms build on the foundation provided by DYMON. DYMON provides for a suite of agents, each of which addresses a particular performance issues. Our goal was to define an agent that would identify the appropriate event structure by first identifying all the event handling routines in the code. As a DYMON agent, the implementation needs to be in two parts. The first part receives periodic stack samples to collect the necessary information. It uses DYMON to send this information back to the second part which does appropriate data analysis and identifies the event handlers. These two parts are described in the next two sections.

4. DATA COLLECTION

The back end or our event structure agent is in charge of collecting and summarizing the information needed to actually identify event handlers.

The input to the back end is a set of stack samples, one for each active thread in the application given, provided periodically at a rate dependent on the allowable overhead. Each sample identifies the routine that is currently active in that thread and the set of routines that are calling that routine. This is provided by the Java Management Framework calls that get the current set of threads and then for each thread get the current stack trace as an array of *StackTraceElement* objects. Each of these objects holds the class name, method name, and line number of the corresponding stack frame. DYMON augments these objects with information as to whether each routine on the stack is from the user's application or from a system library routine, and, if it is a system library routine, whether that routine represents an input/output operation, a sleep, or a wait.

We first note that some event handling can be obtained directly from the information contained in a stack trace. A callback is going to involve system code that calls user code. Because the routines on the stacks are tagged as user or system, we can readily identify user routines that are called from system code. For each such routine we keep a count of the number of times it is called and return the set of such routines as potential callbacks.

Identifying callbacks that are based on I/O or waits called directly by user code is more difficult. Because the stack samples only provide snapshots of the execution, they don't provide direct evidence for event processing. Instead, one has to accumulate a significant number of stack samples and build a model of execution from which one can determine which routines correspond to waits or input/output and whether these routines are actually used for event processing and, if so, which routines are the corresponding event handlers.

Our data collection agent builds such a model by constructing a trie that accumulates the relevant part of all the stacks from all the stack traces. Assuming that the top of the stack trace is the main routine for the corresponding thread and the bottom is the currently executing routine, the first step we take is to edit the stack traces by removing all system routines at the trace bottom so that the either the new bottom of the stack trace is an application routine or the stack trace is empty. Next we assign a state to the bottom entry of the edited stack trace based on what the system was doing in the currently executing routine in the original trace. This state is one of WAIT if the system was waiting, IO if the system was doing an input or output operation, or RUN if the system was executing. Note here that waiting implies an explicit call to a wait routine and does not reflect threads that are blocked from executing due to synchronization.

From these edited stack traces we build a trie which reflects the whole set of stacks. The first level of the trie corresponds to the set of routines that occur at the start of any sample stack. The second level of the trie then corresponds to any routine called from a routine at the top level. This is iterated down to the leaves of the trie which

```
<REACTIONS LAST='1206456881022' MONTIME='350731' SAMPLES='6696' TSAMPLES='22904'>
  <CALLBACK STACK='6660' USER='solardraw.SolarDrawImpl@display' />
  <TRIE>
    <TRIENODE CLASS='solar.SolarMain' IO='0' METHOD='main' RUN='0' WAIT='0'>
      <TRIENODE CLASS='solar.SolarSystem' IO='0' METHOD='loadFile' RUN='1' WAIT='0'>
        <TRIENODE CLASS='solar.SolarBaseObject' IO='0' METHOD='&lt;init&gt;' RUN='0' WAIT='0'>
          <TRIENODE CLASS='solar.SolarVector' IO='0' METHOD='&lt;init&gt;' RUN='14' WAIT='0' />
          <TRIENODE CLASS='edu.brown.cs.ivy.xml.IvyXml' IO='0' METHOD='getAttrString' RUN='1' WAIT='
          <TRIENODE CLASS='edu.brown.cs.ivy.xml.IvyXml' IO='0' METHOD='getAttrDouble' RUN='4' WAIT='
          <TRIENODE CLASS='edu.brown.cs.ivy.xml.IvyXml' IO='0' METHOD='getElementByTag' RUN='1' WAIT
        </TRIENODE>
      </TRIENODE>
      <TRIENODE CLASS='solar.SolarSystem' IO='0' METHOD='cycle' RUN='0' WAIT='0'>
        <TRIENODE CLASS='solar.SolarGroupObject' IO='0' METHOD='getObjectCount' RUN='1' WAIT='0'>
          <TRIENODE CLASS='solar.SolarGroupObject' IO='0' METHOD='computeGroup' RUN='1' WAIT='0' />
        </TRIENODE>
        <TRIENODE CLASS='solar.SolarRootObject' IO='0' METHOD='computeLocalGravity' RUN='0' WAIT='0'
          <TRIENODE CLASS='solar.SolarRootObject' IO='0' METHOD='rebuildTree' RUN='38' WAIT='0' />
          <TRIENODE CLASS='solar.SolarGroupObject' IO='0' METHOD='computeLocalGravity' RUN='10' WAIT
          <TRIENODE CLASS='solar.SolarSystem' IO='0' METHOD='waitForRoot' RUN='0' WAIT='6336' />
        </TRIENODE>
        <TRIENODE CLASS='solar.SolarRootObject' IO='0' METHOD='finishCycle' RUN='0' WAIT='0'>
          <TRIENODE CLASS='solar.SolarGroupObject' IO='7' METHOD='finishCycle' RUN='119' WAIT='0' />
          <TRIENODE CLASS='solar.SolarGroupObject' IO='0' METHOD='checkPositions' RUN='156' WAIT='0'
          <TRIENODE CLASS='solar.SolarRootObject' IO='0' METHOD='migrateObject' RUN='1' WAIT='0' />
        </TRIENODE>
      </TRIENODE>
    </TRIENODE>
    <TRIENODE CLASS='solar.SolarSystem$WorkerThread' IO='0' METHOD='run' RUN='18' WAIT='10864'>
      <TRIENODE CLASS='solar.SolarSystem$GravityRequest' IO='0' METHOD='perform' RUN='0' WAIT='0'>
        <TRIENODE CLASS='solar.SolarGroupObject' IO='0' METHOD='computeLocalGravity' RUN='6' WAIT='0
          <TRIENODE CLASS='solar.SolarGroupObject' IO='0' METHOD='computeLocalGravity' RUN='1032' WA
          <TRIENODE CLASS='solar.SolarSystem' IO='0' METHOD='queueComputeLocalGravity' RUN='11' WAIT
        </TRIENODE>
      </TRIENODE>
      <TRIENODE CLASS='solar.SolarSystem' IO='0' METHOD='access$100' RUN='5' WAIT='0'>
        <TRIENODE CLASS='solar.SolarSystem' IO='0' METHOD='handleDone' RUN='73' WAIT='0'>
          <TRIENODE CLASS='solar.SolarSystem$Finisher' IO='0' METHOD='noteDone' RUN='7999' WAIT='0'
        </TRIENODE>
      </TRIENODE>
    </TRIENODE>
    <TRIENODE CLASS='solardraw.SolarDrawImpl$DrawThread' IO='0' METHOD='run' RUN='0' WAIT='6665' />
  </TRIE>
</REACTIONS>
```

Figure 1. Output from the back end agent showing callbacks and the trie.

are the set of routines that represent the currently executing application routine at the time the stack was sampled. Each node of the trie is then augmented with state counters that indicate how many times that node reflected the bottom of an edited stack trace that was in one of the states WAIT, IO, or RUN.

This trie represents a relatively compact summary of the global behavior of all the threads. The counters in the trie provide a statistical indication of what type of processing is done at the various nodes and how frequently each type of processing is done. The latter is needed because even nodes that are waiting might be not always be in a wait state when the stack is sampled and routines that are actually event handlers might be do internal I/O part of the time and execute other parts of the time. A sample trie as output by the data collection agent is shown in Figure 1.

The concept of accumulating stack samples has been used before for performance analysis and program understanding. For example, STAT analyzes stack traces from large numbers of processes to build a statistical model of what the processes are doing and then cluster these according to their behavior, thereby reducing the problem space for performance debugging [2]. CosmOpen accumulates stack traces obtained when specific routines are called to build a model of program behavior for reverse engineering [15].

5. DATA ANALYSIS

The front end of the DYMON agent is charged with analyzing the collected data and identifying the event handling routines. Callback routines for system events are found directly by the back end agent and are reported directly. The main work of the front end is to analyze the information in the trie to identify event handlers based on either I/O or waiting.

The front end operates in stages. It first determines for each node of the trie, what type of processing is represented by that node. Here the agent determines if the node represents program execution (RUN), program input or output (IO), waiting (WAIT), undetermined (ANY), or some combination (MIXED).

In order to determine the trie node type, it first determines if the node has been sampled enough times for the returned statistics to be meaningful. By looking at a variety of applications, we experimentally determined that this should be at least 10 samples and at least 0.0001 of

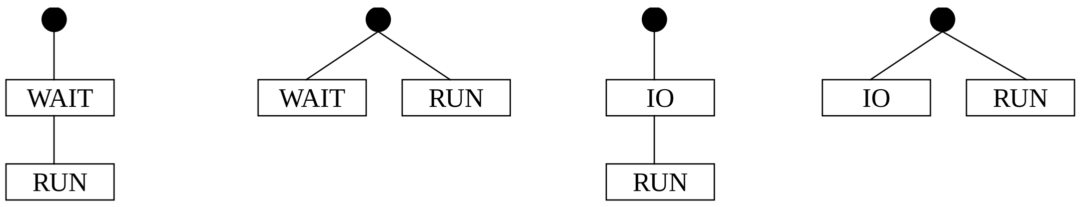

Figure 2. Trie patterns indicative of event handling

the total samples collected. The absolute minimum forces the analysis to wait until the system has been sampled often enough. The relative minimum serves to eliminate outlier nodes that are not relevant to the analysis.

If the node has been sampled enough times, we consider the various ratios of the individual counts to the total count to determine the state. If the ratio of the number of waits to the total count is greater than a cutoff value and the ratio of the number of waits to the number of samples is greater than a total cutoff value, then we assign the node a type of WAIT. If the ratio of the I/O count to the total count is greater than a second cutoff value and the ratio of the number of I/Os to the number of samples is greater than a second total cutoff value we assign the node a type of IO. If the IO cutoff was not met, but the ratio of the sum of the run and I/O counts to the total count is greater than a third cutoff value we assign the node a type of RUN. Finally, if none of these occurs, we assign the node a type of MIXED. The cutoffs here were determined experimentally based on a sampling of different applications. The ratio cutoffs are 0.99 for WAIT, 0.999 for I/O and 0.90 for RUN, while the total cutoffs are 0.01 for WAIT and 0.05 for I/O.

Because we only assign counts to nodes in the trie when they appear at the bottom of a stack trace, it is possible to have nodes that are not sampled sufficiently themselves but have children that are. In practice, this often occurs when the top level routine that gets the next event actually consists of a hierarchy of calls. To handle this, if a node has not been sampled enough times, we consider the total counts of all of its trie children. This has to include at least 40 samples and at least 0.0001 of the total samples to be considered, values again determined experimentally. If these criteria are met, we use the total counts to determine the type of the node using the same ratios and cutoffs as with the case of single nodes. If neither the node nor its children have been sampled enough times, we assign the node a type of ANY.

Once we have assigned a type to each of the trie nodes we are ready to identify event handlers. Based on the types of patterns that event processing can take in an application, we need to look for portions of the trie that have one of the structures shown in Figure 2. The first two structures shown here represent cases where the program is waiting for an event, for example, using an event queue. The first occurs where the function doing the wait calls the event handling routine directly, while the second occurs where there is a routine that returns the next event to process (potentially after waiting for it). In both cases, the single node labeled RUN can actually be a set of nodes labeled either RUN or IO as long as at least one of the nodes is labeled RUN and no node in the set is labeled WAIT.

The second two patterns represent the case where an event is obtained by an I/O operation, typically a read. These roughly correspond to the patterns for wait. In the first of these patterns we require that none of the children are marked as waiting or doing I/O.

In order to identify these patterns, we look at each trie node starting with the top-level children. For each node we count the number of children, and the number of children labeled WAIT, IO, and RUN. We then look at the label of the current node and the counts of its children. Children that are labeled ANY are ignored. If a node does not correspond to one of these patterns, we consider its children.

6. RESULTS

In order to determine whether this simple scheme can actually correctly identify event handlers, we used it as part of the DYMON monitor and ran it on a variety of different applications. The results of this are shown in the outputs in Figure 3.

The first output enumerates the event handlers in a multithreaded system that simulates thousands of objects interacting via gravity. The callback method for the graphical display is identified as are the three event handlers that are used by the task queue. The additional routine, *access$100* is an artifact generated by the Java compiler. As a check for here, the second output is for the same system but this time running without threads and without the task queue. Here our system correctly just identifies the display callback.

The third output is from the kernel of a peer-to-peer system that handles message traffic. Here the system correctly identifies the callbacks, although the first entry, for *TombMessage.toString*, is not logically a callback but occurs because some system routine is causing output to be printed. All the event handlers in the system are correctly identified. The call to *currentTimeMillis* is not logically an event handler, but is a routine called when an event occurs to record the time of the event, so its inclusion is acceptable.

The fourth output is from a server that handles message requires. The *findCategory* routine is not actually a callback. It is chosen because it is called using Java reflection which yields a false indication. The event methods that are identified here are the correct ones, one for the thread that waits for remote connections and one for the handler that waits for messages.

The final output is from a search engine. Here all the event handlers are identified as callbacks as the wait loop was implemented using the *JavaThreadPool* classes.

```
<REACTION>
        <CALLBACK METHOD="edu.brown.cs.cs032.solardraw.SolarDrawImpl@display"/>
        <EVENT METHOD="edu.brown.cs.cs032.solar.SolarSystem@access$100" TYPE="NODE_WAIT"/>
        <EVENT METHOD="edu.brown.cs.cs032.solar.SolarGroupObject@computeLocalGravity" TYPE="NODE_WAIT"/>
        <EVENT METHOD="edu.brown.cs.cs032.solar.SolarSystem$GravityRequest@perform" TYPE="NODE_WAIT"/>
        <EVENT METHOD="edu.brown.cs.cs032.solar.SolarRootObject@rebuildTree" TYPE="NODE_WAIT"/>
</REACTION>


<REACTION>
        <CALLBACK METHOD="edu.brown.cs.cs032.solardraw.SolarDrawImpl@display"/>
</REACTION>


<REACTION>
        <CALLBACK METHOD="edu.brown.cs.cs032.crawler.crawl.CrawlSwingParser$Callback@handleEndTag"/>
        <CALLBACK METHOD="edu.brown.cs.cs032.crawler.crawl.CrawlSwingParser$Callback@handleStartTag"/>
        <CALLBACK METHOD="edu.brown.cs.cs032.crawler.crawl.CrawlSwingParser$Callback@handleText"/>
        <EVENT METHOD="edu.brown.cs.cs032.crawler.crawl.CrawlMain@startThreads" TYPE="NODE_WAIT"/>
        <EVENT METHOD="edu.brown.cs.cs032.crawler.crawl.CrawlMain@loadUrls" TYPE="NODE_WAIT"/>
</REACTION>

 <REACTION>
        <CALLBACK METHOD="edu.brown.cs.taiga.tomb.TombMessage@toString"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.tomb.TombClient$ReplyFlipper@run"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.tomb.TombHost$LogTimer@run"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.tomb.TombClient$ConnectChecker@run"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.comm.CommHost$LogFormatter@format"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.tomb.TombPublisher@run"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.kernel.KernelStore$CacheChecker@run"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.kernel.KernelComm$SocketCopyPinger@run"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.kernel.KernelComm$SocketCopyThread@run"/>
        <EVENT METHOD="edu.brown.cs.taiga.tomb.TombTcpClient$ClientReader@processXmlMessage" TYPE="N
        <EVENT METHOD="edu.brown.cs.taiga.kernel.KernelComm$SocketCopy@handleRead" TYPE="NODE_IO"/>
        <EVENT METHOD="edu.brown.cs.taiga.comm.CommHost@currentTimeMillis" TYPE="NODE_WAIT"/>
        <EVENT METHOD="edu.brown.cs.taiga.tomb.TombTcpClient$ClientWriter@handleWrite" TYPE="NODE_WA
        <EVENT METHOD="edu.brown.cs.taiga.tomb.TombWriter$StringFail@getString" TYPE="NODE_WAIT"/>
        <EVENT METHOD="edu.brown.cs.taiga.tomb.TombWriter$StringFail@getFailHandler" TYPE="NODE_WAIT
</REACTION>

<REACTION>
        <CALLBACK METHOD="edu.brown.cs.webview.classify.ClassifyXml@findCategory"/>
        <CALLBACK METHOD="edu.brown.cs.ivy.mint.client.MintClient@pollNext"/>
        <CALLBACK METHOD="edu.brown.cs.taiga.rind.RindSecurityPolicy@checkPermission"/>
        <EVENT METHOD="edu.brown.cs.taiga.core.CoreRemoteConnection$MessageThread@process" TYPE="NODE_WAIT"/>
        <EVENT METHOD="edu.brown.cs.taiga.core.CoreConnection$ReaderThread@processXmlMessage" TYPE="NODE_IO"/>
</REACTION>


<REACTION>
        <CALLBACK METHOD="edu.brown.cs.s6.language.java.JavaResolver$RefPass@visit"/>
        <CALLBACK METHOD="edu.brown.cs.s6.keysearch.KeySearchKrugle$ScanSolution@run"/>
        <CALLBACK METHOD="edu.brown.cs.s6.keysearch.KeySearchKoders$ResultCallback@handleStartTag"/>
        <CALLBACK METHOD="edu.brown.cs.s6.keysearch.KeySearchBeagle$LoadFile@run"/>
        <CALLBACK METHOD="edu.brown.cs.s6.language.java.FragmentJava$FindVisitor@visit"/>
        <CALLBACK METHOD="edu.brown.cs.s6.keysearch.KeySearchBeagle$RunBeagle@run"/>
        <CALLBACK METHOD="edu.brown.cs.s6.language.java.JavaResolver$DefPass@visit"/>
        <CALLBACK METHOD="edu.brown.cs.s6.keysearch.KeySearchKoders$LoadSolution@run"/>
        <CALLBACK METHOD="edu.brown.cs.s6.language.java.JavaTyper$TypeFinder@visit"/>
        <CALLBACK METHOD="edu.brown.cs.s6.keysearch.KeySearchKoders$ScanSolution@run"/>
        <CALLBACK METHOD="edu.brown.cs.s6.language.java.JavaTyper$TypeSetter@endVisit"/>
        <CALLBACK METHOD="edu.brown.cs.s6.engine.EngineTester@testMethod"/>
        <CALLBACK METHOD="edu.brown.cs.s6.language.java.JavaResolver$RefPass@endVisit"/>
        <CALLBACK METHOD="edu.brown.cs.s6.language.java.JavaTyper$TypeSetter@visit"/>
        <CALLBACK METHOD="edu.brown.cs.s6.keysearch.KeySearchKrugle$LoadSolution@run"/>
</REACTION>
```

Figure 3. Result of event handler analysis.

From these examples, we see that the system is quite successful in correctly identifying the event handlers in an active application without knowledge of the application. It identifies a few extra routine that the programmer would not at first characterize as event handlers, but some of these could be easily eliminated (e.g. ignore internal Java calls and calls from java.lang.reflect).

The analysis is also quite stable. Although the system continually recomputes the set of event handlers, this set tends to stabilize within thirty seconds and then remain the same for the rest of the run.

7. FUTURE WORK

In this paper we have shown that it is possible to use relatively simple dynamic analysis techniques to quickly and accurately find the event handlers in a running application. While our approach isn't perfect, it is quite acceptable for its intended purpose of performance analysis and can be easily improved, for example by detecting uses of reflection and Java compiler artifacts.

Finding event handlers, however, represents only the first step towards understanding program performance and behavior in terms of events. There are several more steps that need to be accomplished before we can provide the programmer with accurate and complete information.

The next step is to use the event handler information as a basis for organizing and presenting performance information to the programmer. This can be done by observing which event handler the system is currently processing in a stack trace and allocating run time to that event handler. To provide more detailed information, we would have a DYMON agent that instrumented the event handling routines to determine the number of times they were called and the time (or other resources) spent per call.

The subsequent step involves detecting and following events throughout a programs execution. While many transactions may be handled completely with one event handler, other transactions require multiple event callbacks, with processing partial processing of the event being done on each callback. Here one event triggers other events, for example deferred processing. Similarly, in distributed systems, events in one process might trigger events in another process. In both these cases, a complete understanding of event processing requires tracking the causality of events. We are planning to tackle this problem with additional dynamic analysis possibly combined with some static analysis of the program and user input on how to uniquely identify messages.

We also plan to use this framework to help automatically define interesting events for program visualization, replacing the programmer's event definitions in our VELD framework [13] with ones that are determine heuristically.

The code described here is available as part of the DYMON package which can be found in the WADI system at ftp://www.cs.brown.edu/u/spr.

8. ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation through grant CCR0613162.

9. REFERENCES

1. Ziya Aral and Ilya Gertner, “Non-intrusive and interactive profiling in Parasight,” *Proc. ACM/SIGPLAN Conf. on Parallel Programming*, pp. 21-30 (January 1998).

2. Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory Lee, Barton P. Miller, and Martin Schulz, “Stack trace analysis for large scale debugging,” *Proc. IPDPS 2007*, pp. 1-10 (March 2007).

3. Bryan M. Cantrill and Thomas W. Doeppner, Jr., “Threadmon: a tool for monitoring multithreaded program performance,” *Proc. 30th Hawaii Intl. Conf. on Systems Sciences*, pp. 253-265 (January 1997).

4. Rodrigo Fonseca, Goerge Porter, Randy H. Katz, Scott Shenker, and Ion Stoica, “X-Trace: a pervasive network tracing framework,” *Proc. 4th USENIX Symp. on Networked Systems Design and Implementation*, pp. 271-284 (2007).

5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley (1995).

6. Michael M. Gorlick, “The flight recorder: an architecture for system monitoring,” *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 175-183 (December 1991).

7. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, “gprof: A call graph execution profiler,” *SIGPLAN Notices* Vol. **17**(6) pp. 120-126 (June 1982).

8. Minwean Ji, Edward W. Felten, and Kai Li, “Performance measurements for multithreaded programs,” *Proc. ACM SIGMETRICS/Performance '98*, pp. 161-170 (August 1998).

9. Wim De Pauw and Gary Sevitsky, “Visualizing reference patterns for solving memory leaks in Java,” in *Proceedings of the ECOOP '99 European Conference on Object-oriented Programming*, (1999).

10. Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan, “Drive-by analysis of running programs,” *Proc. ICSE Workshop of Software Visualization*, (May 2001).

11. Steven P. Reiss, “Connecting tools using message passing in the FIELD environment,” *IEEE Software* Vol. **7**(4) pp. 57-67 (July 1990).

12. Steven P. Reiss, “Event-based performance analysis,” *Proc 11th IEEE Intl Workshop on Program Comprehension*, pp. 74-81 (2003).

13. Steven P. Reiss, “Visualizing program execution using user abstractions,” *SOFTVIS 06*, pp. 125-134 (September 2006).

14. Steven P. Reiss, “Controlled dynamic performance analysis,” *Proc. 2nd Intl. Workshop on Software and Performance*, (June 2008).

15. Francois Taiani, “CosmOpen: a reverse-engieering tools for complex open- source architectures,” *Proc. Intl. Conf. on Dependable Systems and Networks*, pp. A49-A51 (June 2003).