CLUE: System Trace Analytics for Cloud Service Performance Diagnosis

Hui Zhang*, Junghwan Rhee*, Nipun Arora*, Sahan Gamage†, Guofei Jiang*, Kenji Yoshihira* and Dongyan Xu†
*Department of Autonomic Management
NEC Laboratories America
Princeton, New Jersey, 08540 USA
Email: {huizhang,rhee,nipun,gfj,kenji}@nec-labs.com
†Department of Computer Science
Purdue University
West Lafayette, Indiana, 47907 USA
Email: {sgamage,dxu}@cs.purdue.edu

***Abstract*—In this paper, we present *CLUE*, a system event analytics tool for black-box performance diagnosis in production Cloud Computing systems. *CLUE* provides an unified and extensible means of profiling service transactional behaviors, and builds structured data called event sketches. *CLUE* further offers a set of analytic tools for summarizing and analyzing event sketches by integrating data mining and statistical analysis.**

***CLUE* has been developed in NEC as an internal tool and applied in diagnosing a diverse set of real performance problems for multi-tiered IT applications running on multi-core servers of major platforms including Linux (Redhat, Fedora), Unix (HP-UX), and Windows (Windows Server 2008). We demonstrated the evaluation of our framework on real-world IT systems, and showed how it can enable visibility and effective diagnosis of service system performance problems.**

***Keywords*—performance diagnostics; system troubleshooting; data centers; Cloud Computing; data analytics**

I. Introduction

Performance diagnosis in production Cloud Computing systems desires an efficient, unsupervised tool for locating fine-grained performance anomalies. As light-weighted and flexible kernel tracing tools(e.g., SystemTap [1], DTrace [2], and Event Tracing for Windows [3]) get mature, performance debugging with low level system events (e.g., system calls) has been studied in many approaches [1], [2], [4], [5], [6], [7], [8]. These are called black-box approaches which do not rely on application software's internal knowledge.

A. Related Work

Many of these black-box approaches take a process-centric view and offer interactive query tools to explore the statistical properties of the recorded kernel events and system resource usage information for system-related performance diagnosis. For example, LTTng trace analyzer [9] generates per-process resource utilization (including CPU, disk, file and network usage) from kernel events which are useful for system administration and software development. Another tool, the LTTV delay analyzer [10] builds multiple state machines for each process that describe its running status (e.g., working, blocked, interrupted) and the related interfering processes to let users quickly understand the dependencies among processes and see how total elapsed time is consumed by its main components.

While process-level performance debugging provides meaningful local views, bottlenecks in mission critical segments of large systems can affect several services in a domino effect. Understanding the dependencies and causalities of multiple local performance problems often prove to be a headache for the administrator.

Several black-box approaches target capturing the path and the timing of an individual service request across the components of a multi-tiered system. Imprecise black-box approaches such as Project5 [11] and WAP5 [12] accept imprecision of probabilistic correlations. Project5 proposes a nesting algorithm which assumes RPC-style (call-returns) communication, and a convolution algorithm which does not assure a particular messaging protocol cannot build individual causal paths for each request. WAP5 infers causal paths for wide-area systems in a per-process granularity using library interposition.

Precise black-box approaches such as SRAMD [13], BorderPatrol [14], vPath [7], and PreciseTracer [15] build request traces for specific protocols or application threading models. SRAMD traces service component's packet-level traffic unobtrusively and can identify RPC-based inter-service communications. BorderPatrol isolates and schedules events or requests at the protocol level to precisely track service requests with the explicit knowledge of diverse protocols used by multi-tier services. When multi-tier services are developed from commercial components or heterogeneous middleware, BorderPatrol needs to write many protocol processors and requires more specialized knowledge than pure black-box approaches. vPath continuously logs in virtual machine monitor (VMM) which thread performs a send or recv system call over which TCP connection, and makes assumptions about the threading model of distributed system in the log processing phase, such as synchronous communication among components of the system and a single thread handling all the messages common to one request. PreciseTracer [15] offers an online request tracing system by reconstructing network communication events into causal paths, classifying these causal paths into different

978-1-4799-0913-1/14/$31.00 ©2014 IEEE

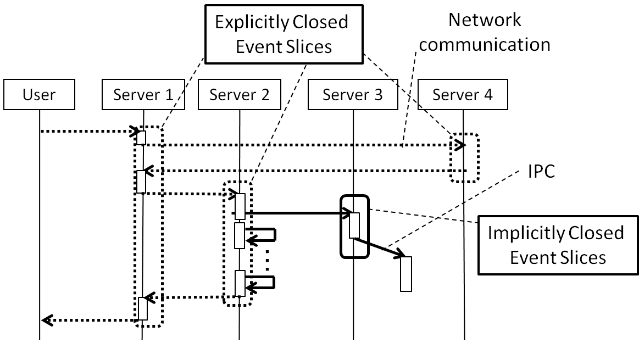
Explicitly Closed Event Slices
Network communication
User
Server 1
Server 2
Server 3
Server 4
IPC
Implicitly Closed Event Slices

Fig. 1. Explicit closed event slices and implicitly closed event slices used to understand program behavior.

patterns according to their shapes, and presenting an macro-level abstraction, dominated causal path patterns, to represent repeatedly executed causal paths that account for significant fractions. These black-box approaches take only a small subset of system events, network events, into their trace analysis, and rely on explicitly request-reply communication patterns.

More intrusive approaches [16], [17], [18] built on low-overhead end-to-end tracing can capture the path and the timing of an individual request across the components of a distributed system. For example, Pip [17] logs actual system behaviors with the insertion of annotations into source code. Magpie [16] uses an event schema to correlate events collected at different points in a system into causal paths; it must obtain the source code of applications to build the schema in order to track a request from end to end. In production environments, intrusive instrumentation is usually a concern due to potential impact on the operation; also, service software may be composed of multiple components from different vendors, and domain knowledge on the structure or the operation of the software may not be available. For these reasons, CLUE takes a black-box approach tracing at the kernel space that requires no knowledge on the application space, while aims at inferring application behavior in an offline stage through trace analysis.

B. Our Contribution

In this paper, we present a trace analytic tool, called *CLUE*, for service performance management. It offers a data-centric approach for analyzing service performance issues manifested through its interaction with the underlying Cloud systems.

CLUE provides an extended application modeling mechanism using multiple types of kernel event sequences to address a wide scope of application performance problems. Figure 1 illustrates an example of real world applications that we diagnosed. It has multiple tiers shown as server 1 $\sim$ 4 which interact through network communication (thick dotted arrows) and inter-process communication (IPC, thick solid arrows). Previous work [7], [14], [15] modeled application behavior using network communication which is relatively more reliable than other kernel events to be inferred due to strong patterns enforced by network events (e.g., TCP/IP events). This type of kernel events are illustrated as *explicitly closed event slices* in the Figure 1.

While this set of events are effective in many cases, we believe that causal path inference on the basis of explicit network events is not enough in complex cases. For instance, the same figure also shows execution patterns surrounded by IPC events shown as *implicitly closed event slices*. To recognize and analyze this type of kernel events, causal paths based on network events may not be the best solution. As another example, worker threads can potentially interfere with each other due to a logic failure of synchronization code. This portion of application code (and corresponding kernel event trace) has nothing to do with network events. This drives us to explore application performance diagnosis using an expanded set of OS kernel events beyond a set of network-oriented event sequences.

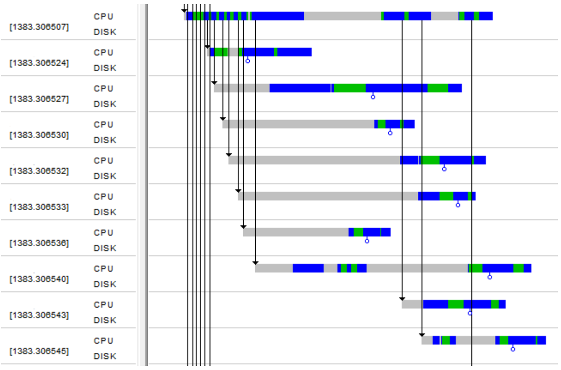

Fig. 2. A scheduling problem in multi-threaded software which has unusual idle time in children threads. (Colors: green=user code execution, blue= kernel code execution, gray= idle status)

CLUE, motivated by DNA sequencing research [19], [20], develops new algorithms to construct structured traces called event sketches by linking event slices through causality relationships. The main challenge of characterizing those fine-grained application behaviors is the lack of application semantics due to low level input, OS kernel events. To remedy this problem, CLUE further provides new kernel event analytic techniques that brings application context information into understanding performance metrics commonly derived from kernel events. CLUE builds feature vectors for event sketches based on their included kernel events, and applies data mining techniques such as clustering for data summarization. CLUE also offers advanced performance statistical analysis called conditional data mining by using the slice causality information and statistical data analysis techniques. Those event-based techniques are complementary to existing time series based techniques [21] for service performance modeling and anomaly analysis.

Figure 2 shows a real scheduling problem in multi-threaded software that CLUE can diagnose through implicitly closed event slices defined on thread synchronization events (please refer to Section VI-B for details). CLUE has been developed in NEC as an internal tool and applied in diagnosing a diverse set of real performance problems for multi-tiered IT applications running on major platforms including Linux (Redhat, Fedora), Unix (HP-UX), and Windows (Windows Server 2008). The CLUE framework offers new analytic workflows to assist users gaining deeper understanding of system traces collected from the Cloud systems.

The rest of the paper is organized as follows. Section II presents the background information and Section III describes

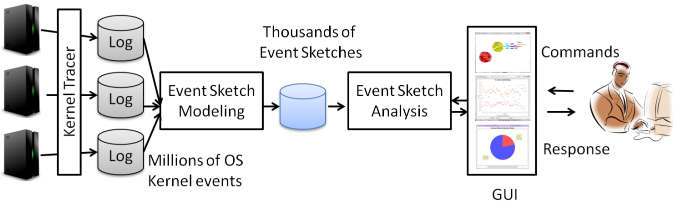

Fig. 3. Overview of CLUE architecture.

the architecture. The main components, event sketch modeling and event sketch analysis, are respectively presented in Section IV and Section V. Section VI presents the evaluation and case studies. Section VII concludes this paper.

II. Background

OS Kernel Events. As a basic core component of a computer system, an OS kernel provides the lowest-level resource abstraction layer for application software. Typical examples of kernel events include system calls from processes, scheduling events, interrupts, I/O operations, and locking operations.

Kernel Tracing. Here we discuss several utilities for tracing OS kernel events in major operating systems including Linux, Unix, and Windows. SystemTap [1] (for Linux) and DTrace [2] (for Solaris) provide scripting languages resembling C that enable probe points in the kernel and implement their associated handlers. Event Tracing for Windows (ETW) [3] is a general-purpose event-tracing platform on Windows operating systems. Using an efficient buffering and logging mechanism implemented in the kernel, ETW provides a mechanism to log events raised by both user-mode applications and kernel-mode device drivers.

III. CLUE Architecture

In this section, we present the CLUE architecture. Figure 3 illustrates the four main components:

1) **Kernel Tracer**. This component collects kernel event traces of all active processes in the infrastructure and then pushes to a central log store. A proprietary kernel tracer from NEC, mevalet [22], is integrated into the framework for this component. We note this is a generic component, and other kernel tracing tools such as SystemTap and DTrace can be interchangeably used.
2) **Event Sketch Modeling**. This component aggregates raw event traces, and applies several algorithms to extract a group of kernel events having causality relationship that we call sketches.
3) **Event Sketch Analysis**. Given the kernel event sketches, this component offers performance analysis.
4) **GUI**. This is the interface that a user interacts with CLUE by sending queries and using our interactive workflows. The GUI is a front-end which in tightly coupled with our event-sketch analysis engine. It provides rich information and visualization for understanding the analysis results.

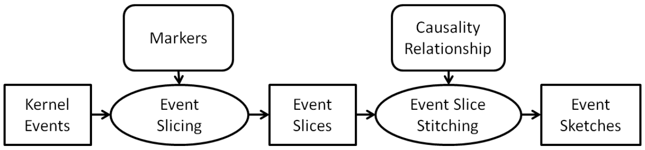

Fig. 4. Overview of event sketch modeling.

IV. Event Sketch Modeling

The first step for performance diagnosis in CLUE is to model and profile application behaviors. Figure 4 illustrates the overview of this procedure which has two steps. Firstly, a set of events within a thread is extracted based on their event types called boundary *markers*. Using the set of markers, kernel events are sliced out forming an independent subsequence of kernel events for each thread. We call this chunk of kernel events within a thread an *event slice (ES)*. An event slice can be interpreted as an application activity upon executing an individual task. Secondly, these slices are aggregated depending on causal dependency among them to compose a bigger group of kernel events called *an event sketch*. Event sketches consist of event slices from multiple tiers correlated using specific matching rules called *causality relationships*. An event sketch can be interpreted as aggregate system activities upon coordinating processes executing a single task.

A. Pre-Processing

A kernel trace can contain a variety of kernel events including system call traces, scheduler events, network events, context switches and hardware/software interrupts. These events can be easily captured by standard low overhead kernel tracing tools using system call tracing and hooks in the kernel code.

As a first step, we pre-process kernel event traces with thread/process information so that per-thread kernel traces are generated. Each kernel event in CLUE has 5 tuples: (1) thread information, (2) time stamp, (3) CPU ID, (4) kernel event type, and (5) optional kernel function call parameters (e.g., transfer size for network events, file descriptors).

```
m1.memcached.30746, 39.88, 0, TCP_RECV, ...
```

For instance, the above example is a TCP receive event generated by a memcached process (PID: 30746) at the server m1; this event occurred at time 39.88 seconds in the CPU 0. The function call parameters for this kernel event is omitted here for brevity.

B. Event Slicing

The previous step converts a raw kernel event stream into a set of event streams for individual threads. In this step, a block of thread events are sliced into separate identifiable event subsequences (i.e., event slices).

An event slice begins and ends based on different communication patterns. Correspondingly we define two types of event slices: explicitly-closed event (EC) slices, and implicitly closed event (IC) slices.

EC Markers	IC Markers
TCP (Accept, Close)	Msg. queue (Send,Recv)
TCP (Connect, Shutdown)	PIPE (Write,Read)
TCP (Send, Recv)	Semaphore (Permit)
UDP (Send, Recv)	Signal (Send, Handle)
Unix stream (Send, Recv)	Futex (Wake,Wait)
	Process (Clone,Create)
	Process (kwakeup,ksleep)

TABLE I. MARKING EVENTS USED IN CLUE.

```
<An event slice at the Memslap client>
m1.memslap.17668,39.882272339,0,TCPSEND_LX
m1.memslap.17668,39.882281155,0,SVC_BGN_SOCKETCALL
m1.memslap.17668,39.882282102,0,SVC_END_LX
m1.memslap.17668,39.882283046,0,SVC_BGN_POLL
m1.memslap.17668,39.882283892,0,PSAVE_LX
m1.memslap.17668,39.882763903,0,PRESUME_LX
m1.memslap.17668,39.882764546,0,SVC_END_LX
m1.memslap.17668,39.882765456,0,SVC_BGN_SOCKETCALL
m1.memslap.17668,39.882767659,0,TCPRECV_LX

<An event slice at the Memcached server>
m1.memcached.30746,39.882586495,0,TCPRECV_LX
m1.memcached.30746,39.882590352,0,SVC_BGN_SOCKETCALL
m1.memcached.30746,39.882591130,0,TCPSEND_LX
```

Fig. 5. Event slices in the memslap client process and the memcached server process. (kernel event parameters are omitted.)

a) Explicitly Closed Event Slice.: An explicitly closed event slice is formed on the basis of communication patterns which follow an explicit and strict communication pattern. Explicitly closed event slices are well studied and have been covered in previous approaches [7], [23]. For example, a web server replying to a web request is typically composed of a series of kernel events for a complete TCP connection, starting from a TCP_ACCEPT kernel event and ending with a TCP_CLOSE.

Figure 5 presents a pair of event slices for a client program called Memslap and a server program, Memcached [24], which is an in-memory key-value store for small chunks of arbitrary data (strings, objects). In particular, this pair of event slices is captured when the client issued one *get* operation to the server. We can observe this transaction between a client and a server initiates with a TCP_SEND event in the client. This request is received at the server (shown as a TCP_RECV event). Memcached sends the result to a client and finally the transaction ends with the TCP_RECV event in the client. This example shows how CLUE uses communication events to capture explicitly closed event slices.

b) Implicitly Closed Event Slices.: As explained earlier, explicit communication patterns are easier to catch because network-driven events have strong pattern defined by a pair of markers. However, many service applications also use more complex communication patterns such as IPCs. For example, a pipe communication is realized by processes writing and reading from the same pipe identified by a file pointer. Unlike TCP send/recv, the start and the end markers of such communication may not be explicitly paired. We call such communication patterns as implicit communication patterns, and our corresponding event slices as implicitly closed event slices. They are implicitly closed because there is no explicit information to identify the related events invoked by a task except the single IPC event.

```
/* producer thread */
ap_queue_push(fd_queue_t *queue,
 apr_socket_t *sd, apr_pool_t *p) {
  pthread_mutex_lock(&queue->one_big_mutex);
  elem = &queue->data[queue->nelts];
  elem->sd = sd;
  elem->p = p;
  queue->nelts++;
  pthread_mutex_unlock(&queue->one_big_mutex);
}

/* consumer thread */
ap_queue_pop(fd_queue_t *queue
 apr_socket_t **sd, apr_pool_t **p) {
  pthread_mutex_lock(&queue->one_big_mutex);
  elem = &queue->data[--queue->nelts];
  *sd = elem->sd;
  *p = elem->p;
  pthread_mutex_unlock(&queue->one_big_mutex);
  /* caller uses values in sd & p after return */
}
```

Fig. 6. Synchronization code of Apache web server causes futex kernel events.

Since there is no direct correlation to decide the start and end of an implicitly closed event slice, we employ several heuristics. One such heuristic relies on context switch events which, when coupled with nearest communication events, can assist in event slicing.

Note that the kernel events in an implicitly closed slice may not have strong causality relationship. Rather they are used as signatures to model the application execution context; they represent the characteristics of the application behavior (indirectly shown in the low level kernel events) related to this communication and help on advanced trace analysis such as similarity searching or event summarization.

Figure 6 shows an example of an application code that implicitly closed event slicing can capture. This is a critical section of inter-thread message passing using shared memory in Apache web server. The producer thread ap_queue_push stores a message in the shared queue (sd and p) and the consumer thread pulls the data in the ap_queue_pop function. Two threads are synchronized by using pthread_mutex_lock and pthread_mutex_unlock calls. In the kernel, this pair of code generate a pair of futex wait and wake events which are used to identify beginning and ending of the slice. Therefore, performance problems related to the synchronization code based on futex can be traced and analyzed by CLUE.

C. Event Slice Stitching

Event sketches are formed by connecting event slices, based on the causality among their marker events One event sketch (S) is defined through a two-item data structure $< E, D >$, where E is a set of event slices (ESs), and D is a set of causality relationships between the event slices.

The stitching algorithm has mainly two steps: (1) for each event slice, the algorithm searches for marker events in other event slices to find causal relationships; (2) after the search, it constructs a full-path sketch of the communication in the multi-tiered system over multiple-hops of causality.

Stitching together explicitly closed event slices: as explained earlier, based on the communication pattern explicitly closed event slices are easier to stitch together. All explicit event slices have a single pair of marker events. These event slices are causally connected and are triggered by an earlier event slice which sends a message or initiates a communication.

Stitching together implicitly closed event slices: implicit communication patterns such as pipe, message queue are harder to decipher by only observing their corresponding kernel events. In order to connect multiple slices and compose event sketches, we use the properties of kernel events such as the event type, event parameters, and the timestamps. We found related kernel events can be reliably associated using their event parameters (e.g., file descriptors) associated with the events. Timestamps are used to apply the time causality relationship among slices.

Specific rules on what conditions are used to connect event slices are empirically captured by examining APIs causing kernel events. Typically more than one rule are applied in combination. For example, pipe kernel events have a pointer address for the file descriptor and this information is used to associate read from and writes to the same pipe. Then we apply time causality rules based on the observation that the consumers should follow producers. This is reflected by reads following writes.

Time causality is an obvious relationship that can be found within an ES and across ESes. Within an event slice, the causal order among kernel events are already made because the kernel events are serialized in time order. Time causality among event slices have been used in existing black-box transaction profiling approaches [7], [23]. They used network events by matching IP addresses and time proximity to capture kernel events along with TCP/IP network flows. Our approach also adopts such mechanism to cover communication-driven application transaction logics.

Another example is the analysis of synchronization code of pthread locks/unlocks. A program may have multiple synchronization code among threads in its code base. To analyze the synchronization behavior reliably, it is important to tell apart each code set surrounding a different lock and perform local analysis per a lock. Such code set on distinct locks can be differentiated because the pointer addresses of underlying lock data structure could be recorded as event parameters. In Figure 6, this specific synchronization behavior of threads can be uniquely and deterministically associated due to the lock ID captured as a parameter of *futex wake* and *wait* events. Similarly the parameter information for other events are used for identifying deterministic association between event slices.

V. Event Sketch Analysis

After extracting event slices and stitching them together to form event sketches from raw kernel events, we build feature vectors for event sketches based on their included kernel events, and apply existing data mining techniques such as clustering [25] for data summarzation. Besides, we develop new techniques to improve black-box analysis of OS kernel events by using the slice causality information and statistical data analysis techniques with event sketches.

A. True Latency Analysis

One of important metrics to represent system performance is request latency of services. However, determining the latency of event slices is not straightforward because of the relationship among event slices. For instance, in a multi-tier system of a webserver and an application server, an application behavior of one tier (e.g., webserver) may depend on the behavior of another tier (e.g., application server). After the web server sends a request to the application server, it will wait till the application server returns data. Then all latency of the application server is reflected to the latency of the webserver.

This challenge requires understanding on how distributed system executes in multi-tier systems, which is non-trivial to measure. CLUE generates event sketches by using causal relationship among event slices. Using this information, we can capture this behavior and derive true latency accordingly by *recursively subtracting latency of dependent components*. We call this process *true latency analysis*. In the evaluation section, we present how this true latency of an individual tier can assist the root cause localization in performance diagnosis.

B. Conditional Data Mining using Sketch Structures

Inferred application context in the form of slice casuality structures is important in our black-box analysis for the following reasons. First, it can narrow down the focus on comparable event sketches due to inferred application code behavior based on kernel event sequences. Also we can use certain performance information such as a range of latency values to focus on more problematic symptoms (e.g., event sketches with high latency).

Such conditions can be considered as filters of successive steps of analysis which prune out unrelevant or less interesting events. This process refines the scope of analysis, thus improves the focus on the data relevant to problems.

To model this iterative process, we adopted conditional probability that is a well-known statistical data analysis technique on multiple features. For example, a user may query what is the probability that the transaction latency is larger than a threshold x for service requests of a kind. Here the first condition is a performance metric on how long a trace takes. The second condition "a kind" refers to an application context (e.g., a function in the application program). These two successive queries can be formulated as a conditional probability of two features $P(C_2|C_1)$, where C_1 and C_2 represent the conditions on application context or performance.

We believe determining the types of possible conditions and the sketch of application could be potential research topics that require further efforts and evaluation. As our initial attempt as an example, we list a set of application conditions that we use in Table II. More details on the condition types and their usage are described as follows.

- Each event sketch has corresponding markers which define its start and end. The types of markers decide what types of application behavior we would like to analyze. For the problems regarding application transaction logic, network oriented markers such as `TCP_ACCEPT` or `TCP_RECEIVE` would provide relevant kernel behavior. Futex events show

Condition Type	Examples
Marker type of an event sketch	Marker type == `TCP_ACCEPT`, Marker type == `Futex Wake`
Process name, process ID number	Process name == httpd, `PID` == 3213
Similarity of system behavior features	Cluster name == c_1
Latency (w/ or w/o true latency analysis)	Latency of an event sketch >= 0.02
Idle time (mean, variance)	Mean idle time of an event sketch >= 0.5

TABLE II. TYPES AND EXAMPLES OF APPLICATION (CONTEXT, PERFORMANCE) CONDITIONS.

kernel behavior regarding thread synchronization and scheduling; thus effective for troubleshooting scheduling problems.

- The kernel tracer that we use provides process names and PID numbers. Hence event sketches can be tagged with process name and PID numbers. For event sketches including event slices for multiple tiers, this could represent information of an individual tier. This information allows users to select a certain program they would like to examine in the analysis.
- While kernel events in the thread level can be distinguished using process names and PIDs, a finer grained distinction within a thread is not available just using information from the trace since our input, kernel events, does not have specific information on program execution. This can be done using the inferred program characteristics, system behavior features, presented in the previous section. After clustering, each cluster represents a group of similar user code patterns.
- Latency is a commonly used performance metric to determine the performance of an application. We can use latency defined for either an event sketch or an event slice that is computed as the difference between the timestamps of the first and last kernel events. If the event sketch is in the form of network oriented service requests, taking out the dependent components' latency gives true latency taken only in each tier; thereby, allowing tier level anomaly detection based on latency.
- Idle time is another metric useful to represent performance problems. If this metric is combined with other conditions, we can describe some types of kernel behavior. For instance, if a scheduling/synchronization code is poorly designed or configured, it could cause worker threads interfere each other. Such interference of multiple threads can cause temporary idle time with a varying degree. It can be modeled by the combination of high mean and variance of idle time.

We use at least three conditions in combination as the default analysis of applications. First, CLUE will generate various event sketch based on multiple markers. Each set produced by a marker represents application behavior of a specific nature. Given the overall summary of all sets, users decide which behavior he is interested (e.g., TCP behavior or scheduling behavior).

Second, the process name differentiates the events of different programs and instances. GUI visualizes data points clustered by the process names along with the latency of each point. Users can determine which set of application data he is interested based on the program name and their latency.

The third condition gives more specific semantic information within a process on the program code. Event sketch are clustered using the inferred application context (i.e., system call frequency vector) and visualized as groups.

On top of the default analysis, users will be guided to select additional conditions depending on usage scenarios. More informations specific to each analysis will be presented in the case study at Section VI-B. In general, we use conditional probability along with conditions on application context and performance metrics to achieve improved accuracy in performance diagnosis.

VI. Evaluation

CLUE has been tested intensively in NEC for multiple system integration and operation projects. The main usages of CLUE covered (1) business function profiling; (2) service performance debugging; (3) system troubleshooting. Besides, we also created a J2EE PetStore 2.0 [26] service testbed, and reproduced seven real performance anomaly or misconfiguration bugs described in [27].

A. Performance Overhead

We start with the overhead evaluation of CLUE as a runtime management tool. The CLUE usage includes tracing and analysis of kernel events. We describe the overhead of each component as follows.

1) Overhead of Tracing: As we discussed in the implementation section, kernel tracing is a general component and any kernel tracer such as DTrace, SystemTap, and ETW can be used. Some of these tools are designed and used for production systems; hence, we think the overhead of kernel tracers are reasonably accepted. Since we did not design this component, we present a simple evaluation on our tracer as an example on the cost for kernel tracing.

We measured the overhead of mevalet for three-tier PetStore system composed of an Apache webserver, a JBoss application server, and a mysql database server. Three machines commonly have Intel Core 2 CPUs with 2 GB memory. As workload, we used `httperf` utility to replay 14 URLs accessing different pages in the PetStore application. This replay of a log is performed at the rate of 70 requests per second for 200 seconds period. We recorded resource utilization using `sar` utility in two scenarios: (a) when the mevalet system is not tracing system activity, and (b) when the mevalet system is recording system activity. In this test, mevalet adds only about 2% overhead to the overall CPU utilization.

2) Overhead of Analysis: In our framework, the only runtime component is the kernel tracer, and the entire analysis of traces is performed offline. Therefore, CPU and memory overhead of our analysis is not a critical concern for the deployment of systems.

For trace analysis, we used a desktop machine which has a Intel Core 5 CPU, 2GB memory, Windows 7 for its operating system. This is a moderate specification for a desktop machine at this moment. The cases to be presented in the following sections are analyzed in this machine, and all of such cases including event sketch modeling and event sketch analysis took no more than five minutes. We think this speed and resource demand is in an acceptable range as offline analysis in a regular desktop or a laptop.

B. Case Study: Diagnosis of Inefficient Synchronization Behavior

As the case study of CLUE [1], we present an examination of a performance problem of a real world software that we call Internet Gateway Transaction Application. This software is being installed and tested by one of our customers; its specific product identity is not presented for confidentiality. This case is regarding a reported performance problem: having an unexpected low transaction throughput in the deployment on a HP-UX high-end server with 16 cores.

We obtained a set of mevalet kernel event traces recorded from this system and analyze them using CLUE to understand this performance problem. Note in this real scenario, we do not have direct access to the real machines or software since the company considers some of such information confidential. Instead, we get traces indirectly recorded in the premise through the owners.

Due to a pure black-box design which does not require any configuration or knowledge in the deployment, we can analyze customers data and provide reports as a third party consultant. Many existing whitebox or semi-black-box approaches that require any domain knowledge or instrumentation on the software [29], [30] cannot be applied in this type of scenario.

a) Trace Analysis.: As customers identified this problem is reproducible, they activated the kernel tracer in their system around the symptom leading to a relatively short kernel trace of total 89568 kernel events. From this set of events, event sketch modeling process extracted total 82 event sketches. Among them, 78 sketches (over 95%) are constructed using implicit event slices, particularly `kwakeup` and `ksleep` system calls, which are related to synchronization of threads.

b) Diagnosis with Application Context.: Once event sketches are extracted, we applied default application conditions and refined the scope of analysis via clustering. First, CLUE reports two different types of event sketches: `TCP_SEND` and `kwakeup`. The majority of these sketches were produced by the `kwakeup` category, thus, we selected it. The second clustering based on process names gave two clusters as shown in Figure 7. The first cluster has 70 event sketches of the service program that is reported to have performance concern. The second cluster has 8 event sequences. Since we know the program of interest, we chose the first cluster.

After that, clustering is applied based on system behavior features for inferring application behavior within threads, and the traces with a similar system call distribution are aggregated into a cluster. CLUE produced total 7 clusters illustrated in Figure 8. The transactions shown as + and ∗ are spread over the tracing period of 0 ∼ 6 seconds. Except these data most other samples of Cluster 1 and 3 are in the range of 2.701 ∼ 2.706 seconds, as shown in Figure 9. This highlights severe synchronized behaviors among service working threads.

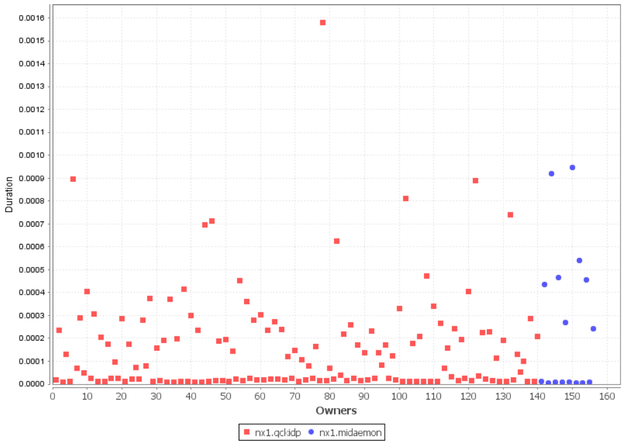

Fig. 7. Clusters based on program names. Distinct colors represent different clusters. X axis shows program names (sorted by name). Y axis shows latency.

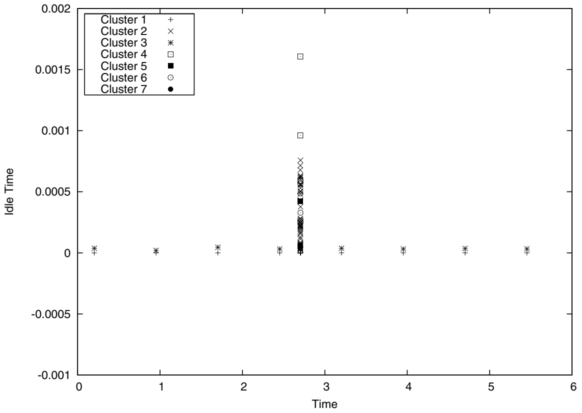

Fig. 8. Clusters based on System Behavior Features.

This set of event sketches are generated by `kwakeup` events which are highly related to scheduling behavior. Hence, we analyzed idle time to test any contention among threads. Specifically we applied two conditions regarding idle time: ranking clusters regarding mean and variance of idle time. If threads interfere each other due to a synchronization problem, event sequences corresponding to such symptoms can have high idle time with a varying degree due to interference each other. Such behavior can be modeled as relatively high idle time values with having high mean and high variance.

After ranking clusters with these idle time metrics, the clusters with low mean/variance are filtered out, thereby showing the candidates at the top. The identified behavior is illustrated in Figure 10. The arrows visualize fork activity and each row represents the execution of a thread. Small boxes labeled as 1, 2, and 3 show different kinds of code that is executed in stages. Multiple boxes share the same numbers being clustered into the same clusters. CLUE correlates these pieces of code execution that are scattered around in time, but showing similar application context inferred by system behavior feature and

[1] For more CLUE case studies, please refer to the paper [28].

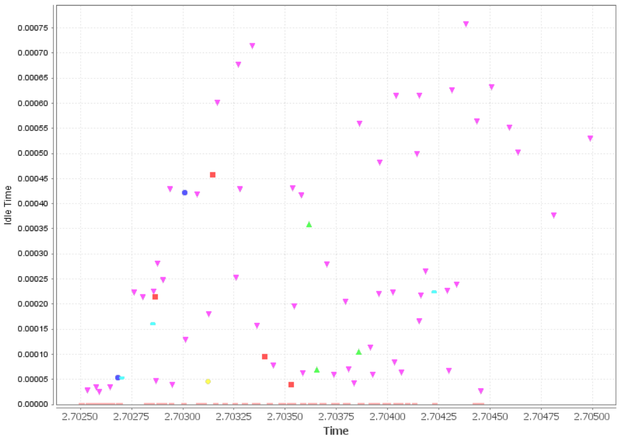

Fig. 9. Clusters based on system behavior features(zoom-in). Distinct colors represent different clusters. X axis shows time. Y axis shows idle time.

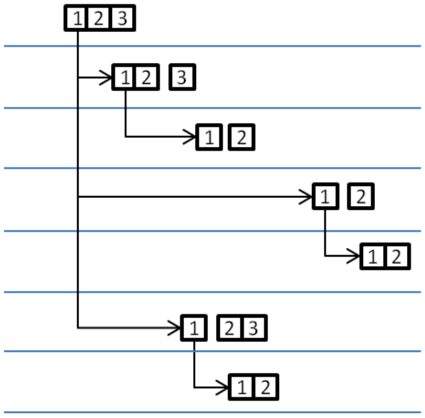

Fig. 10. Illustrating black-box identification of hierarchy of event slices. Event slices in the same hierarchy are captured in the same cluster.

interfering behavior each other based on idle time properties. This process, essentially, infers the event sequences of kernel events having scheduling interference automatically from tens of thousands of kernel events. Similar to finding needles in a haystack, this is a highly challenging task in general due to complexity and lacking semantic information for a pure black-box approach, but it was possible for CLUE because of our techniques to infer application context and iteratively and systematically narrow down the focus of analysis.

c) Validation.: As validation, we manually examined the trace samples of the top clusters to figure out where they are located in the execution of application with a visualization tool of our tracer – it visualizes the entire set of recorded events – shown in Figure 2. Interestingly we found visually that the event slices belonging to the same cluster are located in the similar hierarchy of process relationship.

Similar to the diagram in Figure 10 each horizontal bar represents the execution of one process whose process identification and thread identification are shown in the left side of the figure (e.g., [1383.306507] where 1383 is the process ID and 306507 is the thread ID). Green bars and blue bars show the execution time spent respectively in the user and kernel space. The gray bars represent idle time.

Here the process (1383.306507) forks 7 children processes and later more processes. The problematic symptom is that these forked processes are staying in the idle state (gray bars). These threads only begin work (shown as the following blue bar) after taking idle time in a varying degree.

This behavior is automatically captured by CLUE simply using clustering kernel events in a black-box method. This precise inference of the symptom shows the effectiveness of application context and iterative refinement of scopes used by the CLUE framework.

Customers confirmed that the localized events in a process group point to the exact execution causing low throughput; it turned out that all those processes were pinned onto a single processor instead of spreading over the 16 cores. The performance problem was resolved after fixing this code behavior.

VII. Conclusions

Black-box performance diagnosis with OS kernel events is hard due to lack of semantic information, which deters the interpretation of service level behaviors. Previous approaches focused on service behaviors caught by network events. However, there are many other types of service behaviors critical for diagnosis accuracy.

We presented new analytic techniques to analyze a wider set of service behaviors beyond network oriented requests. We have expanded trace modeling that captures implicit communication patterns such as synchronization and inter-process communication. Subsequently, we applied unsupervised learning with statistical analysis on the structured data for a holistic view of low level OS kernel events. With the mature and wide availability of kernel tracing tools such as SystemTap and ETW, we believe our proposed techniques can be efficiently deployed and used in the Cloud for service performance analysis where users can collect system traces from either guest OS within VMs or the hosting OS.

References

[1] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," in *Proceedings of the 2005 Ottawa Linux Symposium (OLS)*, 2005.

[2] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247415.1247417

[3] *Event Tracing for Windows (ETW).* http://msdn.microsoft.com/library/en-us/perfmon/base/event_tracing.asp, 2002.

[4] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu, "Fay: extensible distributed tracing from kernels to clusters," in *SOSP '11*, 2011.

[5] T. Marian, H. Weatherspoon, K.-S. Lee, and A. Sagar, "Fmeter: Extracting indexable low-level system signatures by counting kernel function calls," in *Middleware*, ser. Lecture Notes in Computer Science, P. Narasimhan and P. Triantafillou, Eds., vol. 7662. Springer, 2012, pp. 81–100.

[6] R. J. Moore, "A universal dynamic trace for linux and other operating systems," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 297–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=647054.715769

[7] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vpath: Precise discovery of request processing paths from black-box observations of thread and network activities," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, 2009.

[8] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *Linux Symposium*, June 2006.

[9] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers, "Recovering system metrics from kernel trace," in *Proceedings of the Linux Symposium*, 2011.

[10] P.-M. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 77–87, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773932

[11] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *SOSP '03*, 2003.

[12] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "Wap5: black-box performance debugging for wide-area systems," in *Proceedings of the 15th international conference on World Wide Web*, ser. WWW '06. New York, NY, USA: ACM, 2006, pp. 347–356. [Online]. Available: http://doi.acm.org/10.1145/1135777.1135830

[13] H. Liu, H. Zhang, R. Izmailov, G. Jiang, and X. Meng, "Real-time application monitoring and diagnosis for service hosting platforms of black boxes," in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, May 2007, pp. 216–225.

[14] E. Koskinen and J. Jannotti, "Borderpatrol: isolating events for black-box tracing," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08. New York, NY, USA: ACM, 2008, pp. 191–203. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352613

[15] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, and Z. Zhang, "Precise, scalable, and online request tracing for multi-tier services of black boxes," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2011.

[16] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley, CA, USA: USENIX Association, 2004, pp. 18–18.

[17] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: detecting the unexpected in distributed systems," in *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 9–9. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267680.1267689

[18] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'11. USENIX Association, 2011.

[19] M. Margulies, M. Egholm, W. E. Altman, and et al., "Genome sequencing in microfabricated high-density picolitre reactors," *Nature*, vol. 437, no. 7057, pp. 376–380, Sep. 2005.

[20] J. Shendure, G. J. Porreca, N. B. Reppas, X. Lin, J. P. McCutcheon, A. M. Rosenbaum, M. D. Wang, K. Zhang, R. D. Mitra, and G. M. Church, "Accurate multiplex polony sequencing of an evolved bacterial genome," *Science*, vol. 309, no. 5741, pp. 1728–1732, Sep. 2005.

[21] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Automated anomaly detection and performance modeling of enterprise applications," *ACM Trans. Comput. Syst.*, vol. 27, no. 3, pp. 6:1–6:32, Nov. 2009.

[22] *SystemDirector mevalet*. http://www.nec.co.jp/cced/SystemDirector/.

[23] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.

[24] *memcached*. http://memcached.org/.

[25] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/331499.331504

[26] *The Java PetStore 2.0*. http://www.oracle.com/technetwork/java/index-136650.html.

[27] C. Stewart, K. Shen, A. Iyengar, and J. Yin, "Entomomodel: Understanding and avoiding performance anomaly manifestations," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, aug. 2010, pp. 3 –13.

[28] J. Rhee, H. Zhang, N. Arora, G. Jiang, and K. Yoshihira, "Software system performance debugging with kernel events feature guidance," in *IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2014.

[29] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *OSDI'04*, 2004.

[30] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *DSN '02*, 2002.