Software System Performance Debugging with Kernel Events Feature Guidance

Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Kenji Yoshihira
NEC Laboratories America
Princeton, New Jersey 08540 USA
Email: {rhee, huizhang, nipun, gfj, kenji}@nec-labs.com

***Abstract*—To diagnose performance problems in production systems, many OS kernel-level monitoring and analysis tools have been proposed. Using low level kernel events provides benefits in efficiency and transparency to monitor application software. On the other hand, such approaches miss application-specific semantic information which can be effective to differentiate the trace patterns from distinct application logic. This paper introduces new trace analysis techniques based on event features to improve kernel event based performance diagnosis tools. Our prototype, AppDiff, is based on two analysis features: system resource features convert kernel events to resource usage metrics, thereby enabling the detection of various performance anomalies in a unified way; program behavior features infer the application logic behind the low level events. By using these features and conditional probability, AppDiff can detect outliers and improve the diagnosis of application performance.**

***Keywords*—performance debugging; system management; black-box monitoring; kernel events monitoring; distributed systems**

I. Introduction

Localizing performance anomalies in enterprise software systems have constraints different from conventional debugging environments. A monitoring agent for deployed systems should run efficiently with minimal overhead. Otherwise performance overhead of the monitoring software can hide the performance problems. While using the source code is effective for debugging in a typical scenario using commercial off-the-shelf software, administrators lack knowledge of the software. Moreover the source code of third party software components is often not available. Efficient black-box performance debugging is in high demand for production systems.

There have been several major approaches in traditional program debugging to localize performance anomalies in software. Traditional debugging [1], [2], [3] takes application programs executed in a debugging mode (e.g., single-step mode, ptrace, and dynamic compilation). While this approach provides a fine grained control and detailed information deep inside the program, it inhibits the observation of workload with the full fidelity for deployed systems because the overhead prevents the program from executing in full production-level speed.

Other approaches [4], [5], [6], [7], [8] embed a monitoring agent into the program by modifying source code or binary code and observe the program's behavior in a fine grained way. These approaches have advantages in understanding application status since internal information such as transactions, functions, or objects are visible to the monitoring agent [5], [7], [8]. Such approaches would be effective if source code is available or software relies on common libraries whose structures are well known. However, proprietary software or third-party components provided in the binary format will limit the applicability of such approaches.

978-1-4799-0913-1/14/$31.00

There is another family of approaches that uses low level events (e.g., system calls) to determine application status [9], [10], [11], [12]. Since such methods do not rely on the knowledge on software internals, they are called black-box approaches. Compared to other approaches, these solutions have the advantage to monitor software without involving the constraints in the application level. Tak et. al.'s work [12] models application behavior by building paths of kernel events. The collected traces in this work are treated as homogeneous traces. In real deployment scenarios, high complexity and diverse set of functions of enterprise applications generate a highly dynamic set of operations in the application behavior. The assumption on homogeneous behavior can trigger false alarms since different types of application activities can be considered as anomaly.

Our Contributions: We propose a new kernel event analytic system, called *AppDiff*, to address the aforementioned challenges in the existing approaches. It introduces several data analytic technologies to solve the problem.

We use a new metric called *system resource features* which represent the resource usage statistics of kernel events. This information is obtained by applying resource transfer functions to kernel events. These features provide means to observe behavior of kernel events in various resources. By monitoring any significant change in each resource metric, we can identify anomalous behavior of kernel events.

The diverse sets of application behavior do occur and they need to be differentiated. Otherwise, comparing and treating such heterogeneous patterns as homogeneous would cause false positives. We solve this problem by inferring the characteristic of application code using the distribution of system call vector. This information is called *program behavior features* and they represent unique characteristics of application code.

Based on these new features, we apply *conditional probability* on the analysis of traces. It enables us to distinguish trace data generated from different application logic and improve the analysis by identifying outliers. The proposed approach provides an accurate and reliable anomaly detection when multiple types of application logics exist in traces by considering their similarities and dissimilarities.

II. AppDiff Design

Large scale software systems typically consist of one or more software components (i.e., nodes or tiers). An example is a three-tier system that is composed of a web server, an application server, and a database server. We use low level kernel events for these nodes to monitor their status and detect anomalous status of the software systems. The kernel events generated from this system represent their interactions and operation units. For example, a multi-tier system receives a web request and serves the web content generated from multiple nodes. This request processing is one example of the transaction of this application system. The kernel events used in this request processing are collected into one trace based on the causal relationship of the events and we call it a *transaction trace*.

We propose a performance diagnosis tool called *AppDiff* that operates with two inputs: the kernel events from a training operation of software and the kernel events from the system monitored for anomaly. These inputs are processed in the *trace generation* component and turn into two sets of traces : a set of transaction traces for a normal scenario (T_T) and another set of traces from the same system in the deployment scenario (T_M). On these inputs, first *Anomaly Trace Localization* is applied to determine anomaly in a global scope. If any anomaly is detected, a finer grained anomaly test, *Anomaly Event Localization*, is applied.

A. Transaction Trace Generation

Kernel traces are constructed by using the causal dependency of kernel events. For instance, when a service request comes to a webserver which sends back a reply to a client, multiple components such as an application server and a database server in the monitored system work together to serve the request. The overall activity of the system to serve a request is considered as *one transaction*. The kernel events corresponding to this transaction are collected and organized into one trace file. A group of kernel events for a process is called a *transaction unit (TU)*. A trace file includes TUs involved in a transaction and the causal relationship among TUs. Kernel event monitoring software (e.g., SystemTap [11], DTrace [9], Dprobe [10], Mevalet) does not provide the causal relationship. Hence, such relationship is constructed by traversing kernel events and following causal patterns such as communications and IPCs.

There are several fields for each kernel event necessary to apply our analysis: time stamps, process ID numbers, serial numbers of TUs, event types (e.g., a system call and an interrupt), and event details (e.g., IP address for a connection).

B. Anomaly Trace Localization

For each trace file, two new data features (System Resource Features and Program Behavior Features) are generated to improve the quality of anomaly detection.

1) System Resource Features: From raw traces, AppDiff extracts a new perspective of trace information regarding resource usages called *System Resource Features*. The resource transfer functions are defined to convert kernel events to a set of resource features. The examples are as following; the

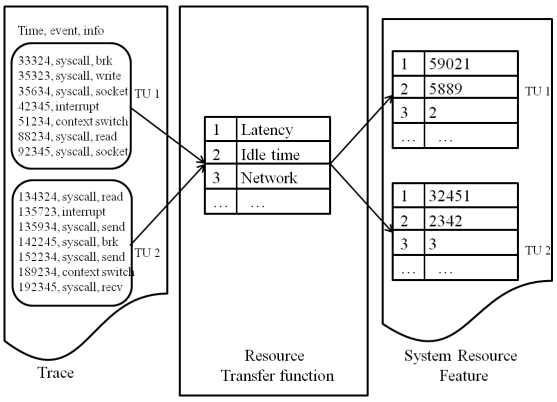

Fig. 1. Generation of System Resource Features.

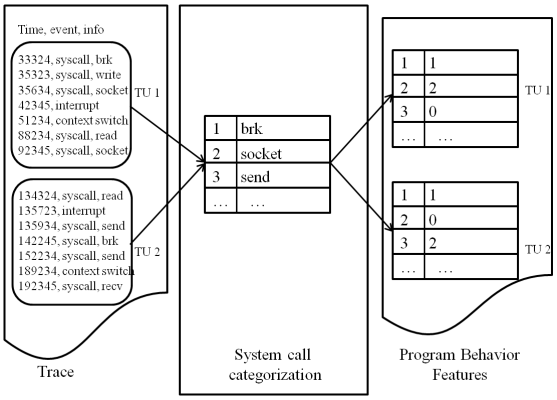

Fig. 2. Generation of Program Behavior Features.

latency function measures the time spent on the transaction unit. The idle function measures the idle time periods in the process scheduling using the kernel events. Network activity is measured by the number of kernel events for networking (e.g., TCP accept, TCP receive). The functions for other resource events are similarly defined. Figure 1 shows the conversion process from trace events to system resource vectors.

2) Program Behavior Features: In black-box approaches, internal knowledge on application program is missing. To solve this problem we provide a new perspective of trace information that infers what application code triggers kernel events. The insight behind our solution is that *the same user code in a similar workload is likely to trigger a similar set of system call events*. In other words, our hypothesis is that the system call vector distribution can indicate the characteristics of similar application code behavior. We observed that many software systems have multiple functions which produce various transaction traces. This scheme is the key mechanism to allow our anomaly detector to differentiate the kernel-level traces caused by different code. We call this information *Program Behavior Features*. Figure 2 illustrates how this information is generated. For each transaction unit, the counts of system calls are computed and stored in a vector.

Once the system resource features are extracted from the two sets of training and monitored execution traces, the distribution of resource features are compared to detect anomaly. If any significant difference in a resource feature is detected, its index is reported and conditional data mining component

is executed to find any outliers.

C. Conditional Data Mining and Anomaly Detection

Given the list of resource features having anomalies, this component further investigates and reduces potential false positives. It uses program behavior features to rule out outliers, which are the cases that the anomalous resource usages are caused by different kind of application logics. Specifically, first traces are clustered based on program behavior features. Then the clusters from data set are tested whether they have counterparts in another set in terms of behavior patterns. For the cluster pairs having the matched behavior patterns, anomaly detection is applied in the cluster level.

1) Trace Clustering: We used a connectivity based clustering (i.e., hierarchical clustering) with a threshold (θ_C) in the distance function. It uses the agglomerative method (bottom-up approach). To connect clusters, single-linkage (nearest neighbor approach) is used. We use the Euclidean distance for the comparison of system call vectors. Each pair of traces are compared by applying this distance function on their program behavior features, and they are connected if their distance is less than the threshold value.

2) Conditional Data Mining: Using low level OS kernel events enables us to monitor application software without knowing its details. However, the lack of details on user level code becomes a challenge to diagnose software behavior. In order to understand irrelevant application context and improve this black box analysis, we use conditional probability in the analysis of anomalies. The conditions provide refined context to the set of events analyzed. Therefore, they enable us to improve the detection of anomaly.

Several conditions that represent program characteristics can be used in this analysis. Program behavior features are one example of the condition. Let R denote the anomaly condition derived from system resource features (e.g., latency $>$ 5 seconds) and B denote a trace set having common program behavior features. Then the probability used in this anomaly detection is represented as $P(R|B)$. This is the conditional probability using the traces clustered based on the behavior features that reflect the application logic triggering system call events.

Anomaly Detection in Clusters: Anomaly is detected by using the factors, $P(R_T|x \in C_{T,k})$ and $P(R_M|x \in C_{M,k})$, which reflect the probability of anomalous trace conditioned by the cluster ($C_{T,k}$: a cluster for a normal trace, $C_{M,k}$: a cluster for a monitored trace). They are calculated by:

$$P(R_T|x \in C_{T,k}) = \frac{P(R_T \cap x \in C_{T,k})}{P(x \in C_{T,k})}$$

$$P(R_M|x \in C_{M,k}) = \frac{P(R_M \cap x \in C_{M,k})}{P(x \in C_{M,k})}$$

Anomaly is determined by comparing the resource features conditioned by the clusters from a normal trace and a monitored trace.

D. Anomaly Event Localization

If any anomaly is found in the comparison of a cluster pair, a finer grained analysis in the event level is performed on the clusters. The assumption in this analysis is that the compared traces have a similar structure, but their differences can reflect the root cause of the performance problem.

Given a pair of clusters, $C_{T,k}, C_{M,k}$, the anomaly resource indices, and policies as input, this component compares two clusters and generates the *diff* of traces. For each anomaly resource index, a representative trace is selected from each trace cluster based on policies. One policy for selecting a trace in the set of normal traces is to choose the trace that is closest to the average. Another policy is to select the trace closest to the behavior features of the monitored run. This choice is based on the assumption that it will distinguish the patterns unique to the problem. In the monitored execution the trace with the highest latency could be chosen since it is likely to represent performance anomaly symptoms. The selected pairs of traces and TUs are compared using the Longest Common Subsequence (LCS) algorithm commonly used in `diff` utilities. The differences are examined by developers to understand the root cause of the anomaly.

III. Evaluation

We implemented AppDiff as the analysis framework for the NEC proprietary kernel tracing tool, Mevalet, that supports diverse system platforms such as Linux, HP-UX, and Microsoft Windows. We present a case study of a three tier system, J2EE PetStore 2.0 [13] service testbed. We configured this system with four nodes as a three-tier system with the redundant application servers: an Apache web-server, two Jboss application servers, and a mysql database server. The webserver balances the load to two application servers in a round-robin fashion. The logs have over three million events in total.

The training run is executed without introducing problematic symptoms. In the monitored run, we introduced a performance bug to validate AppDiff's capability that localizes the problem. This bug is called *db_misconfig* bug [14] where the administrator misconfigured the DNS in one of the application servers; therefore the requests going to the second application server fail to connect to the database server. Instead the error pages are returned to the clients.

1) Trace Analysis: **Clustering using Program Behavior Features**: In order to distinguish the traces from different application logics, transaction traces are clustered using *program behavior features*. Specifically the system call vector of each trace is compared with other traces' vectors and their distances are calculated. In the evaluation we used several clustering thresholds of 1%, 5%, 10%, and 20%, and the number of the produced clusters vary depending on the threshold. For instance, when the threshold is 20%, the traces for the training run are grouped into 9 clusters and the traces for the monitored run are categorized into 18 clusters.

Figure 3 and 4 show the distribution of means and variances of these clusters. The execution paths of the requests are mainly two ways. If the requests are sent to the Jboss server without the misconfiguration, they are returned to the clients with a short latency. On the other hand, if requests

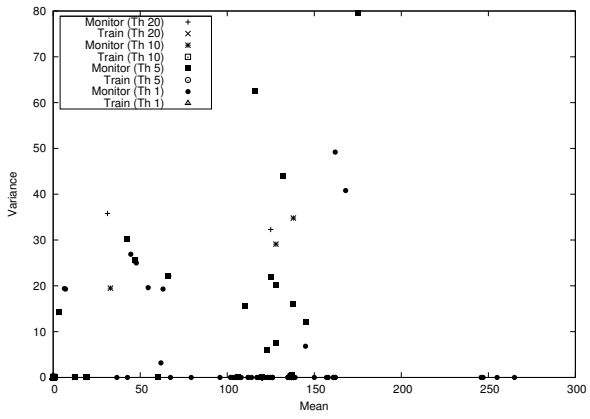

Fig. 3. Means and Variances of Clusters.

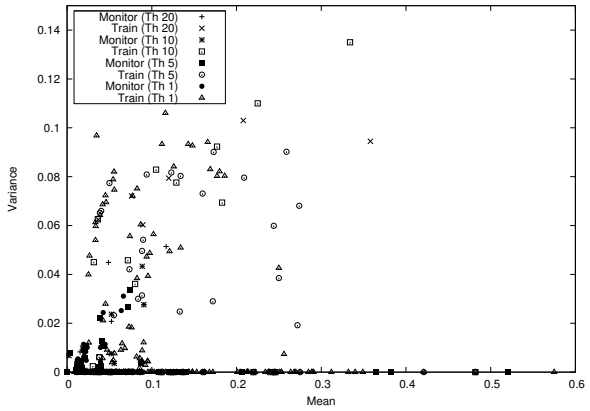

Fig. 4. Means and Variances of Clusters (A zoomed view of Figure 3 in the bottom left corner).

are sent to the misconfigured application server, they experience significant delay. These two behaviors are mixed in the monitored data set. After clustering, the similar behavior patterns handling the requests are grouped together; therefore, two major request processing behavior are separated. For example, in Figure 3 we can confirm the clusters having significantly high means and variances of latency all belong to monitored runs. The clusters of the training runs stay at the left bottom corner, which is further zoomed in Figure 4. This figure in fact shows not only the clusters from the training run but also some clusters from the monitored run having low latency. These clusters reflect the requests processed in the JBoss application server without anomaly. This process shows the clustering based on program behavior features is effective in distinguishing normal and abnormal traces based on application behavior patterns.

2) Anomaly Event Analysis: The performance problem in this experiment generates a lot of traces with extended latencies. These traces increase the means and variances of the clusters. Therefore the anomalies are detected from multiple clusters. We may wonder what kind of root cause led to this problem. To find additional clues regarding this symptom, AppDiff proceeds to a finer level inspection of traces in the event level.

With the clustering threshold $\theta_C = 0.2$, we obtained 18 cluster groups for the monitored data set and 9 cluster groups

```
938          EID_SVC_BGNgettimeofday        EID_SVC_BGNgettimeofday
939                    EID_SVC_END_LX                 EID_SVC_END_LX
940          EID_SVC_BGNgettimeofday        EID_SVC_BGNgettimeofday
941                    EID_SVC_END_LX                 EID_SVC_END_LX
942          EID_SVC_BGNgettimeofday        EID_SVC_BGNgettimeofday
943 L                  EID_SVC_END_LX
944 L       EID_SVC_BGNclock_gettime
945 L                  EID_SVC_END_LX
946 L                EID_SVC_BGNfutex
947 L                   EID_PSAVE_LX
948 L                  EID_PRESUME_LX
949 L               EID_FUTEXWAIT END
```

Fig. 5. Diff Operation on Kernel Events in Transaction Units. (Left: a TU from the execution with DB misconfiguration, Right: a TU from the training execution without performance anomaly).

for the training data set. The mean latency of the clusters for the monitored run vary from 0.00221 to 125.

Conditional probability is applied to the clusters generated by program behavior features. There are other clusters having low latencies which are from properly handled requests by another application server without anomaly. Without clustering, the trace selection might have been biased by normal data patterns which have significantly low latencies compared to the abnormal case.

For event level comparison, two traces are selected each from the monitored run and the training run. For the monitored run, we selected the trace with the longest latency. We ranked all traces in the monitored run with the ascending order of latency and choose the top trace in the list. For a normal trace, we chose the one having the average characteristic in latency. There are several cluster candidates. We chose the cluster group with the most number of TUs and selected the trace which is closest to the average of the cluster.

Diff Events: Figure 5 shows an excerpt of the result from Anomaly Event Analysis. In the comparison, the left hand side is the TU of the anomaly trace and the right hand side is the TU of a normal trace. `EID_SVC_BGNXXXXX` events represent the system calls with the names `XXXXX`. From the comparison result, we can determine the frequent usages of *clock_gettime* and *gettimeofday* system calls. These events are related to the time measurement code.

This event pattern indicates that there are frequent time out events in the abnormal trace when there is high latency. Due to the misconfiguration in the connection to the database, application server experiences excessive time out and delays when it handles requests. As shown in this example, event level analysis can help users to understand the root cause of performance problems.

IV. Conclusion

We present a new black-box approach to analyze kernel events of enterprise software systems and investigate the root cause of performance problems. While many existing black-box approaches handle kernel events homogeneously in their analysis, major software systems have a variety of application logics in their transactions. Without differentiating such underlying execution patterns, anomaly detection faces inaccuracy. We propose new trace analysis techniques to extract semantic information regarding resource usages and application logic behind the traces by using kernel event patterns. Our result shows improvement in performance diagnosis and anomaly detection by pruning out outliers.

REFERENCES

[1] "gdb: The gnu project debugger."

[2] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[3] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*, 2007.

[4] "Microsoft phoenix research compiler (http://research.microsoft.com/phoenix/)."

[5] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012.

[6] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Conference on Compiler Construction (CC'02)*, 2002.

[7] 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 *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011.

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/1629087.1629089

[9] 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

[10] 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

[11] 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) (Jul 2005)*.

[12] 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.

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

[14] 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.