European Space Operations Centre
Robert-Bosch-Strasse 5
D-64293 Darmstadt
Germany
T +49 (0)6151 900
F +49 (0)6151 90495
www.esa.int

LEVERAGING SYSTEM PERFORMANCE METRICS AND EXECUTION LOGS TO PROACTIVELY DIAGNOSE SYSTEM OF SYSTEMS PERFORMANCE ISSUES

EXECUTIVE SUMMARY

Prepared by	**SOFTCOM-INT TEAM**
Reference	**EGOS-STU-GEN-LLMS-ES-1001**
Issue/Revision	**01/00**
Date of Issue	**18/02/2016**
Status	**Issued**

European Space Agency
Agence spatiale européenne

APPROVAL

Title LEVERAGING SYSTEM PERFORMANCE METRICS AND EXECUTION LOGS TO PROACTIVELY DIAGNOSE SYSTEM OF SYSTEMS PERFORMANCE ISSUES EXECUTIVE SUMMARY	
Issue Number 1	**Revision Number** 0
Author SOFTCOM-INT TEAM	**Date** 18/02/2016
Approved By	**Date of Approval**
James Eggleston	**19/02/2016**

CHANGE LOG

Reason for change	Issue Nr.	Revision Number	Date
First draft	1	0	19/02/2016

CHANGE RECORD

Issue Number 0	**Revision Number 0**		
Reason for change	**Date**	**Pages**	**Paragraph(s)**

DISTRIBUTION

Name/Organisational Unit

Table of Contents

I. INTRODUCTION

Syer et al. published a paper titled “Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues” in the International Conference on Software Maintenance in 2013. In that paper, the authors describe an automated approach that combines execution logs and performance counters (e.g. memory heap usage) in order to assist performance analysts in diagnosing memory-related performance issues (e.g. memory leaks) that appear in load tests. The approach is depicted in Figure 1.

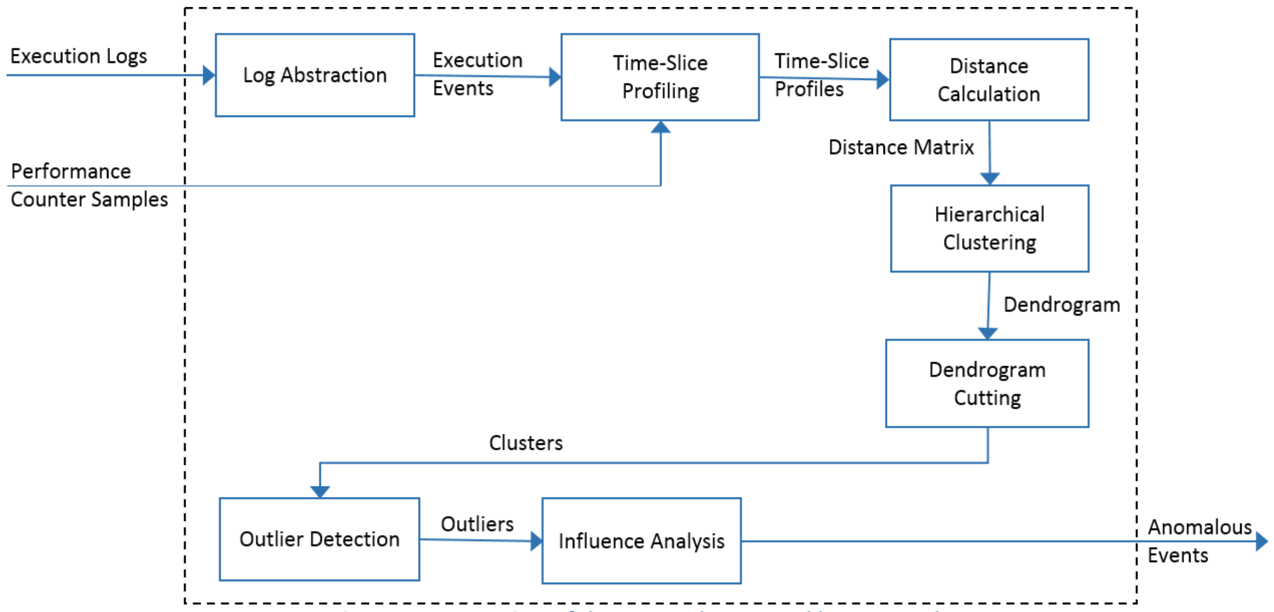


Figure 1. An overview of the approach proposed by Syer et al.

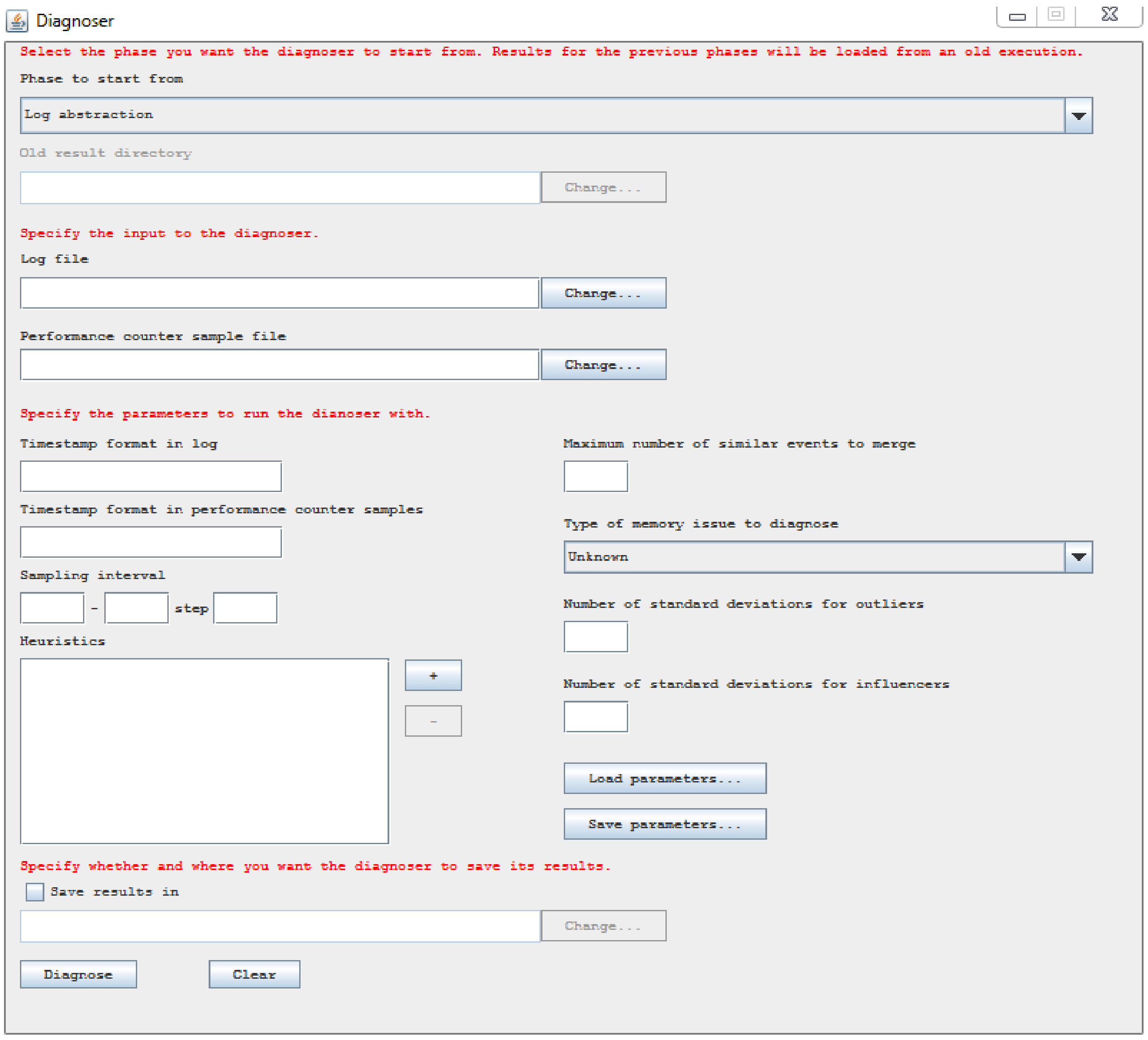
We analysed the approach proposed in that paper by Syer et al. We then designed and built a prototype correlation engine that uses that approach, and applied that engine on both simulation and real-world applications, in order to characterise it. We finally developed a proactive error detection system prototype that proactively collects execution logs and performance counters from running applications, and uses the approach proposed by Syer et al. in order to correlate them, and raise alarms that will give a massive advantage to performance analysts in case of failures.

The rest of the summary is organised as follows: Section II describes the prototype correlation engine in detail. Section III outlines a typical use case of the engine. Section IV presents the results on the load tests we ran on the engine. Section V presents the proactive error detection system prototype we built. Section VI concludes the summary and presents future work.

II. PROTOTYPE CORRELATION ENGINE

A performance analyst performs a load test on an application, and detects a possible memory issue by visually inspecting the memory usage plot. He applies the approach proposed by Syer et al. to the execution logs and the performance counter samples he collected during the test. The approach flags a few events, which the performance analyst analyses manually, and reports his conclusions, along with the issue itself, to the developers.

The prototype correlation engine is a tool that aims to support the use case scenario described above. The user gives as input to the engine the execution log and the performance counter sample file, and receives as output the anomalous events (if any) that are most likely to be responsible for the issue. Each event is accompanied by a confidence value.


Diagnoser

Select the phase you want the diagnoser to start from. Results for the previous phases will be loaded from an old execution.

Phase to start from: Log abstraction

Old result directory

Specify the input to the diagnoser.

Log file

Performance counter sample file

Specify the parameters to run the dianoser with.

Timestamp format in log

Timestamp format in performance counter samples

Sampling interval - step

Heuristics

Maximum number of similar events to merge

Type of memory issue to diagnose: Unknown

Number of standard deviations for outliers

Number of standard deviations for influencers

Specify whether and where you want the diagnoser to save its results.

Save results in

Figure 2. Prototype correlation engine.

The user can adjust the different parameters required by the approach (e.g. the heuristics to be used for the log abstraction), store the intermediate and the final results for further inspection, and/or re-apply the approach on the same data set more than once starting each time from a later phase and re-using results from old executions for the previous phases. All the above functionality enables the user to experiment and retrieve even more information about his data sets.

The tool provides both a command-line and a graphical user interface that offer the exact same functionality. The latter is shown in Figure 2.

In order to support the use of the prototype correlation engine, we built an extensible monitoring tool that can be used to monitor one or more performance counters (e.g. memory heap usage) on one or more running processes. The user specifies what they want to monitor and how, and the tool produces as a result one or more performance counter sample files (one for each performance counter), which can be later used as input to the correlation engine.

The monitoring tool does not know how to monitor performance counters; it uses external tools to perform that task. Examples of such tools include SystemTap, DTrace and JMX. Each one of those tools might be available only on certain operating systems, might be able to monitor only certain performance counters, might provide different accuracy, or might be suitable for monitoring applications that are written only in certain programming languages. The tool selects automatically every time the most appropriate external tool based on the performance counters that are to be monitored and the processes that those counters are to be monitored for.

At the moment, the tool can monitor only the used heap size, and can use JMX, SystemTap, DTrace, or the information provided by the process information pseudo-file system (i.e. in the /proc/[pid]/maps files) in order to monitor that counter. The tool can be extended to support more performance counters, and to use more performance counter collection tools and/or techniques.

III. USE

A load test on an application uncovers a performance issue (e.g. a memory leak) in it. The performance analyst needs to collect more information about the issue, before he can report it to the developer, so that the latter can find out what causes it and fix it.

The analyst feeds the execution logs and the performance counter samples he collected during the load test to the correlation engine. The engine identifies the log lines that are more likely to be responsible for the issue, and flags them. Depending on how familiar the analyst is with the application, he might be able to immediately rule out some of the reported log lines. The analyst gives the final list with the suspicious log lines to the developer.

The developer analyses the log lines one by one. He can locate a log line in the execution log, in order to find out what else took place in the system around the same time, or he can look up the piece of code that generated that line, in order to check it for any hidden bugs.

IV. TESTS

In order to test the correlation engine, we built two simulation applications: the spiker and the leaker. The spiker is written in Java and causes memory spikes, whereas the leaker is written in C++ and causes memory leaks. Both applications generate suitable execution logs.

The mechanics behind the spiker are the following: The spiker constructs processes as sequences of normal and abnormal events that repeat indefinitely. Each process is executed by a separate thread. Once started, the process executes the events in its sequence one by one. The execution of a normal event causes only a line to be printed on standard output, whereas the execution of the abnormal event generates also a spike before printing the line. Each process executes its events with a random sleep time between them. Once the process has finished executing its event sequence, it sleeps for a random amount of time, and then it starts over. The mechanics behind the leaker are similar. Both applications are parameterisable (e.g. by the size of the leak or the spike, or by the number of processes).

Apart from the simulation applications described above, we also ran load tests on two open-source applications: Apache Tomcat and MySQL. Apache Tomcat is a web server and servlet container written in Java, whereas MySQL server is an open-source database written in C++.

We ran over 10,000 load tests on those four applications. The duration of the load tests ranged between 10 and 60 minutes. We injected leaks between 100B and 10MB, and spikes between 1KB and 100MB in the source code of those applications. Table 1 presents one data set for each one of the four applications, as well as the results we got when we applied the correlation engine on them.

	Spiker	**Leaker**	**Apache Tomcat**	**MySQL server**
Programming language	Java	C++	Java	J++
Memory issue	Memory spike (1K)	Memory leak (100B)	Memory spike (10M)	Memory leak (1K)
Load test duration	10 minutes	10 minutes	10 minutes	10 minutes
Sampling interval	20 seconds	10 seconds	20 seconds	2 seconds
Number of samples	30	59	30	296
Number of log lines	5,296	5,803	8,691	36,733
Number of flagged events	1	2	24	6
Number of reported lines	2	4	63	44
Precision	100%	50%	4%	17%

Table 1. Results.

It is important to note that the load tests on the real-world applications made us realise that the formula that is used to calculate the influence of an event favours in a way the more frequent events. That caused frequent normal events to be flagged, when rare abnormal ones were just ignored. We therefore decided to make a simple change to the way overly influential execution events are identified. More specifically, we identified as influencers also events that do not normally occur, but in that particular cluster, where the issue has been detected, they occur at least once. The results reported in Table 1 were produced with the use of the above fix.

V. PROACTIVE ERROR DETECTION SYSTEM

Execution logs and performance counters are periodically collected for an application, as it runs, the approach proposed by Syer et al. is automatically applied to them, and the results are archived. If at any moment the application crashes or the performance analyst detects a memory issue, the analyst can examine the events that were flagged by the approach around the time of the incident, and possibly discover what caused it.

We built a proactive error detection system prototype in order to support the scenario described in the previous paragraph. Figure 3 gives an overview of that prototype.

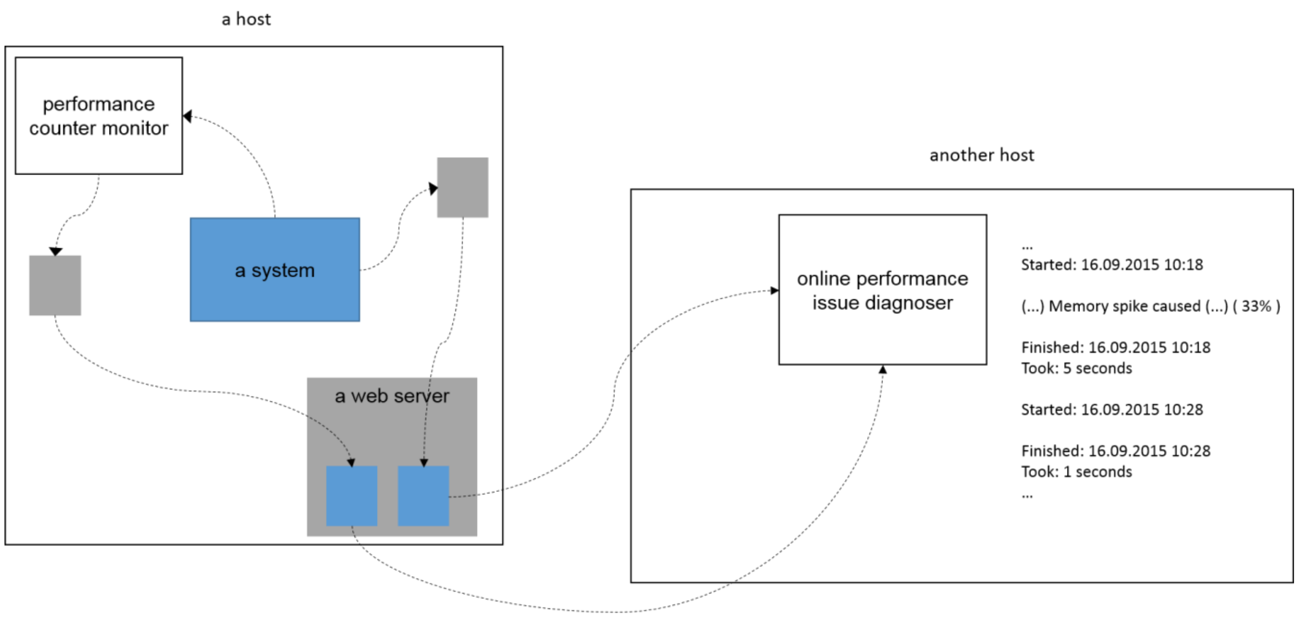

a host
performance counter monitor
a system
a web server
another host
online performance issue diagnoser
...
Started: 16.09.2015 10:18
(...) Memory spike caused (...) (33%)
Finished: 16.09.2015 10:18
Took: 5 seconds
Started: 16.09.2015 10:28
Finished: 16.09.2015 10:28
Took: 1 seconds
...

Figure 3. Proactive error detection system prototype.

In order to build the prototype, we made the following assumptions: (1) A system is running somewhere, and is producing execution logs. (2) Performance counters are periodically (e.g. every 2 seconds) sampled for the system. (3) Both execution logs and performance counter samples are periodically (e.g. every 10 minutes) copied into a web accessible directory under well-known names. That way the directory contains at any point in time the execution logs and the performance counter samples collected from the system during the last period.

The core of the prototype is a tool that periodically (e.g. every 10 minutes) downloads the execution logs and the performance counter samples from the web accessible directory, runs the approach proposed by Syer et al. on them, and reports the results. Figure 3 depicts the information flow in a typical use case scenario of the system. The directory to download the logs and the samples from, the execution period, and the parameters to be used for running the approach, are configurable.

It is important to note that the proactive error detection system prototype we built (despite of what its name implies) does not detect performance issues. Performance analysts or testers still need to discover them. Only now, once they have uncovered them, they can immediately search through the archive for possible causes.

VI. CONCLUSIONS AND FUTURE WORK

We analysed the approach proposed by Syer et al. in their paper titled "Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues", and we built a prototype correlation engine that is based on it. We then ran more than 10,000 load tests on two purpose built applications (one written in Java and one written in C++) and two real-world applications (Apache Tomcat and MySQL).

In most cases, the engine worked (i.e. it did flag the event that was related to the performance issue, for some parameter set). But in most cases, the engine flagged also several events that were not related to the issue. Nevertheless, we concluded that, as long as the number of flagged events is manageable (e.g. less than 20), the analysts and developers can still benefit from using the engine, since it can help them get started.

We also observed that in the cases when the engine did not work, most of the times it was the influence analysis phase that failed. However, in most of those cases, the outlier detection phase did work. Thus, we reached the conclusion that even in those cases when the approach fails to provide information about what caused an issue, it will at least provide information about when the issue was caused.

In our future work, we plan to make the engine work on large data sets, provide a better solution for the influence analysis phase, and extend it to other types of performance issues (e.g. CPU-related ones). We also plan to enable analysts and developers to provide feedback to the engine about the events they have ruled out, so that the time they spend to analyse the reported events is not wasted. Then, in the cases when none of the reported events was the correct one, they will at least be able to run the correlation engine again on the same data set, ignoring those events, in order to get a fresh list of possible suspects to examine.