Lichen: an event-based framework for characterizing sporadic performance bottlenecks in applications

Author: Hewlett-Packard Company, Vishal Kanaujia, Navin Sati, Anshu Bhola

Research Disclosure Database Number 542065

Published in June 2009

Research Disclosure Journal, ISSN 0374-4353
© Kenneth Mason Publications Ltd
www.researchdisclosure.com
info@researchdisclosure.com

Lichen: an event-based framework for characterizing sporadic performance bottlenecks in applications

Abstract

With the continuous improvement in hardware technology, a big onus of ensuring optimal performance lies on the application itself. Application performance analysis is the preferred method to identify the possible bottlenecks in performance critical systems. Application performance characteristics are not uniform over its execution period. A lot of applications experience sporadic performance bottlenecks, which happens only under certain conditions. It is critical to look at the profile around the time of the bottleneck to understand the conditions for these bottlenecks. Traditional profilers either collect data over the entire run, where the profile at the time of bottlenecks gets diluted in the full profile, or they collect data after detecting the bottleneck, where the crucial data about conditions existing prior to the bottleneck is missing.

This paper proposes a generalized system for dynamically discovering & profiling potential bottlenecks that may bear sporadic nature in an application. This system works in a novel manner to gather application performance data at user-defined critical points which are qualified by a breach of threshold for resource utilization, and captures samples only around a performance bottleneck, delivering a concise sample set which provide most relevant, accurate and effective hindsight for critical bottlenecks.

Introduction

The class of performance critical systems typically manages its critical resources (e.g. CPU, memory etc) by imposing a consumption threshold (e.g. CPU utilization<25%) for its applications [1, 2, 3]. In such an environment, threshold breach is considered as a performance glitch in the application. In order to achieve optimal performance, these applications need to be profiled to provide hindsight into application level performance bottlenecks. But, profiling an application in such an environment poses couple of major problems: *(1) Discovering "true" critical performance points and (2) non-uniform nature of application performance characteristics.*

The *first* problem emanates because a typical profiler produces a continuous and indiscriminate stream of samples [4, 5, 6]. Eventually, these samples do not separate critical bottlenecks from other non-critical conditions, masking the real issues. It happens because a performance critical point is biased towards samples that have highest frequency in the sample set. Besides, these profilers collect data *after* detecting a bottleneck, missing the critical data about the conditions that led to a performance bottleneck and hence bring inaccuracy in the profile. Besides application performance characteristics bear non-uniform nature, seeding the *second* problem. Depending on the application execution behavior, a sporadic peak may or may not occur and hence may demand multiple runs of the application to catch the same glitch again. Hence, profiling of a long running application coupled with collecting, storing and analyzing huge sample sets proves to be inaccurate, difficult and time-consuming.

This paper proposes an automatic system comprising a generic statistical profiler and an application monitor. This synergy efficiently and effectively discovers performance critical points and facilitates true and optimal results with least user intervention.

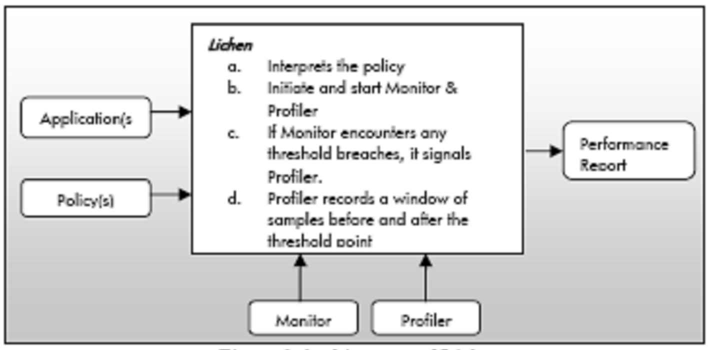
Proposed Solution

Lichen is a collaborative system that prominently consists of the following two components:

Profiler (P): A statistical profiler that generates a stream of samples during the execution of the application. It should be able to maintain a window of '2N' samples from the sample stream. The size of the window could be user defined and can be derived from various factors e.g. sampling frequency etc.

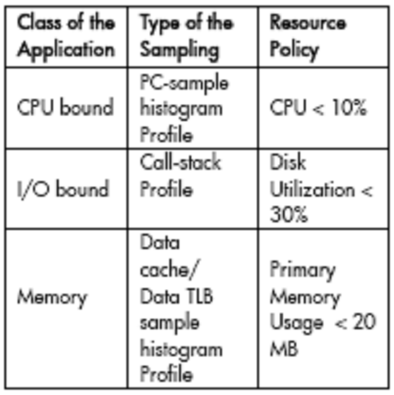
Monitor (M): It should be able to quantify the desired resource characteristics for an application and must be equipped with a signaling system.

The framework of Lichen expects that the system proposes a policy for the applications deployed on it. An end-user needs to classify the application by its expected execution behavior and prepare a 'resource policy' for critical resources with desired consumption threshold. Subsequently this policy is interpreted and Lichen selects the most suitable sampling type to achieve best profiling information (Table 1). Post initialization, Monitor starts to keep track of the desired resource threshold and profiler starts generating a sample stream.



a. Interprets the policy
b. Initiate and start Monitor & Profiler
c. If Monitor encounters any threshold breaches, it signals Profiler.
d. Profiler records a window of samples before and after the threshold point

Figure 1 Architecture of Lichen



Class of the Application	Type of the Sampling	Resource Policy
CPU bound	PC-sample histogram Profile	CPU < 10%
I/O bound	Call-stack Profile	Disk Utilization < 30%
Memory	Data cache/ Data TLB sample histogram Profile	Primary Memory Usage < 20 MB

Table 1: Lichen resource policy

During the course of its execution, if a process breaches the resource threshold, Monitor immediately notifies the Profiler about this event. Profiler reacts by mapping this event to a sample (S) in the sample stream and captures a window of '2N' samples that includes 'N' samples each *on either side* of sample event 'S'. The captured samples are stored in a repository by the Profiler. This cycle of event generation by Monitor and subsequent action by Profiler continues till either the application exits or the user explicitly shuts down Lichen. Finally, Lichen requests the Profiler to read in the collected samples from the repository and analyze them.

This approach holds following advantages:

- Collected sample set is *more effective and unbiased* because samples are collected only at defined critical points. This sample set effectively corresponds to the performance hotspots encountered during the application execution and hence discovers *true* critical bottleneck in the application.
- The approach of collecting the window of 2N samples ensures a complete *cause & effect* view of a performance bottleneck. First N samples provide data about the conditions that led to the bottleneck and next N samples capture the effect of the bottleneck.

- It is *very effective* to profile an execution around asynchronous events like data cache bottleneck, OS page fault etc. Such events bear sporadic nature and may affect the application performance. For example, the data cache misses or OS page faults might have an infrequent spike in applications; as the monitor collects the data cache or data TLB samples around the spike, it will give details even for the short lived bottleneck. Similarly, resource policy for delays in network communication, thread synchronization etc. can be set up for the application. If the application threads block for more than some given percentage as per the resource policy, Lichen can gather samples around these blocking points to find out the call stacks and blocking primitive details leading to the thread blocking state. The advantage here is that the tool can capture details around various such points in a pinpointed way and the resource policy can be tweaked to gather data about bottlenecks at a granularity level chosen by the user.

Experimental Results

The environment for Lichen was simulated with the help of available tools: HP Caliper [3] and HP Glance [6] on a simple *matrix multiplication* application which makes huge number of calls to a trivial function *lichen::foo()*. The resource policy is a threshold of 75% for CPU consumption and has been prepared in HP Glance Advisor syntax. HP Glance continuously monitors the process (*lichen.exe*) for CPU utilization value per second and if threshold is breached, it prints the timestamp of such an event.
After starting HP Glance, the application is run with HP Caliper to get a trace of samples time-stamped with interval-time counter and their corresponding IP addresses. After the application exits, we map the HP Glance's time stamps to time-stamps of HP Caliper's samples. Figure 2 shows that CPU utilization threshold gets breached at various sporadic time-stamps T1- T12.

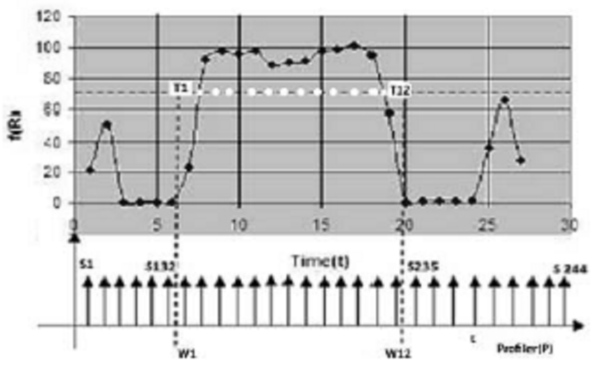


Figure 2: The resource utilization graph

- f_t :R $\rightarrow$ *Th* where *f* is a continuous function such that *f*(R) = Resource Utilization for a resource R.
- CPU Utilization Threshold = 75%
- Sample Windows: W

Sampled IP	IP- count with HP Caliper	Lichen's result
lichen::foo()	135	0
lichen::matmul()	103	103
libc.so.1::free()	6	0

Table 2: Profile of lichen.exe

These events can be mapped to sample points around S132-S235 and in a set of sample windows W1-W12. Thus, with Lichen, Profiler ends up collecting a total of only 103 samples for profile report. Table-2 shows the data to highlight the difference between the sample frequency count generated by Lichen and HP Caliper profiler. It is evident that most critical optimization points lies in *lichen::matmul()* instead of HP Caliper reported *lichen::foo()*.

Alternative Approaches

Traditional profilers have taken a *start and stop* approach for application profiling and processes data between the start and stop events [4, 5]. It expects the end-user to manually discover a performance slowdown during the application execution and processes only a timed-window of past samples from the sample stream. This approach provides an incomplete solution due to following reasons:

1. It profile only cause of a performance slowdown and overlooks the effects. Lichen collects post-event 'N' samples too to make sure that complete application behavior is reflected around a performance bottleneck.
2. Apple Shark [5] recourse on end-user’s ability to discover a performance slowdown. It may bring inaccuracy to the final results as an end-user may overlook/miss most critical bottlenecks. Besides, for long running application this manual intervention becomes a tedious task. In contrast, automated nature of Lichen and its well defined method to discover critical performance hotspot ensures complete and accurate results.
3. It is able to discover and profile an***y single*** performance bottleneck as the tool needs to be stopped immediately after a performance slowdown is perceived by the user.

References

[1] HP OpenVMS Systems Documentation
[2] Grid computing
[3] Understanding Application Quotas with Google App Engine
[4] HP Caliper performance analyzer
[5] Optimizing your Application with Shark 4
[6] HP Glance

Disclosed by Vishal Kanaujia, Navin Sati, Anshu Bhola, Hewlett-Packard Company