DYVISE: Performance Analysis of Production Systems

Research Demonstration

Steven P. Reiss
Department of Computer Science, Brown University, Providence, RI. 02912
spr@cs.brown.edu

Abstract

Many of today's complex systems are multi-threaded servers that effectively run forever and need to work under varying loads and environments. Understanding the behavior of such systems, both their normal behavior and any abnormal behavior, is a necessary step in their development, deployment, and use. We have developed a performance analysis framework that is targeted for such systems. The framework deals with a variety of performance issues including cpu usage, memory utilization, I/O behavior, thread interactions, and event handling. Data is collected within a fixed overhead set dynamically by the user. The tool's user interface shows the behavior of the system as it is happening and lets the user browse through the execution history.

1. Introduction

Programmers often want to know about the dynamic behavior of their systems. They want to know about performance anomalies and how they can eliminate them. They want to know where the system is spending its time, what it is allocating, how long it takes to respond to an event, where the memory leaks are, and why they are not getting expected performance out of multiple threads. They want to know how long it takes to respond to an event. They want to know about potential bottlenecks and memory leaks.

While a large number of tools have been developed to address these issues, most of these tools do not address modern software systems. Today's systems are typically more complex than previous ones and thus harder to understand. They typically involve multiple threads that can interact in non-obvious ways. They involve significantly more code, with even a relatively simple one incorporating millions of lines of source when libraries are considered. Moreover, they are often long running systems, servers that are designed to run continuously and forever. A detailed summary of related work and of our system can be found in [1].

Traditional profiling tools typically slow the program down significantly, often by 25%-100% or more, in order to get the necessary information. They often provide only postmortem analysis which is not helpful when considering systems that never terminate or when one is interested in a particular time slice of the execution rather than the overall execution. They are also generally limited in domain, concentrating on one aspect of performance or another, and not attempting to address all aspects simultaneously. All this makes them inappropriate for analyzing server systems.

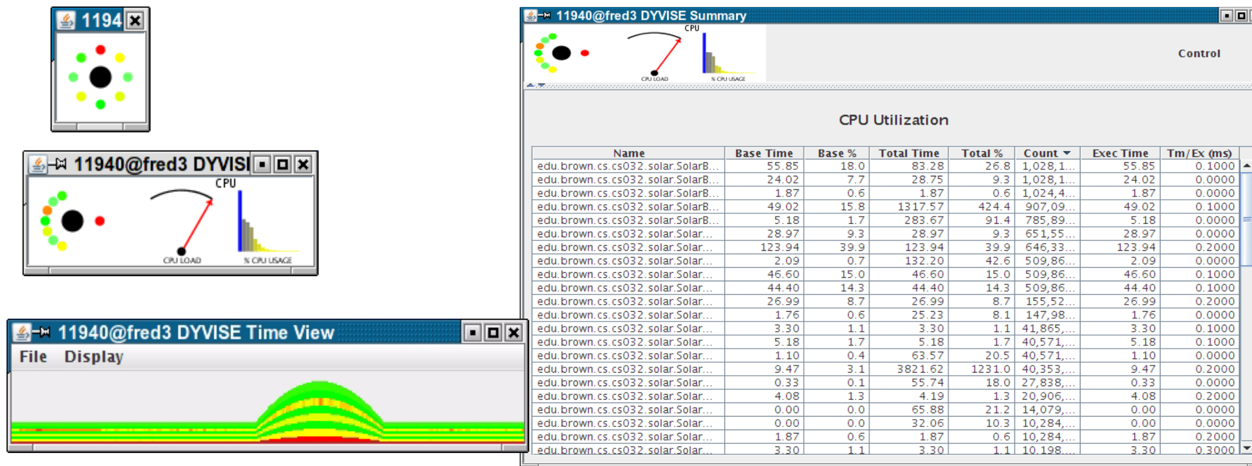
Our tool, DYVISE, provides on-the-fly analysis of complex systems within a user-settable overhead. It deals with multiple performance domains and provides the user with a flexible, details-on-demand interface. The current implementation works for Java systems.

2. DYVISE Overview

DYVISE consists of a set of communicating processes. Java agents (both library and JVMTI) are attached to the user process as needed. These talk to a monitor process that controls the instrumentation, gathers the results, and does appropriate analysis. A separate user interface gets the analyzed data from the monitor. The components can all run on separate machines so that the overhead of monitoring and the user interface do not affect the original process.

DYVISE uses a combination of techniques to achieve fixed-overhead performance analysis. It uses sampling techniques to get a coarse view of performance. Based on these results it then uses dynamic instrumentation to get finer levels of detail. In order to guarantee a fixed overhead, it varies the time between samples and does detailed instrumentation only for relatively short periods of time. This is accomplished by treating instrumentation as a resource allocation problem where the time between stack samples is varied and code is instrumented and run based on appropriate schedules determined from dynamically established priorities.

Detailed instrumentation is achieved statistically. For example, if the system wants to estimate the number of times a routine is called over the run, it would create a patched version of the class containing the routine that counted entries, swap this instrumented version in for ten seconds, and then project the counts so obtained to the overall run. By changing the sampling interval and the frequency and duration of the instrumentation appropriately, the system provides guaranteed limits on the overhead while offering statistically significant performance information. In addition



Name	Base Time	Base %	Total Time	Total %	Count	Exec Time	Tm/Ex (ms)
edu.brown.cs.cs032.solar.SolarB...	55.85	18.0	83.28	26.8	1,028,1...	55.85	0.1000
edu.brown.cs.cs032.solar.SolarB...	24.02	7.7	28.75	9.3	1,028,1...	24.02	0.0000
edu.brown.cs.cs032.solar.SolarB...	1.87	0.6	1.87	0.6	1,024,4...	1.87	0.0000
edu.brown.cs.cs032.solar.SolarB...	49.02	15.8	1317.57	424.4	907,09...	49.02	0.1000
edu.brown.cs.cs032.solar.SolarB...	5.18	1.7	283.67	91.4	785,89...	5.18	0.0000
edu.brown.cs.cs032.solar.Solar...	28.97	9.3	28.97	9.3	651,55...	28.97	0.0000
edu.brown.cs.cs032.solar.Solar...	123.94	39.9	123.94	39.9	646,33...	123.94	0.2000
edu.brown.cs.cs032.solar.Solar...	2.09	0.7	132.20	42.6	509,86...	2.09	0.0000
edu.brown.cs.cs032.solar.Solar...	46.60	15.0	46.60	15.0	509,86...	46.60	0.1000
edu.brown.cs.cs032.solar.Solar...	44.40	14.3	44.40	14.3	509,86...	44.40	0.1000
edu.brown.cs.cs032.solar.Solar...	26.99	8.7	26.99	8.7	155,52...	26.99	0.2000
edu.brown.cs.cs032.solar.Solar...	1.76	0.6	25.23	8.1	147,98...	1.76	0.0000
edu.brown.cs.cs032.solar.Solar...	3.30	1.1	3.30	1.1	41,865,...	3.30	0.1000
edu.brown.cs.cs032.solar.Solar...	5.18	1.7	5.18	1.7	40,571,...	5.18	0.1000
edu.brown.cs.cs032.solar.Solar...	1.10	0.4	63.57	20.5	40,571,...	1.10	0.0000
edu.brown.cs.cs032.solar.Solar...	9.47	3.1	3821.62	1231.0	40,353,...	9.47	0.2000
edu.brown.cs.cs032.solar.Solar...	0.33	0.1	55.74	18.0	27,838,...	0.33	0.0000
edu.brown.cs.cs032.solar.Solar...	4.08	1.3	4.19	1.3	20,906,...	4.08	0.2000
edu.brown.cs.cs032.solar.Solar...	0.00	0.0	65.88	21.2	14,079,...	0.00	0.0000
edu.brown.cs.cs032.solar.Solar...	0.00	0.0	32.06	10.3	10,284,...	0.00	0.0000
edu.brown.cs.cs032.solar.Solar...	1.87	0.6	1.87	0.6	10,284,...	1.87	0.2000
edu.brown.cs.cs032.solar.Solar...	3.30	1.1	3.30	1.1	10,198...	3.30	0.3000

FIGURE 1. Compact, intermediate, detailed and time views of performance

to periodic detailed instrumentation, the system allows continuous instrumentation when the overhead involved is low enough and priorities allow.

To accommodate different types of analysis, the framework supports performance agents we call *proflets*. Proflets are split into several pieces to handle priority determination, data collection based on stack sampling, detailed data collection using instrumentation, data analysis, and visualization. The overall framework provides both run time and analysis support for the proflets, an appropriate scheduling mechanism, support for dynamic code instrumentation, support for using JVMTI, and user interface support.

The current set of proflets provides for a variety of different analyses including CPU performance, IO behavior, socket usage, memory allocations, heap utilization, thread interactions, thread timing, events and callbacks, and program phase.

3. DYVISE User Interface

The user interface we developed is shown in Figure 1. The view at the upper left shows the status of the different proflets, with hue encoding whether the proflet thinks there might be a problem and brightness encoding the confidence the proflet has in that assessment. This view is quite compact and unobtrusive so that multiple visualizations can be displayed at one time to monitor multiple servers or different processes in a complex system. Clicking on the red dot for CPU time in the first view animates the window to include the second view, shown below the first, which shows summary information from the CPU proflet. Tool tips are used in this view to provide additional details on the meter and the bar graph. Finally, clicking again brings up the third view which shows the detailed performance information. Each proflet provides its own set of summary information and detailed tables.

An additional time view, show in the lower left, provides a summary of the proflet states over the whole run and lets the programmer zoom in and select a particular time frame for more detailed analysis.

All views are continually updated as the process runs. This provides a view of immediate program performance. In addition, the user can clear the statistics at any point in the run and can dynamically enable and disable performance gathering. These facilities let the user focus on particular performance problems.

4. Experience

We have used DYVISE to analyze the performance of a wide variety of long-running systems including a particle simulation, a system for learning coding style, a peer-to-peer programming framework, a 3D pinball program, a web crawler, a web server, and a semantics-based code search engine, a news analyzer, and a economic data crawler. From the results, we have obtained the insights necessary to significantly improve the performance of some of these, for example, we were able to reduce the clock time for a test run of the search engine by 50%.

We are continuing to work on the system, providing additional proflets, including application-specific proflets and proflets for anomaly detection. Moreover, we are working on additional user interfaces that will provide insights into the immediate rather than cumulative performance of the system.

The code for the framework is available at: ftp://ftp.cs.brown.edu/u/spr/dyvise.tar.gz.

5. Acknowledgements

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

6. References

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