Delivering Quality in Software Performance and Scalability Testing

Khun Ban, Robert Scott, Kingsum Chow, and Huijun Yan
Software and Services Group, Intel Corporation
{khun.ban, robert.l.scott, kingsum.chow, huijun.yan}@intel.com

Abstract

We all are witnessing the amazing growth in today's computing industry. Servers are many times faster and smarter than those of just a few years ago. The need to ensure the applications run well with a large number of users on the Internet has continued and is getting more attention as the move to cloud computing is gaining momentum. To address the question whether the software system can perform under load, we look to performance and load testing. Performance testing is executed to determine how fast a system performs. It is typically done using a particular workload. It can also serve to verify other quality attributes, such as scalability, reliability and resource usage. Load testing is primarily concerned with the ability to operate under a high load such as large quantities of data or a large number of users. We use performance and load testing to evaluate the scalability of a software system.

In this paper, we characterize the challenges in conducting the application performance tests and describe a systematic approach to overcome them. We present a step-by-step data driven and systematic approach to identify performance scalability issues in testing and ensuring software quality through software performance and scalability testing. The approach includes how to analyze multiple pieces of data from multiple sources, and how to determine the configuration changes that are needed to successfully complete the stress tests for scalability. The reader will find it easy to apply our approach for immediate improvement in performance and scalability testing.

Biography

Khun Ban is a staff performance engineer working with the System Software Division of the Software and Services Group at Intel. He has over ten years of enterprise software development experience. He received his B.S. degree in Computer Science and Engineering from the University of Washington in 1995.

Robert Scott is a staff performance engineer working with the System Software Division of the Software and Services Group. Bob has over ten years of experience with Intel improving the performance of enterprise class applications. Prior to Intel, he enjoyed twenty years engaged in all aspects of software development, reliability and testing, and deployment.

Kingsum Chow is a principal engineer working with the System Software Division of the Intel Software and Services Group. He has been working for Intel since receiving his Ph.D. in Computer Science and Engineering from the University of Washington in 1996. He has published more than 40 technical papers and has been issued more than 10 patents.

Huijun Yan is a senior performance engineer working with the System Software Division of the Software Services Group. Huijun has over seven years of enterprise software performance experience. Huijun received her MS degree in Computer Science from Brigham Young University in 1991.

1 Introduction

Software performance testing has been a difficult challenge to master. It is as much an art as a science. To help meet the challenge, there have been great tool improvements in the recent years to help automate the testing processes and even generate test cases based on the application source code. There are also numerous commercial and open source tools to help with load generation by giving the user the ability to record interactions and replay them as if many users are performing those steps with some random think time between each step.

While the tools are improving, the problem set has also increased in size and scope. With the increasing computing capability of modern servers, it has become increasingly difficult to test the full scaling of the server as many external, often much slower, components can get in the way. Performance optimization has been a challenge for application servers [1]. In addition, modern platform features like power management can also complicate the testing by not giving us an accurate reading of the CPU activities. The principle goal of this paper is to outline the challenges we faced during performance testing and offer the solutions that worked well for us.

Enterprise software systems can involve many systems and components. Figure 1 illustrates one such setup. On the left, there are many clients interacting with the servers through some public interface such as web services, JavaServer Pages (JSP), Java Servlets, or just static HTML pages. Load generators are often used to simulate these users. The clients are then connected to multiple gateway switches that route the traffic to some faster inter-connecting switches which connect multiple database and application servers on the right. Additional storage devices may be attached as needed. In this paper, we concentrate on server software and restrict the system under test (SUT) to the servers and the components required to perform their function, i.e., the components inside the box in Figure 1. This does not imply that client system performance is not important, just that we can easily add more client capability when the client-tier becomes the bottleneck during testing. However, on the server we must identify any issue that may prevent it from scaling to support the maximum number of clients. In addition, clients may come and go, reboot whenever needed. Server systems are expected to have much longer up time when compared to client systems. As such, it is important to test the server to ensure that it can remain healthy and meet service quality levels for a longer duration and during peak usage. To meet this requirement, a load generator can be used to stress test the platform. Using the load generator, we are able to vary the duration and level of load on the system to understand how it performs in these different situations. In Figure 1, load generators would simulate many client users on the left side of the diagram. The right side of the diagram represents the SUT and will have some requirements to test for, such as supporting some number of users within some response time constraints.

It is not always possible to drive server systems to full capacity to meet the requirements due to different bottlenecks or issues in the setup. These issues can range from hardware constraints, operating system configuration, network settings, the choice of Java Virtual Machine (JVM) and its parameters, to multiple dependency locks. In most cases, these items do not become issues until the system is stressed under high load.

As the capability of the SUT grows, e.g., either through hardware or software upgrades, it may be necessary to augment the various components in the test setup. For example, if the current load driver cannot provide enough transaction loads to fully utilize the test system, a second driver system may need to be added. This process can domino to other components in the system. Additional loads processing may increase the demands placed against the database. If adding a second instance of the application on the SUT is required to fully utilize the platform's resources, this may require the addition of some form of load balancing. We can utilize each platform's performance metrics to monitor each resource and determine if there is a performance bottleneck. It is normal to need to go through this process multiple times during the life of an application.

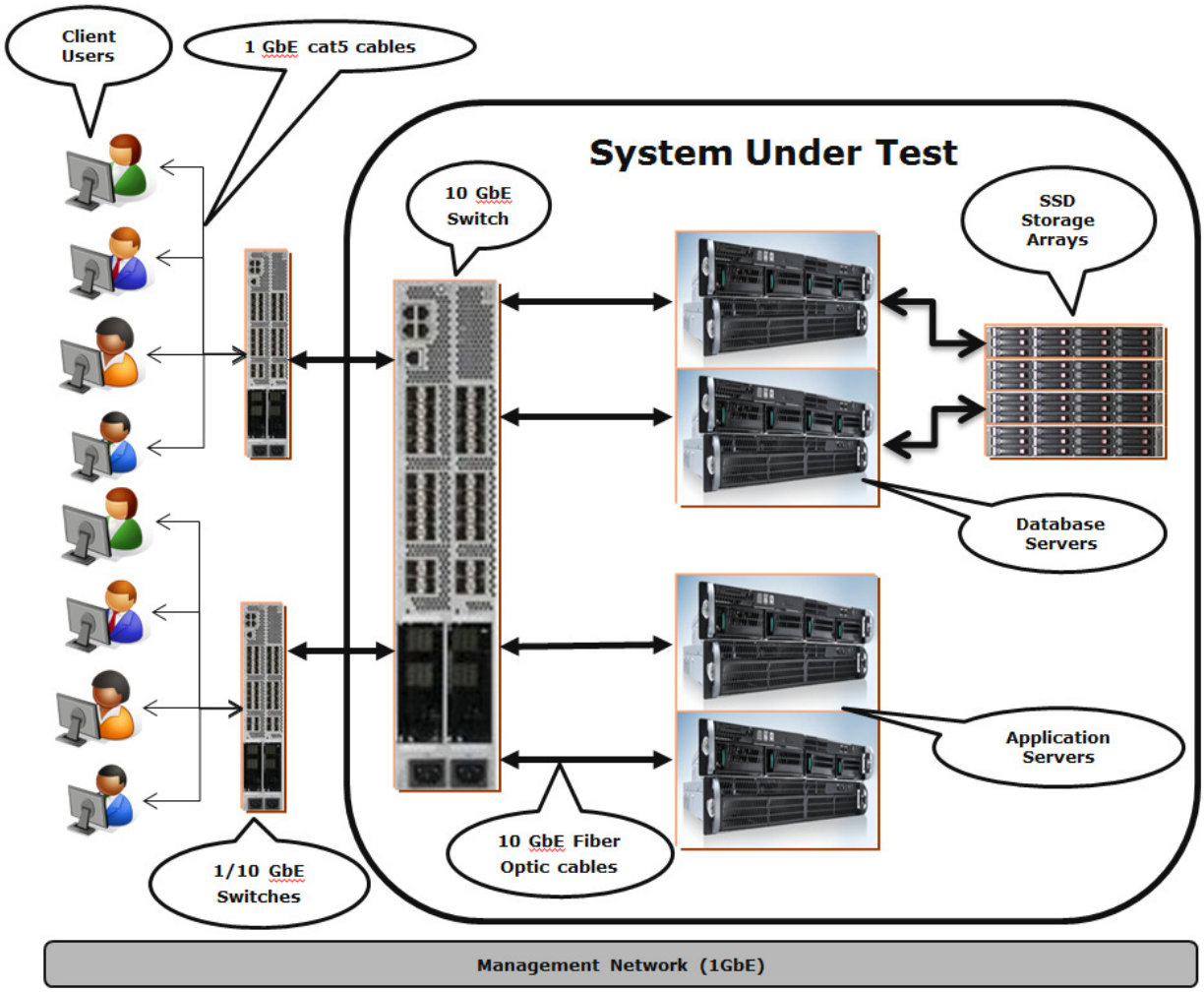

Figure 1. Performance and scalability testing for a multi-tier deployment environment.

Performance and scalability testing may run for hours, days, or even weeks. We need to consider memory and storage requirements. Will the memory management be able to keep performing well? What about the database system as more and more records are added? For example, in a short run of a Java application, the garbage collector may not need to collect the old space or perform a full collection. Instead, it may just work on the young space and leave the old space untouched to keep the garbage collection time short. For the database, accessing a small table with only few entries is usually fast. However, as the tables grow, indexes grow, and proper table design becomes important. Therefore, longer tests runs are required to detect these kinds of issues.

The contributions of this paper are:

1. Characterizing the challenges of performance and scalability testing, and
2. A systematic approach to overcome these challenges

2 A systematic approach to performance testing

Performance testing of enterprise software solutions usually involves multiple computers interconnected over a network. One popular deployment model is to separate the solution to multiple tiers. The mid-tier

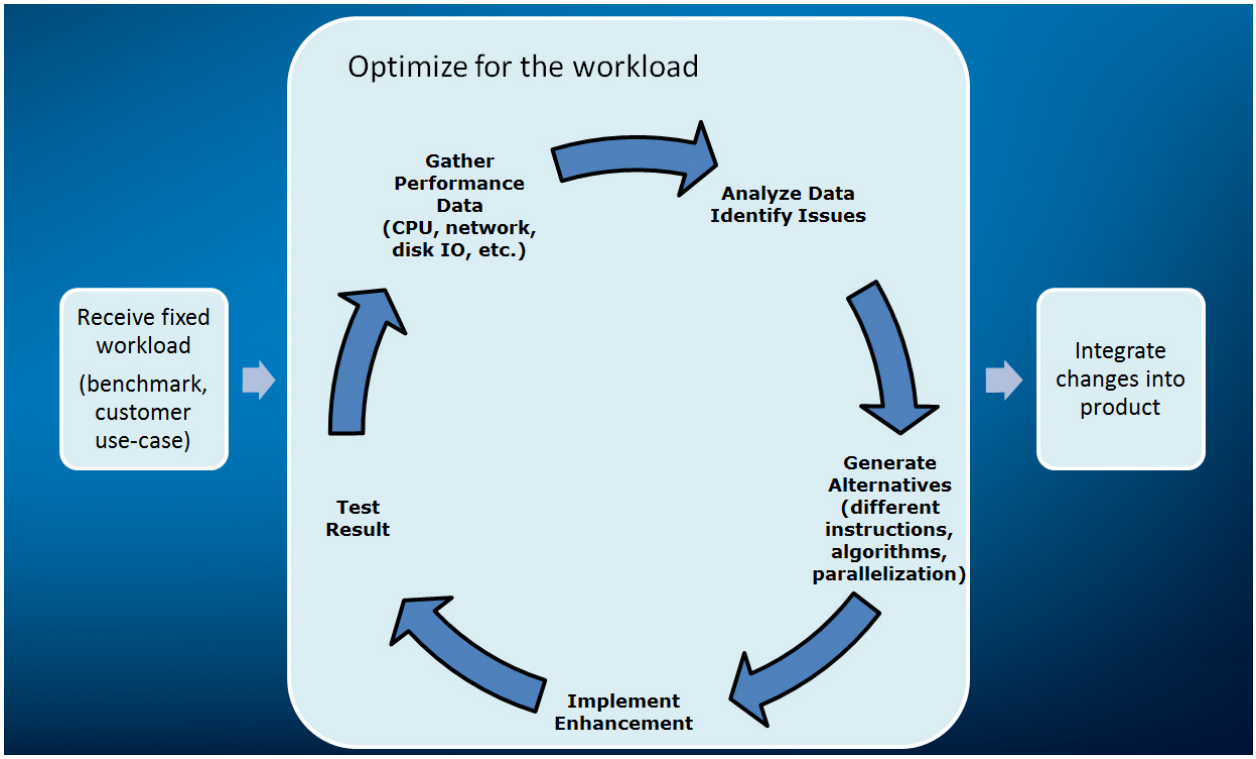

Figure 2. An iterative process for performance and scalability testing

generally handles the business intelligence and processing while the backend tier maintains the important information in some relational database. The client tier deals with the user interface and is not the focus of this paper.

Given the complexity involved, ensuring an adequate level of performance in this environment requires a systematic approach to avoid finger pointing, misdiagnosis, and focusing on the main bottleneck (root cause of the issue). Here we define a bottleneck as some resource that limits the performance of the solution as a whole. We assume that when the constraints on this resource are reduced, performance will increase.

We describe an iterative process to detect issues in performance and scalability testing in Figure 2. This process starts from the left when we are given a workload and some specifications such as the number of users to support and the response time required. From there, we will (1) gather performance data, (2) analyze the data to identify issues, (3) generate alternatives or enhancements that can be implemented, (4) implement enhancements and (5) test again to see if the fixes improve performance or meet the performance specification.

Once one issue is fixed, another issue may surface, so this approach will be repeated until sufficient bottlenecks have been resolved for the platform to perform to expectations. This does not mean that all bottlenecks have been exposed. A change to the software or to the hardware platform may cause new issues to surface.

Using the right tools is essential for measuring a platform's behaviors. On all Linux distributions, there are system performance counters, such as CPU usage, IO wait time, page swap activity, or context switches per second, that can be used to understand and diagnose performance issues.

SAR (System Activity Reporter, available on all Linux distributions) is a system monitoring tool that can take snapshots of the system performance counters. Obtaining periodic snapshots is an effective way of

finding performance bottlenecks and determining whether further action is needed. Several counters we found to be important to capture on all the system involved in the setup are:

- **%usr:** The percentage of time the CPU is spending on user processes, such as applications, shell scripts, or interacting with the user.
- **%sys:** The percentage of time the CPU is spending executing kernel tasks, such as file and network I/O.
- **%wio:** The percentage of time the CPU is waiting for input or output from a block device, such as a disk.
- **%idle:** The percentage of time where the CPU can perform additional work but no process is ready to run.
- **cswch/s:** This counter represents the number of context switches the system performs per second. A context switch occurs when the operating system scheduler switches execution from one process or thread to another.

In addition to knowing the average CPU usage, it is also important to check the utilization of the individual processors. For example, if a single processor has high utilization and other processors utilizations are low, the load being placed on the single processor may be a bottleneck.

If IO wait is more than 2-3%, we need to find the root cause and address the issue. Long IO wait time can be caused by either the system swapping if it is low on memory, or the system is performing large file IO operations. It is essential to know the root cause of the IO wait time. For Java, it is necessary to check the JVM parameters for heap size and utilization of large pages, and the system setting for the allocation of large pages. For example, if we have a system with 12 GB memory and we set aside 10 GB for 2MB large pages, we only have 2 GB left for Linux and other processes that must use small pages (2 KB pages). Then if we set the JVM heap to be anything larger than 2 GB and did not enable large pages for the JVM, the system will be swapping.

The number of context switches is another important metric to monitor for multi-threaded applications. If this number is high, it may be an indication that threads within the application may be blocked on some locks and unable to continue processing. What is considered high is dependent on the platform and the number of logical processors available. Knowing the value from a comparable application and platform can be used as an indicator.

Servers of today can dynamically adjust the operating frequency of their processors and memory, and even disable some processors when the load is light. This provides a huge savings of power, cooling, and cost to operate the system. The power saving features of modern processors may present a challenge to scalability and performance testing. For example, if the tools report 45% CPU utilization, this can be misleading if the cores are running at a reduced frequency of 1.6 GHz instead of the normal 2.93 GHz. For performance and scalability testing, it is sometimes helpful to disable power management and dynamic frequency adjustment to get the accurate reading on the system activities.

3 Case studies

Two case studies are presented in this paper to illustrate applying the iterative process for performance and scalability testing. In the first case study, we show that it is important to collect detailed SAR performance data to root cause the problem. In the second case study, we show that additional data beyond SAR system performance data are needed to isolate a hardware configuration change.

3.1 Case Study 1

This scenario describes how the average CPU usage on the system can be a misleading indicator. In this case, the workload measures the amount of HTTP traffic between two systems. The test environment consists of 2 systems; a driver and an application server. The driver machine uses Faban [2] to send requests to the AppServer, and the AppServer responds to the driver's request. The performance metric is operations per second (ops); the higher the better. A customer reported that running this workload on

the newer server with more cores resulted in 10% lower performance than on their older system with fewer cores. The older system had two quad-core CPU running at 2.67 GHz with 4 MB of L3 cache. The newer system had 2-socket 6-core CPU running at 2.33 GHz with 8 MB L3 cache. On the new system, the average total CPU usage (user + system, no IO Wait) was only 27%. This workload was not very CPU demanding so they did not understand why they were observing lower performance.

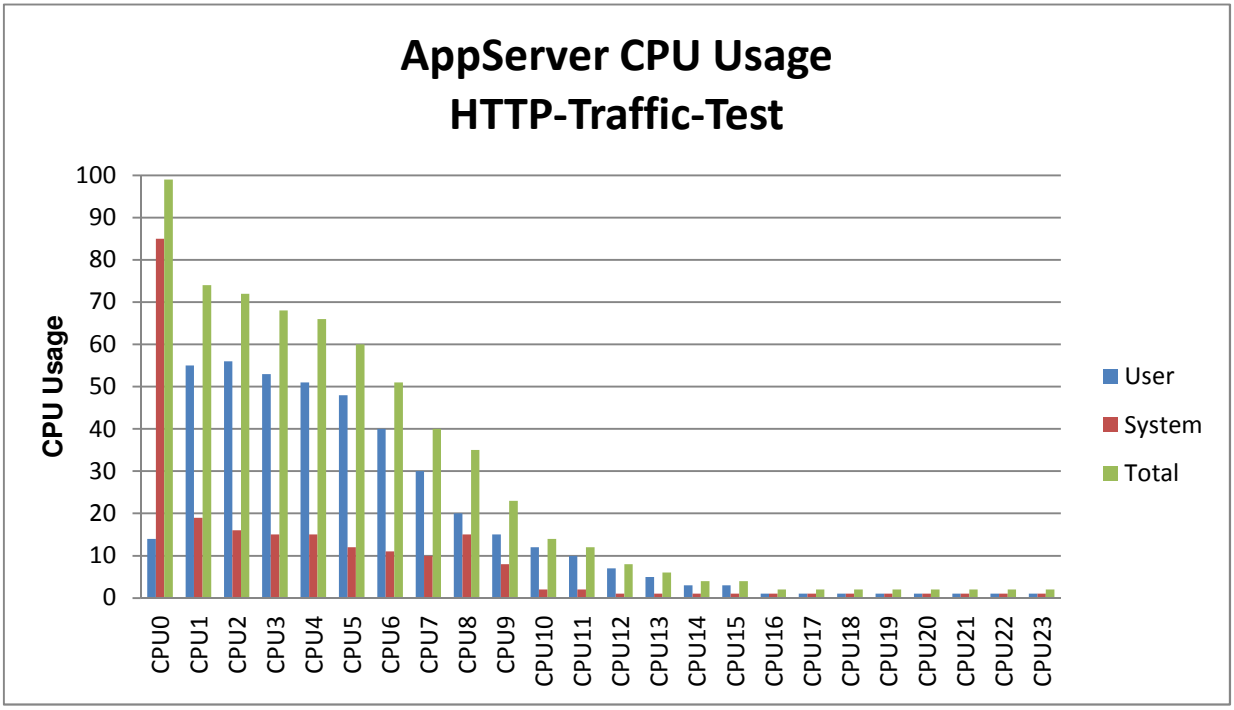

Figure 3. Case Study 1 – Distribution of utilizations across the CPUs.

In order to diagnose and solve this low performance issue, we performed the following steps.

1. We collected performance data from both the older and the newer system. We confirmed that the older system was performing better.
2. Using SAR performance counters data collected during step 1, we observed the CPU distribution outlined in Figure 3. It is quite clear the CPU 0 of the new server was heavily used even though the overall average was less than 30%. It turned out that the system interrupts were mainly handled by CPU 0 after examining the interrupts distribution (/proc/interrupts). The behavior was similar for the older system. However because the older system was running at faster frequency with fewer cores, the effects of this were less severe. We recommended to our customer to balance interrupt handling.
3. We modified the system interrupt handling to distribute the interrupt handling across all the cores equally. We then reran the workload and recollected the data.
4. The performance increased on both the older and the newer system. Additionally, the newer system now performed 20% better than the older system.
5. Then the customer asked why we cannot get even higher performance on the newer server since it still has more CPU headroom.
6. Beside CPU usage, SAR also reported network utilization. On our setup, we have only one 10 Gb/s connection between the driver and the application server. SAR reported our network usage was 9.5 Gb/s. Therefore the network was saturated and had become the new bottleneck.
7. The workload is no longer CPU bound; the available CPU power cannot push performance higher without first adding additional network capability.

3.2 Case Study 2

Sometimes even a good feature like power management can have unintended effects. A platform can reduce the frequency of its processors and save power if the processor's load is low (even for a few milliseconds). Today's platforms can also increase a processor's frequency if the surrounding cores are idle; keeping the same power level but increasing single threaded performance. One customer reported that their single threaded application was running much slower than expected after a BIOS upgrade to the application server. Their application consumed about the same CPU as before the upgrade, 10% total CPU usage, but achieved 50% less throughput and had a 50% higher response time.

Prior to upgrade, all power management features had been disabled so the processors would run at the default frequency (3.33 GHz in their case) all the time. After the upgrade, power management was enabled by default. Because the application required very little CPU, the power management feature reduced the processor frequency to just 1.59 GHz from 3.33 GHz. Therefore even when the CPU usage reported by the OS was about the same, the application was not consuming the same amount of CPU cycles as at the higher frequency. This issue was fixed by disabling power management features on the platform. Note: power management features will have little performance impact if the platform is typically driven at higher load levels.

4 Summary

Software performance testing involves testing not only the software application but also the underlying hardware, its configuration and usage. A failure to reach the specified performance criteria can come from not only the software application itself, but also from the server platform, and/or from the network and system configurations on all systems involved.

When running high loads in conducting scalability and performance testing, it is important to collect adequate sets of system performance metrics to ensure the performance criteria is met, and help to root cause when it is not. We have provided an approach to address the problem. The approach includes a set of system performance counters to collect and how to interpret the data collected. By applying the analysis from the system performance counters to fix the problems in the test environment, we can increase the confidence of the performance test results. We presented case studies to illustrate the problem if the performance test methodology is incorrect, a wrong conclusion may be drawn from an otherwise correct piece of software application that has been written to the specification. We also shared a set of best practices for performance and scalability testing.

Acknowledgments

Jonathan Morris reviewed and provided helpful suggestions to improve the clarity of the paper.

References

1. Khun Ban, Kingsum Chow, Yong-Fong Lee, Robert Butera, Staffan Friberg, and Evan Peers, "Java Application Server Optimization for Multi-core Systems" http://software.intel.com/en-us/articles/java-application-server-optimization-for-multi-core-systems/
2. Faban Harness and Benchmark Framework. http://java.net/projects/faban/