Performance Analysis In the Age of the Internet: A New Paradigm for a New Era

Alberto Avritzer, Richard Farel, Kenichi Futamura, Mehdi Hosseini-Nasab, Anestis Karasaridis, Varsha Mainkar, Kathleen Meier-Hellstern, Paul Reeser, Patricia Wirth
AT&T Labs, Middletown, NJ 07748
(avritzer, farel, futamura, hosseini, karasaridis, mainkar, meierhellstern, preeser, pwirth) @att.com

Frank Huebner
Concert Technologies, Middletown, NJ 07748
frank.huebner@concert.com

David Lucantoni
DLT Consulting, Ocean NJ 07712
david.lucantion@att.net

Abstract

IP application performance must frequently be assessed using empirical methods that combine performance testing and modeling. We have developed an approach to identifying bottlenecks using performance signatures. A performance signature is a set of characteristic and repeatable behaviors (e.g., response time, resource utilization or throughput vs. some measure of load) that arise from specific performance tests. By systematically looking for signatures, scalability bottlenecks can be identified more readily than in the past. We summarize our experiences with performance signatures that have arisen out of multiple Internet application performance studies. The original objectives of those studies were to characterize the capacity and response times of different applications. However we found that we first needed to perform bottleneck and scalability analyses to identify latent problems and opportunities for major improvement. We provide a number of performance signatures that have been encountered in real Internet applications.

1. Introduction

Consider the following scenario: You're a performance analyst for a large telecommunications company. Its 1980, although it could just as easily be 10 years earlier or later. You've been asked to work the following problem. "At what call attempt rate will the switching module of a new switching system currently under development first violate the 98.5% < 3 seconds dial tone delay requirement?" Your approach? Develop a queueing model, of course. There's abundant documentation containing detailed information about the design. What's not covered can be learned by asking the developers. They know the approximate code size and can estimate task times. The answer isn't needed tomorrow – there's time to learn the system internals. The system will be in development for several years and production for

decades. “Of course, what we’d really like is some guidance in assessing the performance impact of some design choices we have.” Right up your alley. You can’t wait to get started.

Fast forward to 2001: You’re still a performance analyst for a large telecommunications company, albeit a considerably changed company. You’ve been asked to work the following problem. “At what login attempt rate will the new proposed security platform fronting our web assets first violate the 99% < 2 seconds response time requirement?” You’re approach? You try the 1980’s method, only to discover – the documentation, what there is, consists of high level slide shows or service descriptions which cover the “what” but little of the “how”; the developers can’t help much – they’re focused on integrating the 3rd party vendor products they’re using and are on an aggressive time schedule; the answer is needed yesterday; and the question was more like “what kind of performance can we expect” (translation “will it scale to support millions of users?”). You don’t know where to start.

Sound familiar? The preceding discussion illustrates a challenge facing today’s telecommunications performance analysts. The world has changed – in part due to deregulation in the industry, but even more so due the technology revolution we call the Internet. To be successful as analysts, we need to change as well. A different approach is needed.

In this paper we share some of our experience is using an empirical “gray box” load testing approach to performance analysis of today’s IP systems and applications. In particular, we introduce and illustrate the concept of a <u>performance signature</u> for assessing system performance and scalability. A performance signature is a set of *characteristic* and *repeatable* behaviors (e.g., patterns in resource measurements) that arise from specific performance tests. A signature may result from the behavior of a single performance metric or from the behavior of a combination of multiple performance metrics. Performance signatures combine elements of load testing [3] and performance tuning [2], each of which has a rich literature. Using signatures, we believe bottleneck detection and scalability questions can be more readily and rapidly answered with significantly less testing effort.

In Section 2 we outline the performance testing and signature process. We describe the types of tests that should be performed as well as the testing process. We also review some of the non-technical aspects of a successful performance testing effort. Section 3 gives many examples of performance signatures encountered in real applications. For each signature we present the problem definition, tests used to detect the problem, analysis, signature, where design changes are required, and the problem resolution.

2. Performance Testing Process

Performance testing of Internet applications is a delicate task that requires customized planning according to the properties of the system under study. However, there are general techniques and strategies that increase the likelihood of success in a signature-based performance testing effort. In this section we present discussions of key components of a successful performance testing strategy for Internet applications. We outline the process flow of a signature-based performance testing effort and discuss important factors that affect the outcome and results of a performance testing project.

2.1 Types of Performance Tests

Depending on the properties of the system under study, various types of performance testing can be implemented to identify performance signatures of the system. In this section we present a comparative discussion of methodologies and results of different types of performance testing.

Atomic Transactions vs. Operational Profiles: The transactions used to generate load to the system can either be in the form of atomic transactions or transaction mixes that reflect the operational profile. Atomic transactions are individual transactions selected on the basis of their frequency of occurrence, importance or resource consumption. Tests using atomic transactions are done in order to assess the impact of a particular type of transaction on the system, as well as to isolate performance issues to a particular transaction type. In operational profile performance testing, the performance of the system is evaluated using transaction mixes that to some degree resemble the operational profile of the production

environment. The purpose of testing with operational profiles is to get a realistic view of performance under field conditions.

Load Testing vs. Stress Testing: Load testing evaluates and models the performance of various components of the system under different controlled levels of load. The number of concurrent users may be fixed at a predefined number with each user generating load at a prescribed rate, or may vary according to the load, i.e., a new "user" may be spawned at each arrival epoch. In stress testing load is generated by a number of concurrent users. As soon as a user receives a response, it immediately submits the next request (with negligible client delay). Each of the simulated users does so independently and in parallel. Tests are run with increasing numbers of users until the system is saturated. Either load testing or stress testing can be done using atomic transactions or using an operational profile.

Sequence of Tests: Ideally, a systematic performance testing effort would integrate all possible combinations of tests in order to develop a comprehensive signature profile. In our experience, we have found it helpful to begin with atomic stress tests, since bottleneck isolation is usually easier in a tightly controlled environment. Then we move on to load testing with operational profiles in both stress mode and load test mode. Finally, we perform a soak run where load tests are performed under the expected operational profile and transaction volume for a long period of time (days) in order to discover any latent performance issues.

2.2 Signature-Based Performance Testing Process Flow

The following is a guideline for the process flow of a signature-based performance testing effort.

Test Planning Activities

- *Verify hardware, software and network configuration of the lab environment.* In order to achieve realistic results, the lab configuration should be identical to the field, although this is not always possible due to cost considerations. If the lab and field are not identical, there should be a clear view of how field performance will be estimated from the lab results. Differences between the lab and field

can usually be ignored for non-bottleneck resources. Extrapolation of performance for bottleneck resources can be more difficult. Extrapolation to additional processors (assuming multiprocessor scalability has been demonstrated), usually works. Extrapolation to different I/O subsystems has proven difficult.

- *Select tools that are appropriate for signature-based testing.* There are sophisticated vendor tools, that automate many of the testing tasks. We have effectively used LoadRunner® by Mercury Interactive and Silk Performer® by Segue. Both tools can be used for either load testing or stress testing. When cost is an issue, there are free tools that can be used or custom scripts can be developed. Typically, these have a less sophisticated user interface, are more cumbersome to use when the load test parameters need to be continually changed, and may be limited to either stress testing or constrained types of load testing. However, they are an excellent way to get started when time and/or money is an issue. Examples of free tools are http-load and MS WCAT. More information can be found at http://www.acme.com/software/http_load/ and http://msdn.microsoft.com/workshop/server/toolbox/wcat.asp. A very good and up-to-date listing of tools (free and commercial) can be found at http://www.softwareqatest.com/qatweb1.html.
- *Define testing objectives, performance criteria and metrics.* "Success criteria" for performance testing should be defined before testing begins. If requirements are not available, then testing should focus on scalability to identify the system breakpoints.
- *Review system architecture, data/task flow, and operational profile. Conduct qualitative performance/reliability assessment and modeling. Identify potential performance bottlenecks based on system architecture and operations.* Whatever information is available about the system internals and expected system operation should be understood before testing begins. Often, knowledge gained in this phase leads to a list of suspected bottlenecks. These can be used to design more effective test cases that isolate whether the suspected bottlenecks are present or not. Sometimes, bottlenecks can be identified from qualitative descriptions alone.

- *Identify critical transactions and define usage/demand forecasts for various transactions and operational profiles.* The idea in this phase is to identify "heavy hitter" transactions, either from a volume, resource consumption or importance perspective. The "heavy hitter" transactions form the atomic transactions. Operational profiles are critical in establishing credible transaction mixes. Generally, the transaction mixes will be mixtures of the atomic transactions. Operational profiles may also include different load and/or operational scenarios (e.g., a startup mode for the service with many customer registrations, a "steady state" mode that reflects a mature service, or a failure scenario). When bottlenecks are identified and need to be fixed at substantial time and cost to the product, there needs to be convincing evidence that the loads that were used are "realistic".
- *Review system log files, monitoring tools, and data collection/instrumentation capabilities. Identify required measurements that would capture all aspects of system performance and yield information on potential performance signatures.* Load test tools are frequently only able to capture user-perceived behavior. Bottleneck diagnosis needs to be based on an understanding of what is happening inside the system. Therefore key diagnostic data must be obtained from log files and system accounting utilities. Some of the system accounting utilities that we use heavily in Unix-based system performance testing are `sar`, `ps`, `iostat`, `mpstat` and `vmstat`. For NT systems we use utilities such as `perfmon`.
- *Define testable workload profiles (i.e., test cases/scenarios).* When designing test cases, it is important to "start simple". Performance testing efforts can be easily bogged down in complicated tests that shed no insight into the causes of anomalous system behavior. As noted in the previous section, we usually start with atomic transaction stress tests, followed by stress and load tests using one or more operational profiles, ending with long load tests at the expected system load and operational profile to uncover any latent problems. Tests should be conducted under different loads and operational profiles.
- *Develop test tool and instrumentation scripts. Include hooks in the scripts to capture potential signatures of interest.*

- *Conduct initial testing for proof of concept.* This critical step should not be overlooked. It is used to identify whether the lab is configured correctly, whether the load test scripts are coded correctly and whether the system instrumentation is functioning as expected. This step also identifies flaws in the basic testing approach. We have learned through painful experience not to neglect this step, when days of load tests results have been rendered useless due to a simple change that could have been made at the outset of testing.

Performance Test Execution and Analysis

- *Execute all load test scenarios and capture resource consumption, task flow, response time, traffic, and failures during all scenarios.* Testing should ideally be performed iteratively, executing and analyzing the results of a few tests, then modifying test scenarios if appropriate, before proceeding.
- *Produce load-service performance characterizations and models.* The test data should be used to produce load-service curves. In some cases, "gray" box modeling is useful for identifying possible bottlenecks [5] and can be helpful in the diagnosis process.
- *Perform a root cause analysis for all observed failures, anomalies.* Signature characterization is often time-consuming and requires consultation with developers and/or vendors to identify what is the cause of observed behaviors. Re-testing with different sets of test parameters may be required to pinpoint suspected problems.
- *Produce performance requirements, engineering rules, guidelines for system monitoring and future signature detection, and recommendations for bottleneck removal and performance improvement.*

2.3 Ensuring a Successful Performance Testing Effort

The guidelines in the previous section discuss various technical aspects of signature-based performance testing. However, there are numerous non-technical challenges in making a performance effort really happen. Often performance does not have the same priority as feature delivery (unless there has been a crisis), and performance is often the first thing to be cut when schedules inevitably get tight. Assuming

the service or product actually gets used, ignoring performance inevitably leads to disaster, but this is not in the minds of engineers urgently trying to deliver a product.

The following is a selected set of effective strategies that we have used in a variety of projects to achieve successful performance results.

1. *Obtain organizational alignment and organization of test resources:* Organizational and resource issues are most likely to delay or stall a performance testing effort. A systematic effort to verify objectives, identify expectations, obtain commitment of resources, and determine all processes that will be followed throughout the testing process is a critical prerequisite of a successful performance-testing project.

2. *Schedule time for performance testing:* Performance testing needs to be planned into the project schedule as part of the delivery process. Failure to meet this condition may result in having a new release in production with unidentified errors, failures, and bottlenecks.

3. *Form a dedicated team of performance testers that work across applications:* Due to the requirements of a release cycle, it is usually infeasible to expect system testers to conduct performance testing. Performance testing is a specialized expertise within testing and as such is best performed by people with experience in that area. This is particularly important in signature-based testing as it eliminates the need for repeated training and speeds up the overall testing process.

4. *Allow for iterative performance testing that continues beyond the end of the system test cycle:* Each performance testing cycle identifies performance signatures and issues that need to be verified or further investigated through additional testing of the current or future releases of the application.

5. *Proactively manage third party software vendors:* Considering that the dynamics and performance signatures of third party software are often unknown to the performance test team, it is important to

share results of “gray/black box” analysis techniques with the vendor to obtain more information about the internal software bottlenecks.

3. Case Studies

In this section we present performance signatures from multiple Internet applications. For each signature, we present the problem definition, the method used to detect the problem, analysis of the data, and a summary of the signature. For some of the signatures, we were able to suggest a design change to alleviate the bottleneck. Where this was possible, the resolution is also described.

Case Study 1: A Fatal Memory Leak

Problem Description: Memory leaks are a common software fault in applications written in languages such as C++ in which the programmer is responsible for memory allocation and recovery. Memory gets allocated, but due to a fault in the code, never gets freed. Big, obvious leaks are usually detected (and corrected) early in the software development process. Small, slow leaks sometimes slip through to load test, or worse, production. A memory leak is considered to be fatal if it results in the application crashing or otherwise failing to function.

Detection: Fatal memory leaks can be detected in load test by executing a long soak run (load test that is run for days) with a fixed workload while monitoring application throughput and memory consumption.

Analysis: The following figures provide example signatures. In Figure 1.1 we show throughput vs. time of a Web server application running on a Unix server we recently tested. The test started at time 0 and was scheduled to run over a weekend. Unfortunately, about 5½ hours into the test, things went awry. Throughput, which had been holding at a steady 62 transactions per second suddenly fell to effectively zero. What happened? In Figure 1.2 we show the memory profile for the Web server process during this test, captured by running a simple shell script executing `ps` and `sleep` in a loop. Initially 13MB, the process size grows rapidly at first, and then steadily until reaching 182MB at the 5½ hour point of the test.

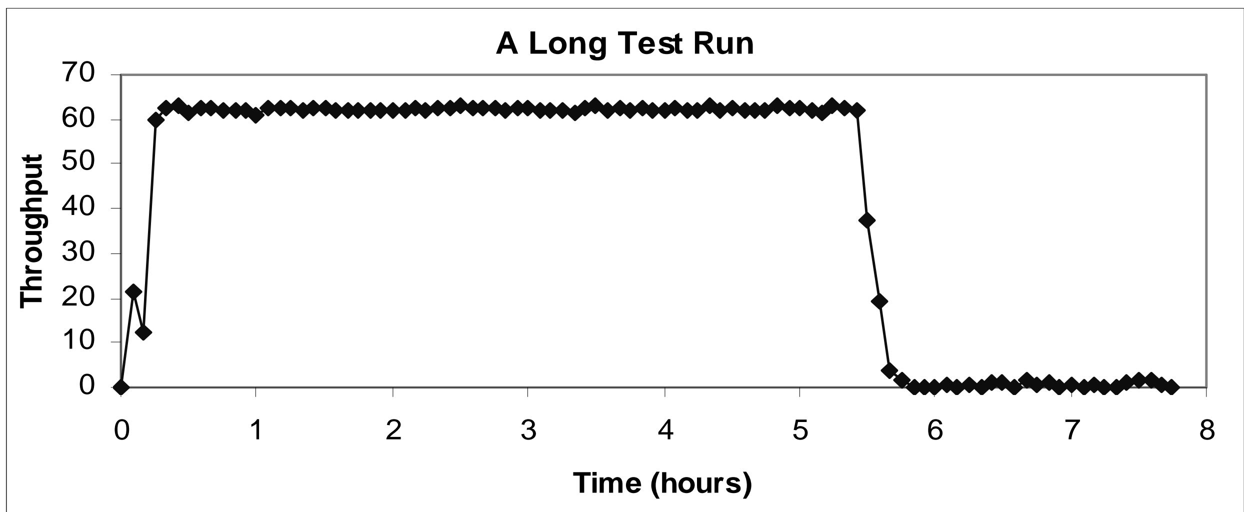

Figure 1.1 - Throughput vs. Time for Application with Fatal Memory Leak

At this point the process stopped growing, but also stopped serving all but a very small number of transactions. The other transactions timed-out and were discarded by the load generator.

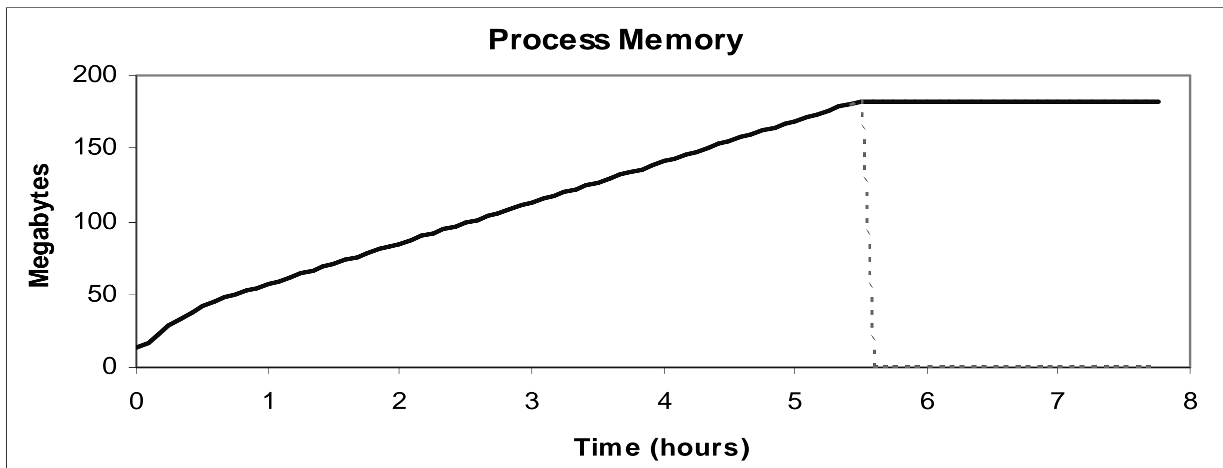

Figure 1.2 – Memory Usage vs. Time for Application with Fatal Memory Leak

The test was re-run several times to make sure the results were reproducible. In each case the profile was repeated. In most cases the process died, as indicated by the dotted line in Figure 1.2, as opposed to remaining active, but otherwise non-functioning as in the first test run (Figure 1.1). This normal growth, which reaches a certain level and stops, needs to be distinguished from a fatal memory leak, in which memory grows to the point that something breaks and the application fails to function. The latter may not reveal themselves except during a long test run as in this example.

Signature: Figures 1.1 and 1.2 together constitute the signature of a fatal memory leak – steady throughput during a long test run accompanied by steady application memory growth followed by a sharp drop in throughput accompanied by no further memory growth, or more typically, an application crash.

Case Study 2: Throughput Degradation Due to Poor Multiprocessor Scalability

Problem Description: Web applications are commonly deployed on symmetric multi-processor (SMP) servers. An N-way server is typically cheaper than N 1-way servers, and definitely easier (and less costly) to administer and operate. A measure of how well an application performs on an SMP server is how close it comes to delivering N times the throughput performance of a single CPU server. Some deviation from strictly linear scaling is common and inevitable, but major deviations can negate the cost advantages cited above. Flat scalability (i.e., add a CPU and throughput stays the same) is clearly not good. Negative scalability (i.e., add a CPU and throughput goes down!) is downright bad and to be avoided.

Detection: Application SMP scalability can be assessed by conducting a series of load tests with different numbers of active CPUs. Each test is run in a series of stages, starting with low load and increasing load at each stage. For Web applications, this is typically accomplished by starting the load test with a small number of "virtual user" load generation agents and relatively long think times between transactions and adding virtual users and/or reducing think times at each stage. Measures of interest are transaction throughput and CPU utilization.

Analysis: Figure 2.1 shows CPU utilization vs. throughput for a multithreaded Web server application running on a 4-CPU Unix server. The first few points are as expected – CPU increases linearly with load as the number of virtual users is increased. Above 140 transactions/second (25% CPU utilization), however, the curve changes showing a more rapid increase in CPU utilization as throughput increases.

Above 200 transactions per second (50% CPU utilization), the slope becomes quite steep. Above 75% CPU utilization, an interesting phenomenon is observed – throughput decreases as more virtual users are added! (and as CPU utilization increases). The maximum sustainable throughput (throughput achieved

with a large number of users and corresponding to 100% CPU utilization) is actually only 180 transactions per second.

What's going on? In Figure 2.2 we show the results of re-running the test with fewer active CPUs. (The `psradm` command available with the SUN Solaris operating system is a simple way to control the

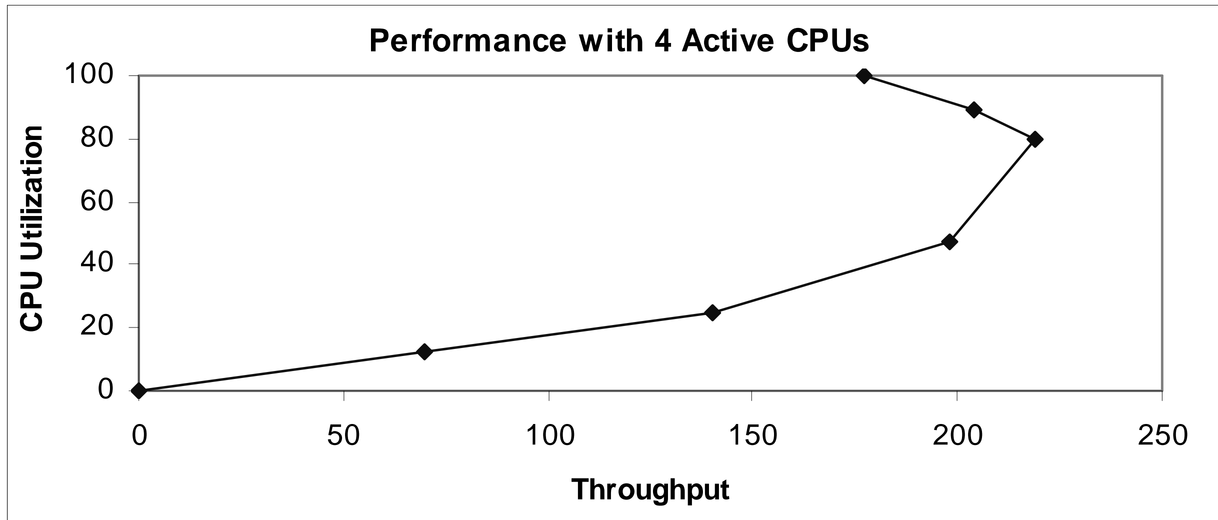

Figure 2.1 – CPU Utilization vs. throughput for Application with Poor SMP Scalability

number of active CPUs). With 3 active CPUs, we see a bit of the same behavior at high CPU utilization, but a higher sustainable throughput than observed with 4 active CPUs. Two active CPUs is better still. Not only is the sustainable throughput higher, the curve no longer turns back on itself. Finally, going to 1

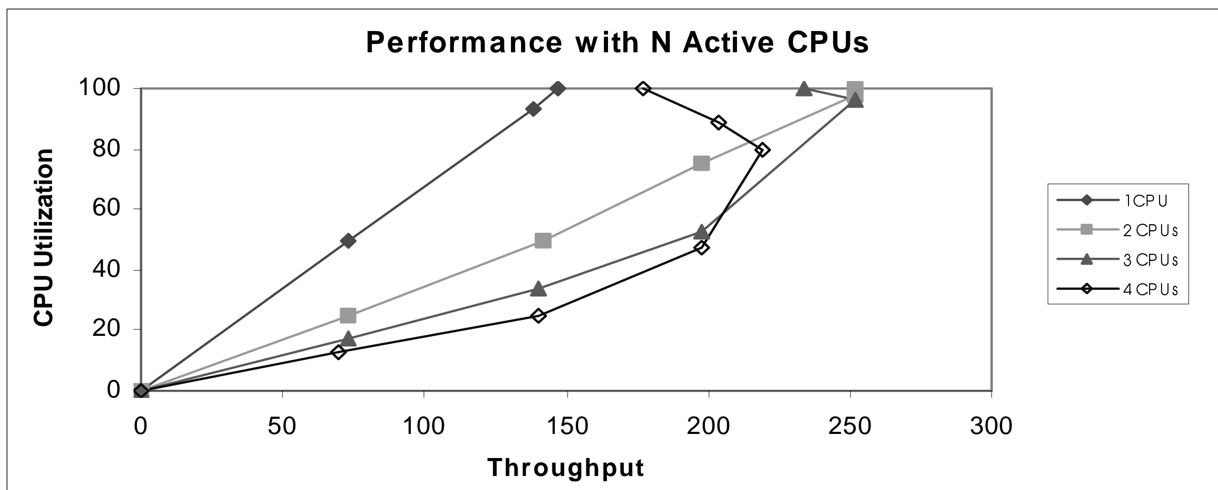

Figure 2.2 – CPU Utilization vs. throughput for Application with Poor SMP Scalability

active CPU produces the behavior we had originally expected – a linear curve with lower sustainable throughput than that observed with 2 active CPUs.

In Figure 2.3 we show the key results – sustainable throughput vs number of active CPUs – from our tests. Scaling from 1 to 2 CPUs increases throughput by a factor of 1.7 – less than ideal, but acceptable. Adding a 3rd and 4th CPU leads to lower throughput – we're better off without them!

Signature: Figures 2.2 and 2.3 constitute the signature of poor SMP scalability – one or more CPU utilization vs. throughput curves that show rapid increase in CPU utilization with little increase, or worse, a decrease in throughput when all or most CPUs are active, accompanied by a sustainable throughput vs. active CPU profile that increases little or actually decreases as the number of CPUs increase.

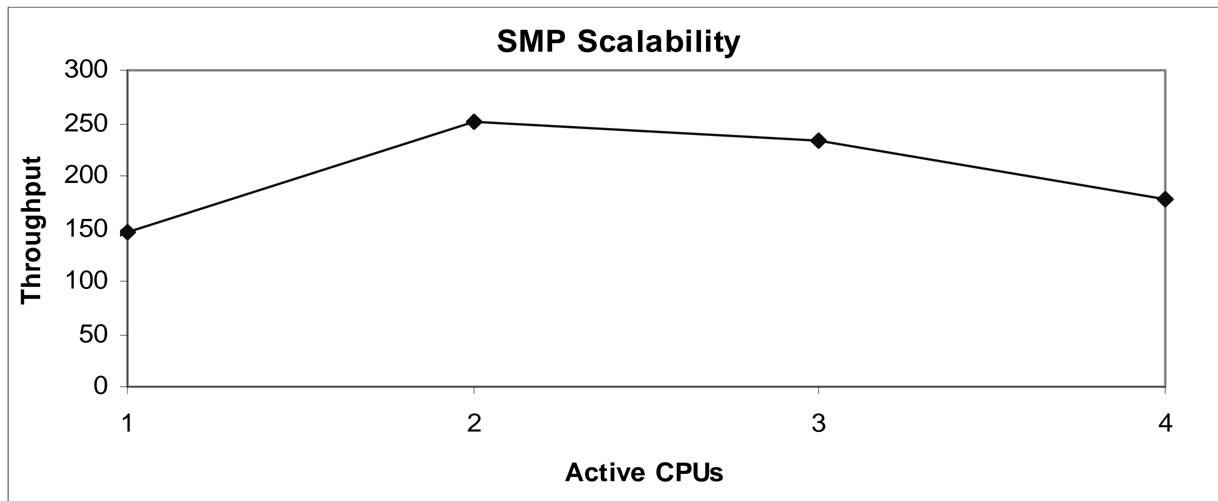

Figure 2.3 – Sustainable Throughput vs Active CPUs for Application with Poor SMP Scalability

Case Study 3: Misconfiguration on a multi-processor machine

Problem Description: With multi-processor machines common now, overlooking an error in configuration can quickly result in performance degradation. In this case study, a database on a multi-processor machine was configured to utilize only a subset of the processors available. This error, not evident during normal operations, was straightforward to detect during a stress test.

Detection: By running a stress test, failure to utilize all CPU's in a multi-processor machine can be detected by measuring CPU utilization.

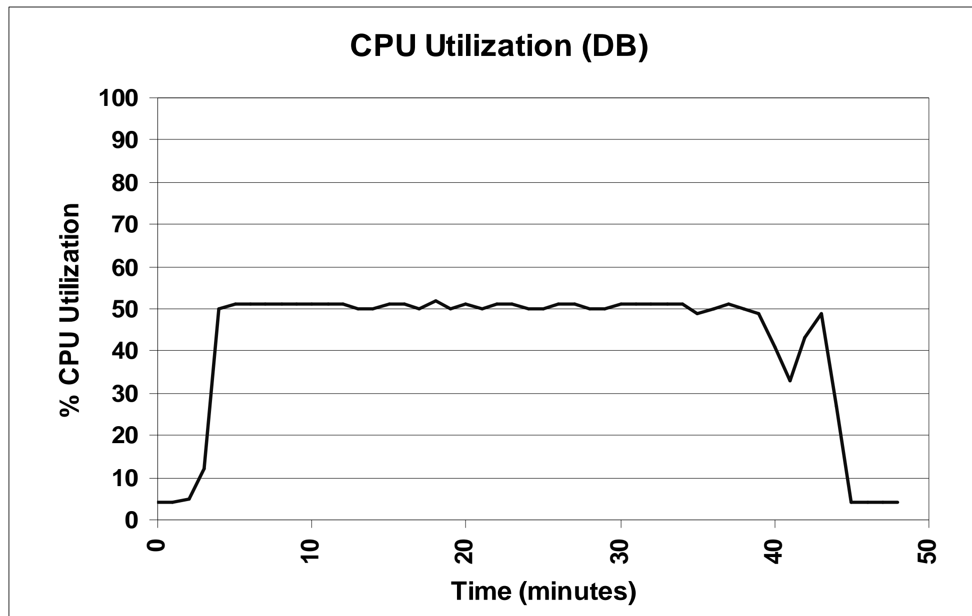

Figure 3.1 - CPU Utilization on Database Machine

Analysis: We performed a stress test on a machine running a work task manager (WTM) that accessed a database (DB) on a separate machine. The CPU utilization of the second machine (with the database) is shown in Figure 3.1. Immediately, we see a distinct feature – constant CPU utilization of around 50% for the duration of the test. This feature suggests that, on this 4-CPU machine, 2 of the CPU's were not being utilized. A quick check of the status of the CPU's confirms that all of the CPU's were online (in fact, commands like *sar*, *iostat*, etc. measure CPU as % of **available** CPU's, so a 50% CPU utilization would **not** suggest CPU's being off-line). Therefore, we concluded (correctly) that the database was not configured to utilize all 4 CPU's. Correcting this configuration problem resulted in over 50% increase in throughput. Note that there are other possibilities for this observation, but all are related to the lack of thread/processes. For example, the WTM front server may be running with 2 threads (either 2 single-threaded processes, or one process with 2 threads) so that the database would only be able to utilize 2 of the CPU's.

Signature: Figure 3.1 constitutes the signature of a misconfiguration on a multi-processor machine: a constant CPU utilization at a fractional level of the number of CPU's (e.g., 25%, 50% or 75% on a 4-CPU machine). The solution is configuring the servers to utilize all available CPU's.

Case Study 4: Concurrency Penalty Due to a Synchronized Linear Search

Problem Description: Java-based applications are extremely popular due to their ease-of-programming and faster development times. However, many of the implementation details are hidden from the developer. One of these is code synchronization. If a number of simultaneous users access synchronized code, then requests must queue for the synchronized resources creating a software bottleneck and preventing the CPU from being fully utilized. If the code execution time increases with the number of simultaneous users (e.g., a linear search through a table holding information on active users), then performance degrades as concurrency increases.

Detection: Software bottlenecks can be uncovered by conducting a series of stress tests with an increasing number of concurrent users. The measures of interest are CPU utilization, throughput, and average processing time per transaction.

Analysis: Figure 4.1 shows CPU utilization vs. simultaneous users for a Java application running on a Unix server. CPU utilization flattens at only 60% at a concurrency level of 25 simultaneous users, and there are no memory or I/O problems. This indicates that the bottleneck is internal to the software. Figure 4.2 shows throughput vs. simultaneous users. The throughput peaks at the same concurrency level where CPU utilization flattens, then degrades, suggesting that the CPU time per transaction is increasing as a function of the concurrency level. We computed the CPU consumption per transaction and this is indeed the case. What's going on? The application under study requires searches over active users as well as creation of new objects for each transaction. Object creation, known to be serialized in Java, contributes to the software bottleneck but is not the sole source, since object creation time should not increase

proportional to the number of active users. We suspect that the searches over active users may have been performed linearly, leading the CPU time to be proportional to the number of active users.

Signature: Figures 4.1 and 4.2 constitute the signature of a software serialization bottleneck together with a linear search – CPU utilization levels off at a value less than 100%, throughput peaks then steadily degrades, and processing time per transaction increases with the number of simultaneous users.

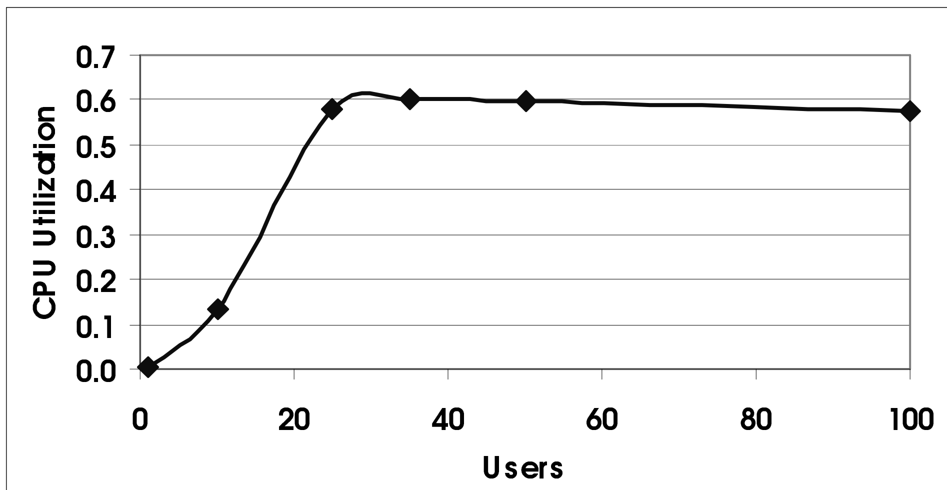

Figure 4.1 – CPU Utilization vs. Number of Simultaneous Users

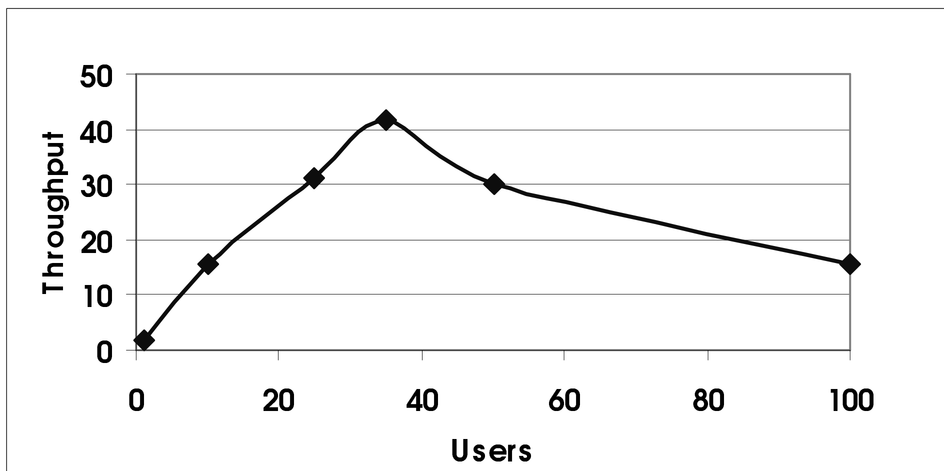

Figure 4.2 – Throughput vs. Number of Simultaneous Users

Case Study 5: Single-Threaded Disk I/O Bottleneck

Problem Description: This example illustrates one of the (many) signatures associated with a software scalability bottleneck resulting from code serialization in a database application. One of the servers serializes read/write requests to the disk sub-system, creating an I/O-related choke-point that throttles file system activity and causing significant values for *wio*[1] (more than a few %). The system in this example consists of the integration of commercial application servers: a Web server (WS), a Java virtual machine servlet engine (SE), and an LDAP database (DB) server, accessed via a Web browser.

Detection: Database code serialization bottlenecks can be uncovered by conducting a series of stress tests with an increasing number of concurrent users. As soon as a user receives a response, it immediately submits the next request (with negligible client delay). Each of the N simulated users does so independently and in parallel. Hardware (CPU, memory, disk, I/O) resource consumptions are measured via standard Unix tools (`mpstat`, `vmstat`, `iostat`). In addition, end-to-end client response time is measured.

Analysis: Figure 5.1 plots the client response time on the left-hand axis, and the *active* (*usr+sys*) and *total* (*usr+sys+wio*) CPU utilizations on the right-hand axis, as a function of the offered load (throughput). The *active* CPU consumption measures CPU utilization explicitly attributable to executing code associated with request processing, while the *total* CPU consumption includes a component (*wio*) that reflects how well the software architecture matches the hardware resources. A significant value for *wio* (more than a few %) indicates that block I/O is a significant bottleneck in the system architecture.

---

[1] CPU cycles are attributed to *wio* **only** when **every** request-handling thread is blocked waiting for the return from a disk operation. *%wio* is the percent of time that the CPU spends in the wio state.

The CPU utilization curves in Figure 5.1 demonstrate that the system has a serious I/O bottleneck. The fact that the *%wio* is so high suggests that disk resources and/or I/O activities are severely limiting throughput. There are several possible causes of a disk I/O bottleneck that leads to high %*wio*, including:

- A physical limitation in the **disk** or **memory** sub-system capacities, leading to large disk queues, long service times, high paging rates, and poor process/memory management;
- An architectural limitation in the application **software** (Java servlets or LDAP server) or **database** design, leading to serialization of read/write requests to the disk or excessive contention (locking);
- A limitation in the OS **software** (`fsflush`), coupled with a limitation in the database or application software, leading to serialization of requests or long commit times.

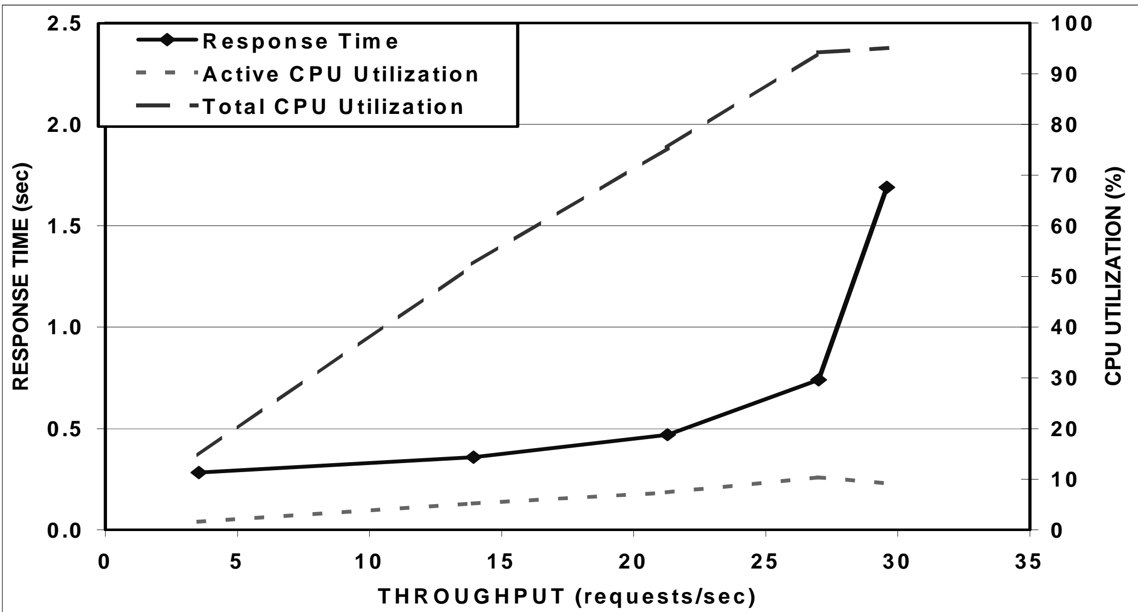

Figure 5.1 – Stress Test Results

Analysis of the raw data (particularly from `iostat`) ruled out a physical limitation in the disk and memory sub-systems. Disk queues are empty, service times are not excessive, swap and free pages are high, paging rates are normal, and process queues are small. Thus, the culprit is likely to be in the (database) software.

Several telling clues arise from the `iostat` data (a sample portion of the output is shown in Figure 5.2). Each row represents a 5-second average. Thus, 2.6 ops/second (row 1) equals 13 I/Os during that interval. As can be seen, disk operations are extremely "batchy". There are long periods (average ~19 seconds) with no disk activity whatsoever, followed by large batches of I/Os (average ~9 operations).

ops /sec	KB /sec	**# in queue**	**# in service**	service time	**%wait time**	%busy time	%usr CPU	%sys CPU	**%wio CPU**	%idl CPU
2.6	19.4	**0**	**0.1**	36.0	**0**	2	6	4	**87**	3
0.2	0.4	**0**	**0**	7.4	**0**	0	6	3	**90**	2
0	0	**0**	**0**	0	**0**	0	6	4	**87**	3
0	0	**0**	**0**	0	**0**	0	6	3	**89**	2
0	0	**0**	**0**	0	**0**	0	5	3	**90**	2
0	0	**0**	**0**	0	**0**	0	5	3	**89**	3
0.2	0.2	**0**	**0**	12.3	**0**	0	5	4	**91**	0
2.8	21.2	**0**	**0.1**	51.1	**0**	2	6	3	**90**	1
0	0	**0**	**0**	0	**0**	0	5	3	**91**	0
0	0	**0**	**0**	0	**0**	0	4	3	**92**	1
0	0	**0**	**0**	0	**0**	0	5	2	**92**	1
0	0	**0**	**0**	0	**0**	0	5	3	**91**	1
0	0	**0**	**0**	0	**0**	0	5	4	**92**	0
0.2	0.4	**0**	**0**	11.8	**0**	0	4	3	**77**	16
2.4	19.2	**0**	**0.1**	49.5	**0**	2	6	4	**90**	1
0	0	**0**	**0**	0	**0**	0	4	5	**72**	19

Figure 5.2 – Sample portion of `iostat` *output*

Note that the disk queue length is always 0, and the number of operations in service rarely exceeds 0.1, suggesting that there is no more than 1 operation submitted to the disk at any time (i.e., I/Os are serialized by the application software). For instance, look at row 1: there are 13 operations over 5 seconds, averaging 36 ms/op. Therefore, the disk is active 13*36 ~ 470ms during the interval, and the occupancy is 0.47/5 ~ 0.1 (matches the output). If operations are serialized, as suspected, then the average number in queue would be 0 (matches the data). If operations were submitted in a batch, then the average number in queue would be roughly given by [12*36 +11*36 + ··· +1*36]/5000 = ~ 0.6 (does not match the data).

Some of this behavior can be explained by other causes, but not *all* of the behavior. For example, the OS flushes disk blocks every 30 seconds, explaining the "batchiness" observed in the disk operations. However, this does not explain the serialization of disk operations, since the batch is flushed at once,

leading to a non-zero disk queue. Thus, the raw data supports the conclusion that the **LDAP server** is the throughput-limiting bottleneck in the architecture, serializing read/write requests to the disk sub-system.

Signature: The performance signature of a database code serialization bottleneck consists of:

- Excessive CPU time attributed to the *wio* state, with relatively low values for *usr* and *sys* CPU time;
- Reasonable levels for disk service times, swap and free pages, paging rates, and process queues; "Batchy" I/O operations, disk queue levels of 0, and number of I/Os in service well below 1.

Case Study 6: Scalability Bottlenecks Due to Semaphore Contention

Problem Description: Web applications often access information that is stored in backend databases. For example customers and customer care agents may access a database that displays customer care content information. Database bottlenecks can be caused when a single process locks the entire database. The bottleneck becomes evident as the multiprocessing engine spends almost all of its CPU cycles in spin locks and context switches. In addition, response time becomes sensitive to minor increases in the database size, as a steep increase in response time is observed due to relatively minor increases in lock holding time.

Detection: Database scalability bottlenecks can be identified by conducting a series of single- and multi-transaction tests under different load levels as a function of database size, the number of requests/sec, and the caching mechanism. Measures of interest are response time, CPU utilization, top processes consuming CPU utilization, number of context switches per second, and number of semaphore spins per second.

Analysis: Figure 6.1 shows the response time as a function of load for different database sizes and a transaction mix representative of the field. For the database sizes of 200 MB and 400 MB, the response times increase dramatically at about 1.2 hits/seconds, while for the 600 MB database the increase starts at 1 hit/second and the average response time decreases beyond 1.4 hits/second. A closer look reveals that the decrease in response time is caused by a decrease in offered load to the database due to request

timeouts. Upon further examination of the system resource data at the 1.2 hits/second rate, we found the following:

- SUN SE toolkit reports a severe performance alarm on semaphore spins, issued when more than 500 spins per second are observed or more than 20% of the CPU is consumed by semaphore spins. In our specific example semaphore spins of about 70,000 spins/second were observed (a severe alarm).
- The number of context switches is very close to the number of spins/second.
- The only process consuming significant processor CPU time is the process managing database access.
- CPU utilization is evenly distributed across all six CPUs, with the system running at over 80% CPU utilization.
- Disk Utilization is around 10%.

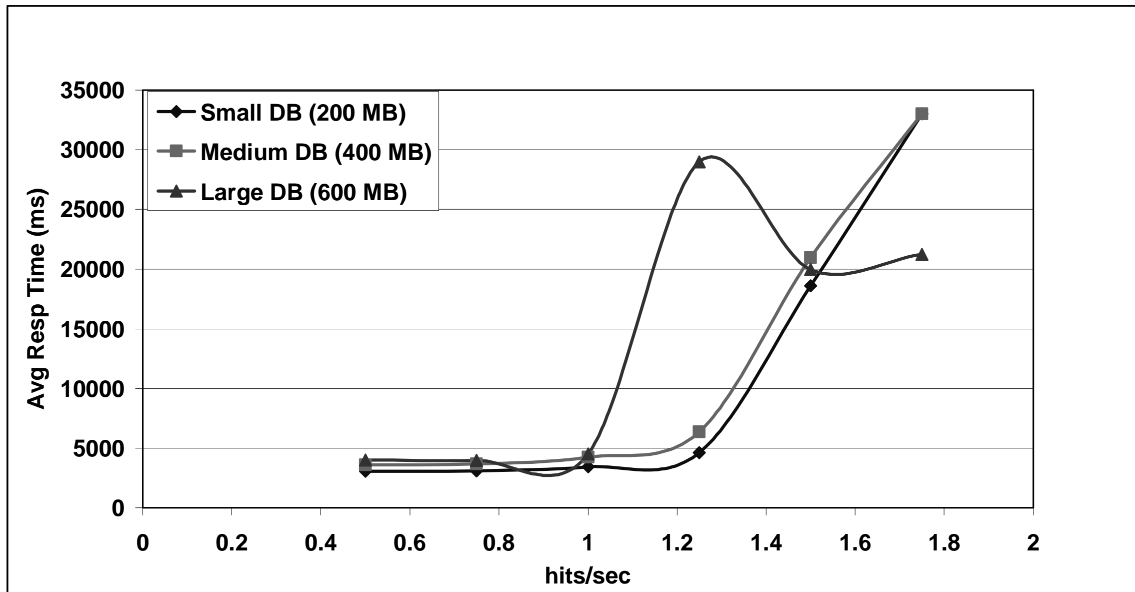

Figure 6.1 – Offered load vs. Response Time for Different Database Sizes

What's going on? As the load grows transactions consume CPU (due to semaphore spins) while waiting for service from the database. The CPU cost per transaction increases as the load increases, reflecting the increased semaphore contention, as shown in Figure 6.2. The decrease beyond 1.4 hits/sec is due to large numbers of timeouts that cause the actual processed load to be much smaller than the offered load. As the

database size increases the database locking time is slightly longer, which causes the instability to be observed at lower loads as the database grows.

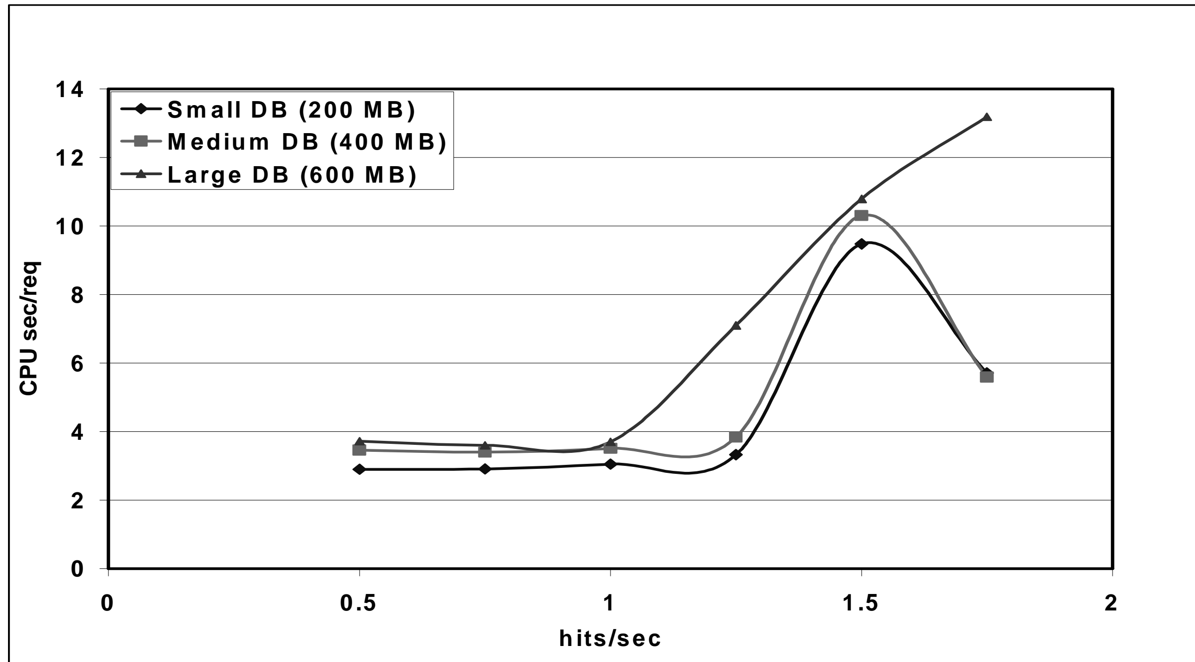

Figure 6.2 – Offered Load (hits/sec) vs. CPU tine per transaction (secs)

In summary, the database bottleneck causes a serialization of all database access. Requests to the database that find the database semaphore locked, queue in busy wait causing the system to waste almost all of its CPU cycles in context switching and spin locks. The CPU utilization approaches 100% as offered traffic increases and the Disk/IO sub-system remains underutilized. The database controller process is the only process in the system reporting consumption of CPU cycles.

Signature: The signature of a database bottleneck caused by single-threaded locking is indicated by a high semaphore spin rate, number of context switches very close to the number of spins/second, and most of the CPU being consumed by the process managing database access, together with the Disk/IO

subsystem being underutilized. A supporting indication is that the CPU per transaction increases with the transaction rate.

Problem Resolution: Three simple remedies were recommended to alleviate the database locking bottleneck: increase the use of caching so that fewer database accesses are required, configure multiple database access processes, avoid database locks all together on read-only transactions.

Case Study 7: Database I/O Bottleneck Due to Poor Use of Caching

Problem Description: Internet applications are frequently used to access backend databases, and may experience database I/O bottlenecks. Symptoms include high disk utilization resulting in processes spending inordinate amounts of time waiting for I/O. The common wisdom is to add more powerful disk subsystems. However, sometimes the root cause of the problem lies in the database access patterns. In these cases, the problem is to distinguish between when there is a real need for expensive hardware upgrades, or when the problem can be fixed with smarter DB query routing. Specifically, exploiting file-caching mechanisms more efficiently may relieve the bottleneck, then fewer disk reads are required and the overall performance and capacity improve.

Detection: This type of bottleneck can be detected during routine load tests with realistic operational profiles that mirror the access patterns in the field and hardware that matches the deployed configuration. The symptoms can be detected by monitoring disk utilization and memory usage (e.g., using `iostat`, `vmstat` and `ps`).

Analysis: Table 7.1 shows the memory measurements (output of `vmstat`) from load tests done with multiple DB server machines and queries generated according to a realistic profile. The queries were routed to the machines using a round-robin policy. In addition to the `vmstat` measurements, the disk subsystem utilization was at 97-100% and the DB server processes were experiencing a wait for I/O of

memory		page							cpu		
swap	free	re	mf	pi	po	fr	de	sr	us	sy	id
1514264	57792	5	554	4398	0	4532	0	873	23	9	68
1514632	52984	8	510	8004	1	8003	0	1574	27	9	63
1514504	51264	2	573	9308	2	9327	0	1831	40	10	50
1514200	52400	3	509	8465	2	8435	0	1659	28	9	63
1513784	54096	4	531	7250	1	7436	0	1438	39	9	52
1514096	55104	10	545	6386	3	6508	0	1276	44	10	46
1514160	55192	1	593	6498	3	6379	0	1268	39	13	49
1514000	53840	2	545	7438	3	7673	0	1479	21	8	71
1513904	53208	5	560	8262	2	8122	0	1581	43	11	46
1513896	54408	4	527	6952	2	7064	0	1398	28	9	64
1513928	52680	1	551	8412	2	8451	0	1672	39	10	51
1513848	50136	8	520	10545	2	10502	0	2041	21	9	70
1513656	50272	4	558	10182	2	10217	0	2029	18	8	74
1513696	55184	6	521	6059	3	6352	0	1294	31	14	55
1513616	57664	2	506	4695	3	4546	0	916	54	9	37
1513568	47800	2	539	12398	2	12414	0	2426	29	9	62
1513320	52800	3	542	8603	2	8669	0	1674	43	13	43
1513368	49560	3	527	10791	3	10868	0	2166	24	10	66

Table 7.1 – Sample output of **`vmstat`**

~40%. One indicator of a memory shortage seen in Table 7.1 is the *page scanner rate*[2] (column *sr*). Ideally this rate should be below 200 scans/second. We see that the page scanner is about 1000-2000 scans/sec. Also the page in rate (column *pi*) is essentially equal to the free rate (column *fr*), indicating that every time the operating system brings data into memory it needs to free up an equivalent amount of memory. The memory measurements led us to conjecture that a memory bottleneck was actually showing up as a disk bottleneck. Surprisingly, the sum of the memory used by all of the processes (found using `ps`) was 1 GB, out of a total of 2 GB of available RAM, which amounts to just 50% memory utilization. The performance is further explained by the following key factors:

- The DB was based on the Unix file-system and therefore implicitly used the Unix file caching mechanism[3]. The usage of memory by the file cache is not captured by standard Unix memory measurement commands. This explains why the memory usage in `ps` added up to only 50%.

---

[2] The page scanner rate [2] is the number of pages per second that the paging daemon scans while it looks for pages to steal from processes that are not using them often.

[3] Unix *file cache* is a main-memory based cache where Unix stores files recently read in from disks. The files remain in the cache even after they are closed and the process exits. The next time a process asks this file to be opened and read, Unix tries to get it first from the file cache, and if not found there, retrieves it from the disk.

- The DB was 50GB, compared with a maximum of 1GB available to the Unix file cache. Queries were distributed randomly across the 50GB on each of the servers (round-robin routing), resulting in a cache hit ratio of 1/50 (2%). The remaining 98% of the time Unix would fetch new files from the disk and replace files currently in the file cache. This explains the high scan rate (file cache is constantly refreshed due to lack of locality) and `the page in rate = free rate` (all of the available memory was used up by the Unix file cache, which had to be freed before pages could be brought in).

Signature: The performance signature for a database I/O bottleneck due to poor use of caching is: high disk utilization, high percent of process time spent in wait for I/O, high page scanner rate, page in rate=free rate, low memory utilization by processes.

Problem Resolution: The performance bottleneck was resolved by finding ways to improve the cache hit ratio. This was done by (1) increasing RAM size (2) reducing the range of the data serviced by each DB server machine. In our case, the Unix files constituting the DB were organized by a category, which was a natural choice by which queries could be routed to servers, thus exploiting cache locality more efficiently. E.g., a 4 GB RAM machine serving a 5GB DB would potentially result in a 3/5 ≅ 60 % cache hit ratio.

4. Conclusion

The Internet development paradigm and rapid time-to-market have created a fundamental change in the nature of application performance modeling and bottleneck identification. As a result, performance analysis has shifted from traditional detailed analytic and simulation approaches to an empirical approach. This paper introduced the notion of performance signatures based on data obtained from targeted performance testing and outlined a process for implementing signature-based performance testing We provide numerous examples of commonly occurring performance signatures that have been found in real IP applications.

5. Acknowledgements

The authors would like to thank William Leighton for his encouragement and suggestions that we write this paper, as well as for many valuable comments.

6. References

[1] Bulka, D., *Java Performance and Scalability: Volume 1,* Addison-Wesley, New York, 2000.

[2] Cockroft, A., Petit, R., *Sun Performance and Tuning – Java and the Internet,* Sun Microsystems Press, California*,* Second Edition, 1998.

[3] Taylor, S., *"A Developer's Crystal Ball,"* ADT January 1997 – Software Engineering, http://www.spgnet.com/ADT/jan97.

[4] *"The Volano Report: Which Java platform is fastest, most scalable?"* http://www.javaworld.com/javaworld/jw-03-1999/jw-03-volanomark.html).

[5] Reeser, P., *"Using Stress Tests Results to Drive Performance Modeling: A Case Study in 'Gray-Box' Vendor Analysis,"* submitted to Proceedings, SIGMETRICS/PERFORMANCE 2001 (published by ACM c2001).