

Diagnosing Performance and Stability Problems with Load Testing

Alfred J. Barchi

ajb@ajbinc.net
<http://www.ajbinc.net/>

Table of Contents

Introduction.....	3
Prerequisites.....	3
References for Interpreting Perfmon Counters	4
The Flowcharts.....	5
Diagnosing Application Saturation.....	9
What Is Saturation?.....	9
Identifying the Bottleneck Device and Predicting X_{\max}	11
Diagnosing Thrashing in Applications	15
Context Switches/Sec.....	15
An Example of a Possible Software Bottleneck	16
An Example of Thrashing Caused by a Memory Leak.....	18
IIS and SQL Server on the Same Machine	20
Diagnosing Software Aging.....	21
Diagnosing Garbage Collection Issues	22
The Garbage Collection Spike	22
What is Happening.....	26
Diagnosing Excessive Use of the ‘finalize()’ Method.....	27
Evaluating Stability Errors.....	29
Diagnosing Load-Induced Errors.....	30
Footnotes to the Flowcharts	32

Introduction

This paper is intended to document an orderly approach to diagnosing performance and stability problems in applications through the use of using load testing. It includes a set of diagnostic flowcharts, similar to those used in medicine and industry. The rationale behind each of the flowcharts is discussed in subsequent sections. Each flowchart includes a reference to its corresponding section.

Diagnosing performance problems is typically 3 parts pattern recognition and 1 part intuition based on experience and understanding of how applications are typically implemented. These flowcharts should therefore be used with the understanding that new and unique situations will be encountered from time to time that aren't covered by these flowcharts.

All of the flowcharts should be used when evaluating the results of a load test.

These flowcharts are intended for diagnosing applications hosted on MS Windows-based systems only.

Prerequisites

In order to use the flowcharts, it is necessary to collect the data.

Throughput data is obtained from the load test tool for load tests, and from web logs for operational systems, assuming that the bytes sent, bytes received and time-taken data elements logging elements have been enabled.

For Windows-based applications, the following Perfmon counters should be collected at a minimum:

- Processor / % CPU Time
- Memory / Available Mbytes
- Physical Disk / % Disk Idle
- Processor / Interrupts/sec
- System / System Calls/sec
- System / Context Switches/sec
- System / Threads
- System / Processor Queue Length
- Network Interface / Bytes Total/sec

'Network Interface / Bytes Total/sec' must be added manually for each server because the name of the interface card is part of the counter name, and can differ from machine to machine.

Additionally, 'Web Service / Total Method Requests' should be added where diagnosing load-balancing problems is important (this is a cumulative counter). The results for all IIS servers for this counter should be nearly identical in a properly load-balanced application.

For .NET applications, the following counters should be added for diagnosing garbage collection issues:

- .NET CLR Memory / # Gen 0 Collections
- .NET CLR Memory / # Gen 1 Collections
- .NET CLR Memory / # Gen 2 Collections
- .NET CLR Memory / % Time in GC

Additionally, if it is suspected that the application is making explicit calls to initiate garbage collections – normally a bad practice – then the '.NET CLR Memory / # Induced GC' counter can be added (this isn't covered in the flowcharts, but if this counter is anything other than zero, then this is an application design issue).

For ASP.NET applications, the following counter can be added to the IIS Server(s) to help diagnose saturation:

- ASP.NET / Requests Queued

This counter is not discussed in the flowcharts, but if it is consistently greater than zero, it indicates that the application is nearing saturation.

References for Interpreting Perfmon Counters

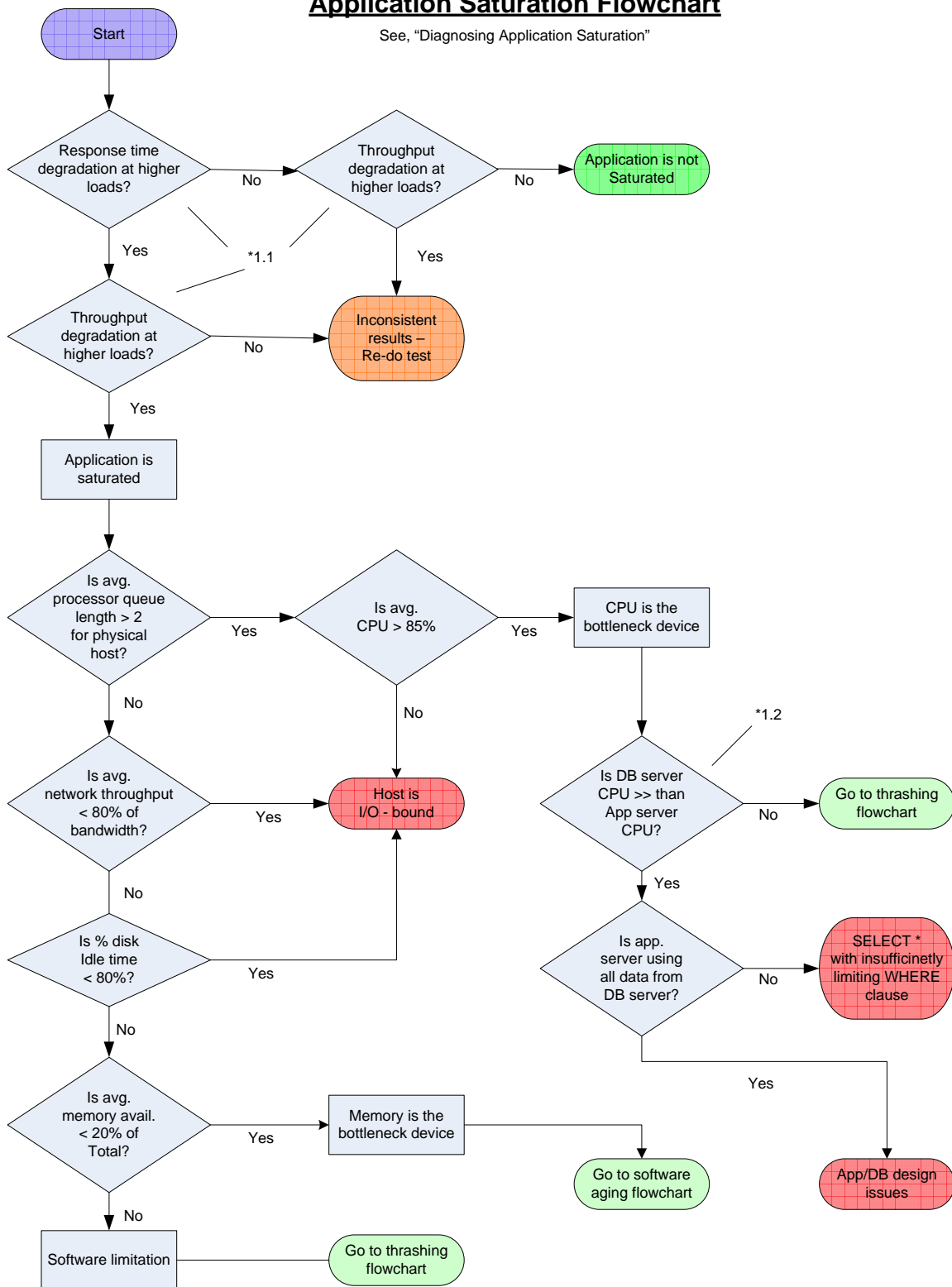
Discussion of Perfmon counter interpretation can be found in:

- "Improving .NET Performance and Scalability.pdf," Meier, Vasireddy, Babbar and Mackman, Microsoft 2004
- <http://msdn2.microsoft.com/en-gb/library/ms998583.aspx>
- <http://www.microsoft.com/technet/prodtechnol/Windows2000Pro/reskit/part6/proch27.msp>

The Flowcharts

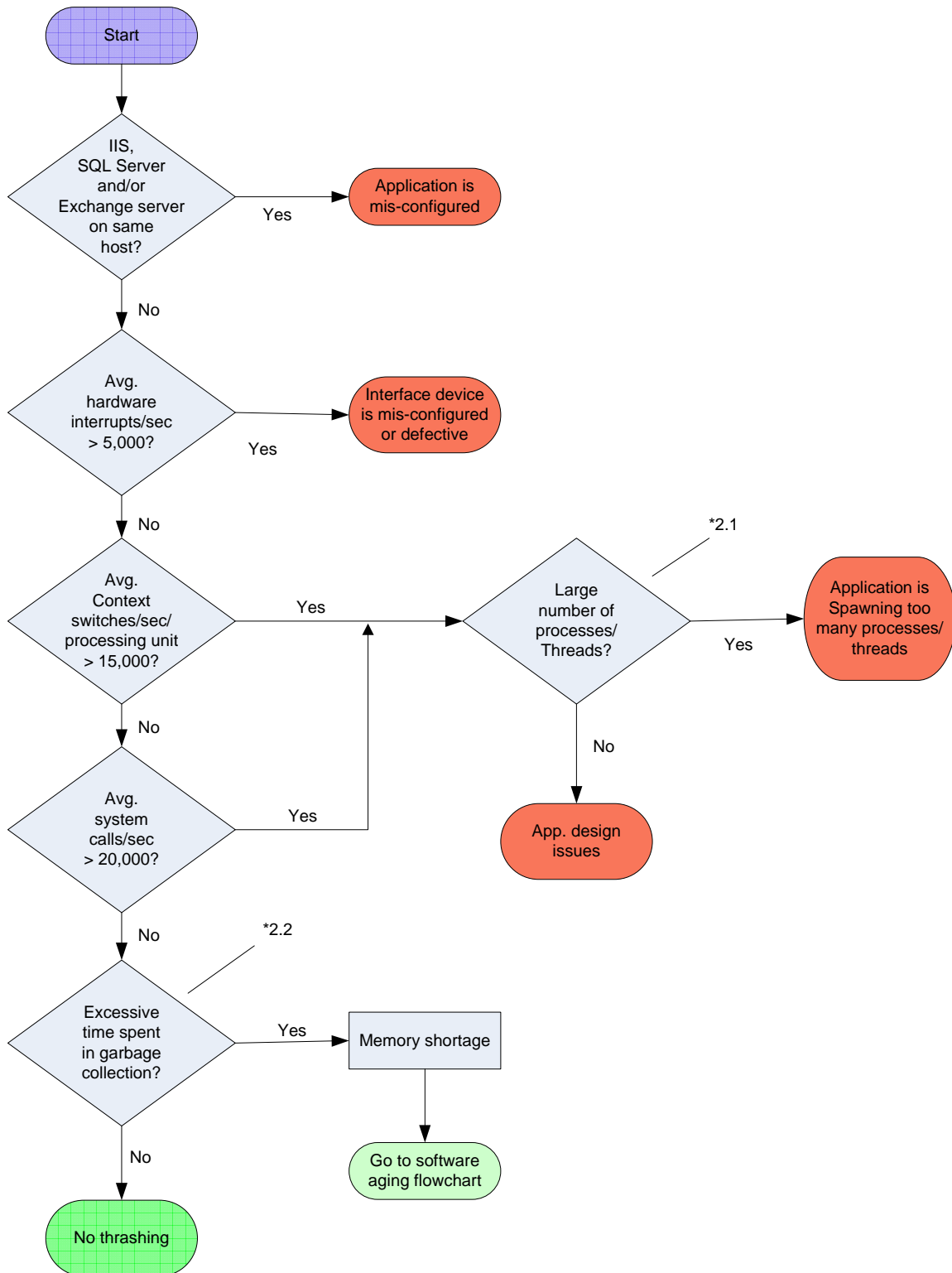
Application Saturation Flowchart

See, "Diagnosing Application Saturation"



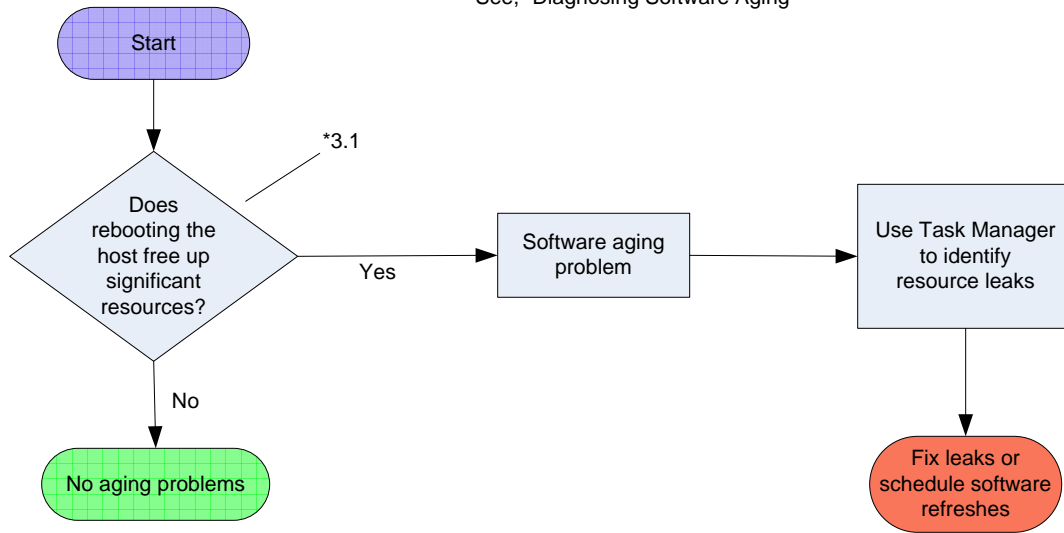
Thrashing Flowchart

See, "Diagnosing Thrashing in Applications"



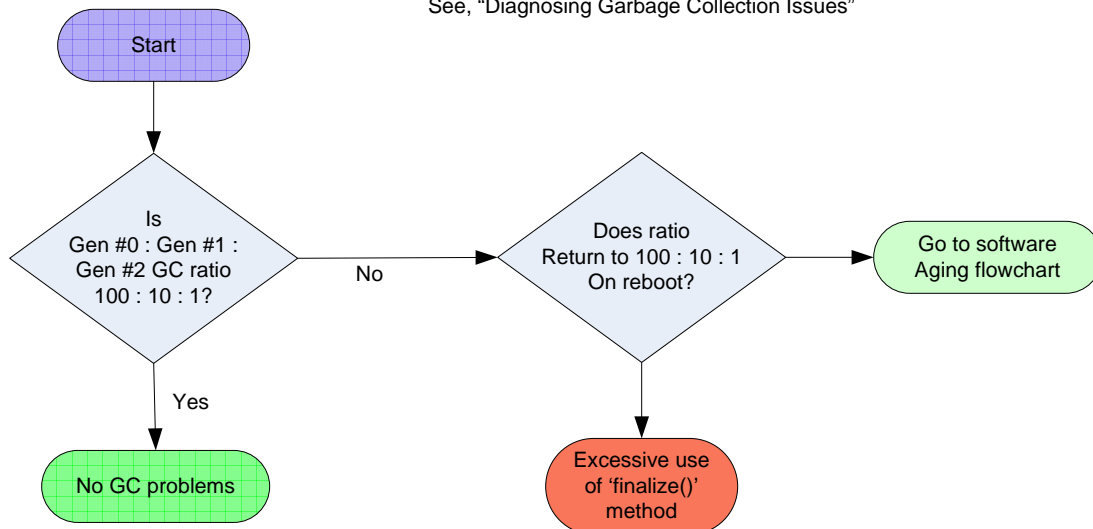
Software Aging Flowchart

See, "Diagnosing Software Aging"



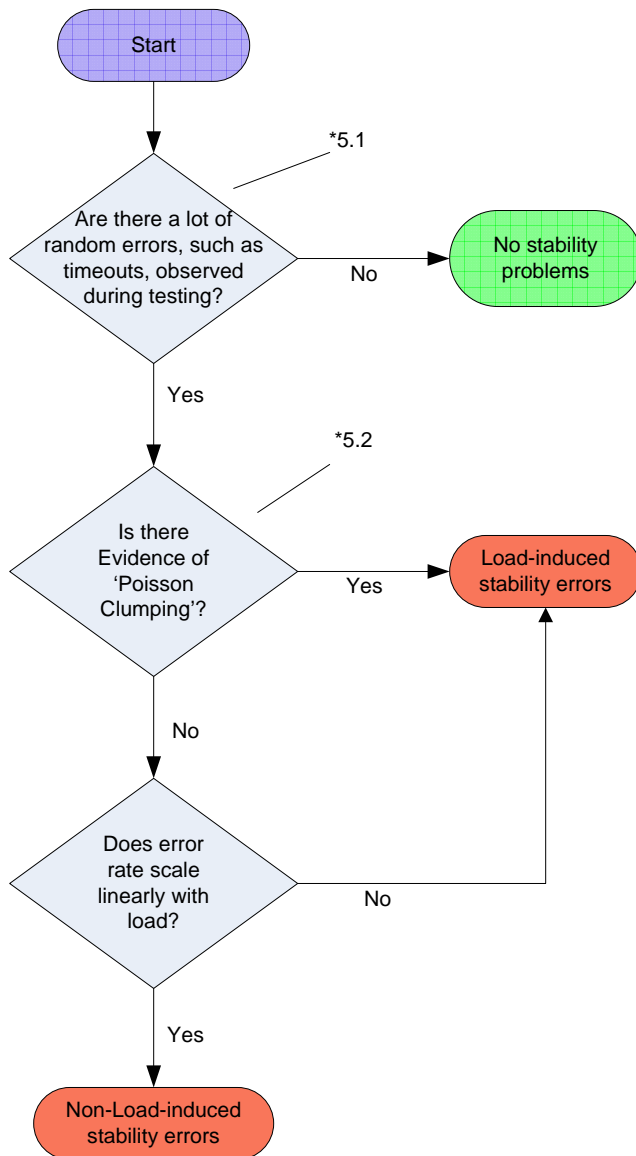
Garbage Collection Flowchart

See, "Diagnosing Garbage Collection Issues"



Stability Errors Flowchart

See, "Evaluating Stability Errors"



Diagnosing Application Saturation

The purpose of load testing is to assess the behavior of an application under load. Specifically, one objective is to identify the capacity of the application and any bottlenecks that limit performance, as well as to determine whether or not the application meets its service level requirements for a particular type of load.

This section discusses how to assess maximum capacity and identify bottleneck devices using load testing and Perfmon counters.

What Is Saturation?

Basically, as load increases, throughput increases, until maximum resource utilization on the bottleneck device is reached. At this point, maximum possible throughput is reached. Saturation occurs once maximum throughput is reached, and at this point, queuing occurs. Queuing typically manifests itself by degradation in response times.

This phenomenon is described by Little's Law:

$$Q = X * R$$

Where Q is equal to the number of transactions/bytes/bits in the system, X is the throughput of the system, and R is the response time. As Q increases, X increases (R also increases slightly, because there is always some level of contention at the component level). At some point, X reaches X_{\max} , the maximum throughput of the system. At this point, as Q continues to increase, the response time R increases in proportion. This is illustrated in the following graphs (taken from "The Art of Computer Systems Performance Analysis", by Raj Jain, 1991):

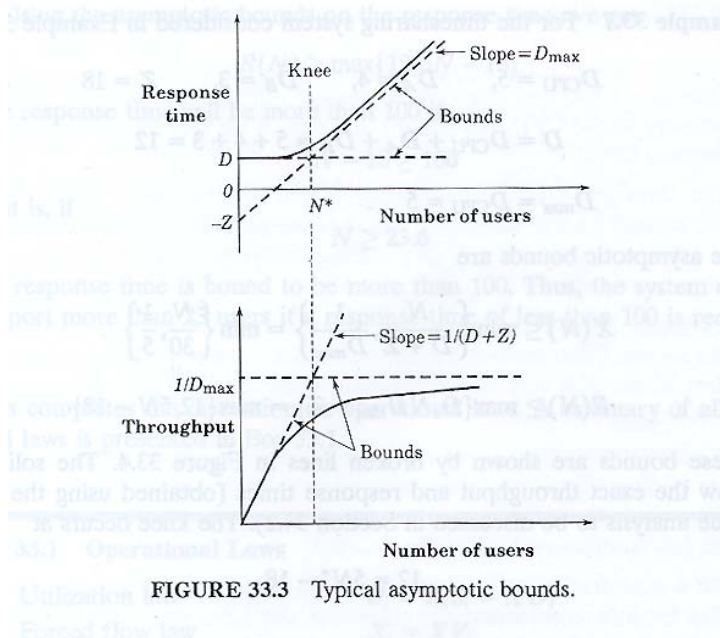
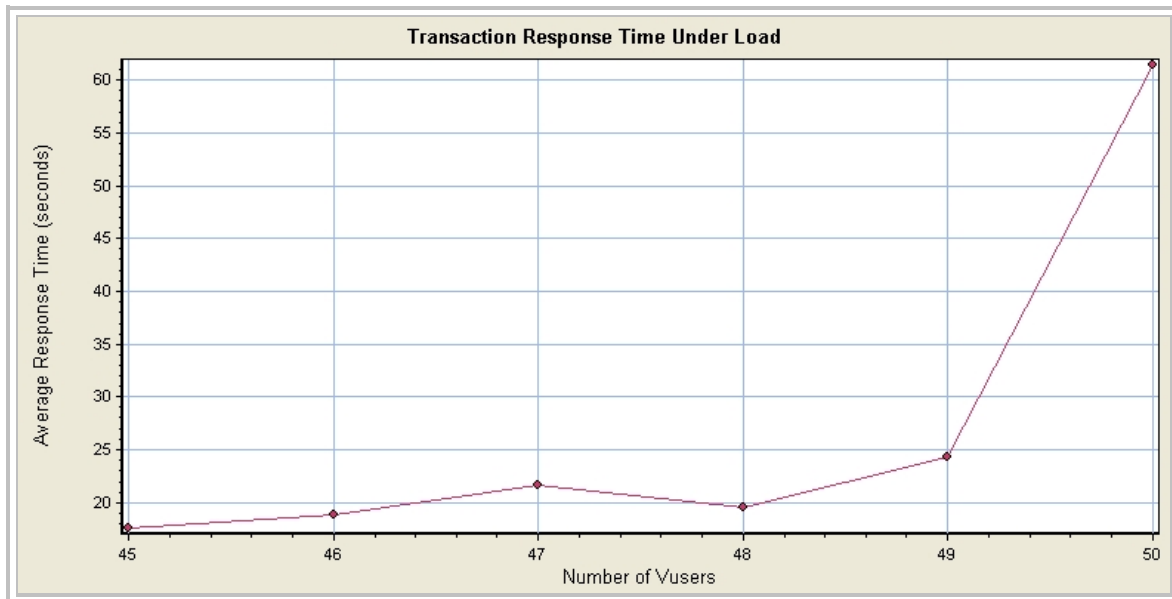


FIGURE 33.3 Typical asymptotic bounds.

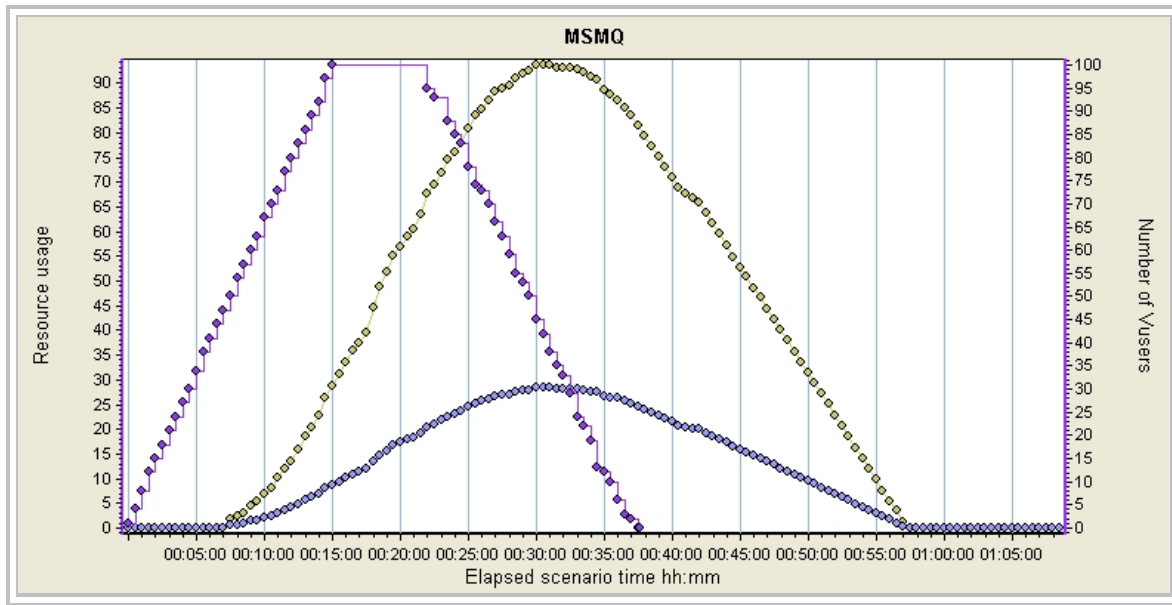
In this diagram, N^* is the number of users (an expression of load) at which the application begins to go into saturation. It is also sometimes referred to as the point of optimal throughput. D is the service demand, which will be discussed below, D_{\max} is the maximum service demand of all of the devices in the system, again discussed below, and Z is user 'think time' (not important to this discussion).

In LoadRunner, for example, we typically see saturation as a degradation in response times with increased load, defined as the number of simultaneous Vusers. For example:



This chart was taken from a recent load test. It is important to always generate this chart in LoadRunner in order to diagnose whether or not the application is saturating.

Queuing doesn't always manifest itself as a degradation in response times, however. In the same application from which the above chart was taken, a different configuration was tested using a message queuing component (Microsoft MSMQ) to de-couple the user presentation layer from the back-end processing by queuing user transaction requests and responding to the user immediately. The back-end database would then pull transaction requests off of the queue and process them as quickly as it could. The chart below illustrates this:



The purple line shows the number of Vusers. The yellow line shows the number of transactions in the MSMQ queue awaiting execution. The gray line shows another processing component of MSMQ. It is interesting to note that queuing occurs at the same level of load in both configurations, i.e., 47 simultaneous Vusers. At this point, the application is saturated.

Identifying the Bottleneck Device and Predicting X_{\max}

Another important concept is the Utilization Law:

$$U_i = X * D_i$$

Where U_i is the percentage of utilization of a device in the application, X is the application throughput, and D_i is the service demand of the application device. The maximum throughput of an application X_{\max} is limited by the maximum service demand of all of the devices in the application.

Typical application devices include: CPU, memory, disk and network interfaces. We get application throughput from LoadRunner. We get device utilization from Perfmon counters. For example, in a load test where LoadRunner reports 200 kb/sec average throughput:

$$\text{CPU}_{\text{avg}} = 80\%$$

$$\text{Memory}_{\text{avg}} = 30\%$$

$$\text{Disk}_{\text{avg}} = 8\%$$

$$\text{Network I/O}_{\text{avg}} = 40\%$$

We use the average values rather than the maximum for two reasons. First, if we used maximum values, our results would be based on outliers. By using average values, our

calculations are based on a large number of samples over time. Second, LoadRunner, for example, gives us actual average throughput, but only give 'graph' maximum throughput. The averages should be filtered from the point where the load is first applied until the point where the load is removed.

In this example, the service demands are:

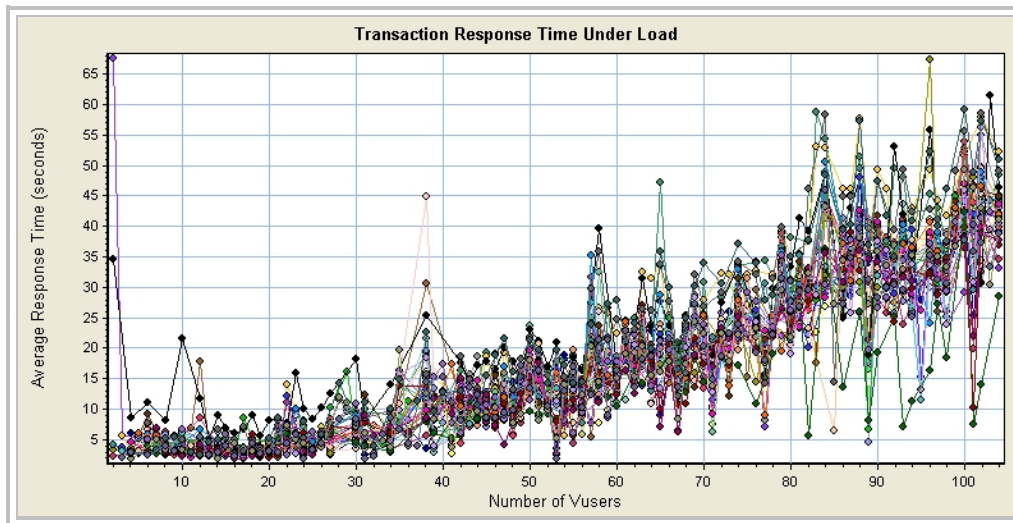
$$\begin{aligned} D_{\text{cpu}} &= 0.8 / 200 \text{ kb/sec} &= 0.004 \text{ sec/kb} \\ D_{\text{memory}} &= 0.3 / 200 \text{ kb/sec} &= 0.0015 \text{ sec/kb} \\ D_{\text{disk}} &= 0.08 / 200 \text{ kb/sec} &= 0.0004 \text{ sec/kb} \\ D_{\text{network I/O}} &= 0.4 / 200 \text{ kb/sec} &= 0.002 \text{ sec/kb} \end{aligned}$$

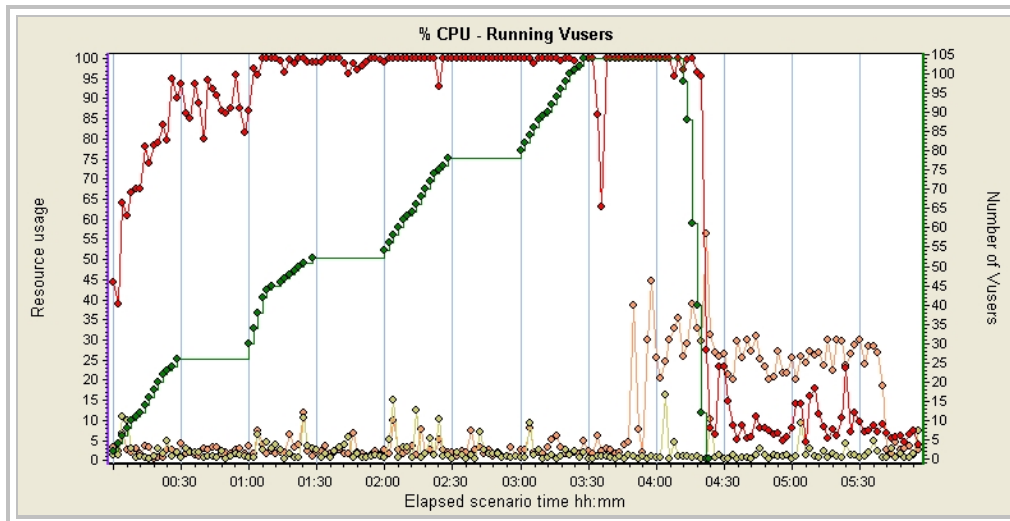
In this case, D_{max} corresponds to the CPU. So, the CPU is the bottleneck device. We can use this to predict the maximum throughput of the application by setting the CPU utilization to 100% and dividing by D_{cpu} . In other words, for this example:

$$X_{\text{max}} = 1 / D_{\text{cpu}} = 250 \text{ kb/sec}$$

In order to increase the capacity of this application, it would first be necessary to increase CPU capacity. Increasing memory, network capacity or disk capacity would have little or no effect on performance until after CPU capacity has been increased sufficiently.

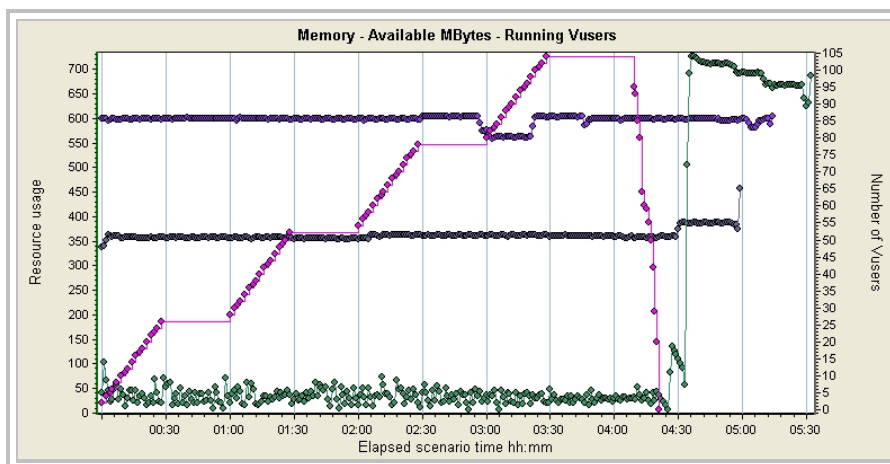
In the following application, these charts were obtained:



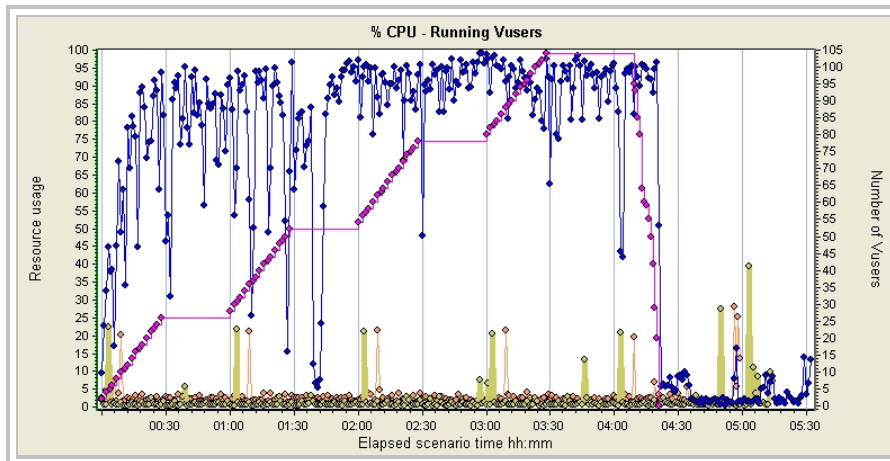






The first chart shows the application going into saturation at a load of about 30 Vusers. The second chart shows CPU utilization. The red line is the CPU utilization of the application server and the green line is the number of Vusers. This chart shows the CPU utilization on the application server going to 100% at a load of 30 Vusers. As an additional check, we can look at the processor queue length. Microsoft indicates that a processor queue length consistently greater than 2 indicates a bottleneck (the acceptable queue length is somewhat higher for virtual machines, such as those hosted under VMWare). If we see that the processor queue length is consistently high, we can confirm that the application is in saturation and insufficient CPU is the cause.

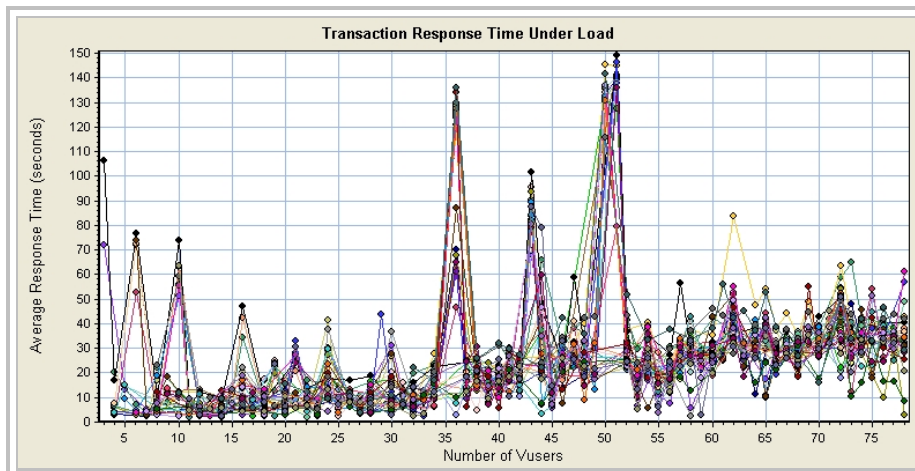
In this next example, the following LoadRunner charts were obtained:



Color	Scale	Measurement	Min.	Ave.	Max.	SD
■	1	Memory (Available MBytes):Server1	336	361.55	463	9.241
■	1	Memory (Available MBytes): Server2	534	596.883	659	10.465
■	1	Memory (Available MBytes): Server3	2	143.297	732	244.255
■	1	Run	N/A	N/A	N/A	N/A



Color	Scale	Measurement	Min.	Ave.	Max.	SD
	1	Processor(_Total) (% Processor Time): Server1	0.0	2.592	100	8.636
	1	Processor(_Total) (% Processor Time): Server2	0.0	1.545	100	7.865
	1	Processor(_Total) (% Processor Time): Server3	0.0	66.161	100	42.288
	1	Run	N/A	N/A	N/A	N/A



In this particular example, the available memory on the application server was extremely low, reaching a minimum of 2 MB available out of a total of 1 GB. The CPU utilization on the application server was also extremely high. Once again, we can see that the application is starting to go into saturation at about 30 Vusers.

In this case, the memory is the bottleneck device, followed closely by the CPU. When memory was increased from 1 GB to 6 GB, the CPU became the bottleneck device (see the previous example). As a well-known expert in performance analysis observed, “You can never really eliminate a bottleneck device, you can only shuffle the deck.”

Diagnosing Thrashing in Applications

Occasionally, we see applications where that have poor performance due to mis-configuration of the hardware, or problems in the design or configuration of the software that result in 'thrashing'. This section provides a description of how to go about diagnosing these issues.

Context Switches/Sec.

The primary indicator of thrashing is generally the Perfmon counter, "Context Switches/Sec." It is important to understand this counter is an aggregate for an entire host. For example, a DL-580 has 4 dual-core CPUs, but there is only one of these counters for the entire machine.

Microsoft's guidance for Windows Server 2003 states that more than 15,000 context switches/sec. per processing unit indicates a problem. So, for the example of the DL-580, there are 8 processing units, and the maximum acceptable number of context switches/sec. would be 120,000.

Microsoft guidance also states that less than 5,000 context switches/sec. per processing unit is "not worth worrying about." So, less than 40,000 context switches/sec. for the DL-580 would be perfectly normal.

Generally, system administrators set their monitoring alarm levels at around 14,000 context switches/sec.

So, what does a high rate of context switching mean? Well, it can mean any number of things. For example, if accompanied by a high rate of interrupts, it typically indicates a problem with an interface adaptor, such as being configured in PIO mode when it should be configured in a DMA mode.

No specific rate for interrupts is usually given. Instead, most analysts look for sudden changes in the rate. However, this type of analysis is typically used in an operational monitoring situation.

For our purposes, we would like to be able to evaluate the interrupt rate in the context of a load test. One Intel article:

<http://www.intel.com/cd/ids/developer/asmo-na/eng/76915.htm?page=7>

suggests that more than 5,000 interrupts/sec. is high. A quick check of old load test reports did not show any cases where this occurred.

If the number of system calls/sec. is high, this typically indicates a software bottleneck. The same Intel article suggests that more than 20,000 system calls/sec. indicates a problem. When this situation occurs, we need to look at the number of processes, the number of threads and the amount of memory available, particularly when we are dealing with a .NET application.

A high number of processes or threads accompanied by a high rate of context switches and a high rate of system calls indicates that there are too many processes

or too many threads active simultaneously, and the application is 'thrashing' itself, i.e., the application has too much parallelism and it is spending an excessive amount of time switching from one thread to the next.

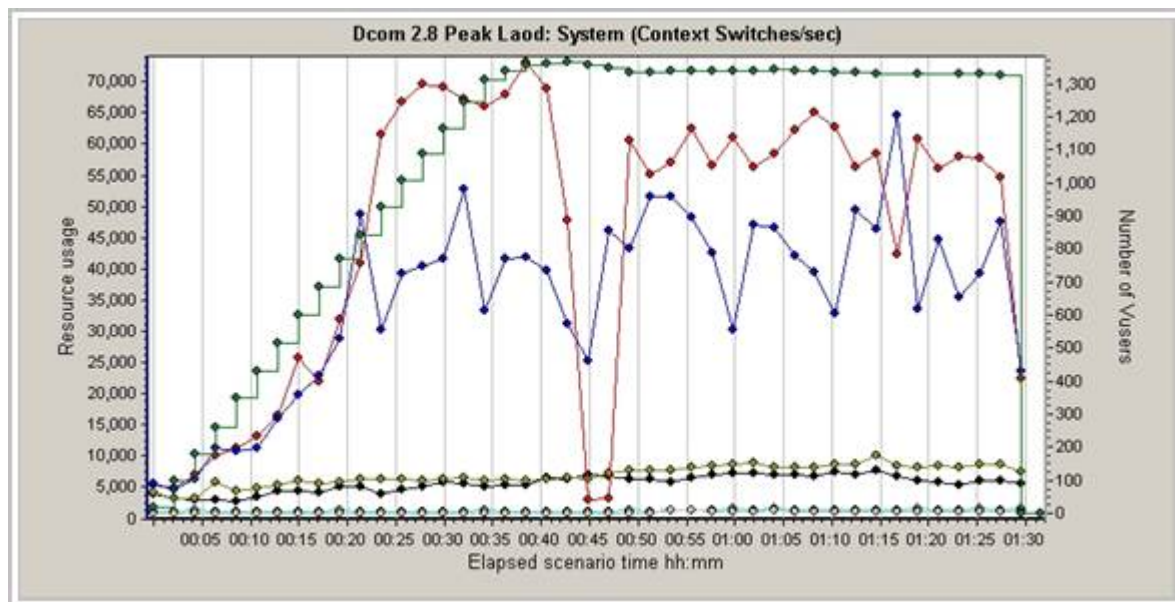
In .NET applications, when we see a high rate of context switches accompanied by a high rate of system calls and low available memory, we next want to look at the number of Gen #0, Gen #1 and Gen #2 garbage collections. Microsoft guidance states that Gen #1 garbage collections should occur at about 10% of the rate of Gen #0 garbage collections, and Gen #2 garbage collections should occur at about 10% of the rate of Gen #1 garbage collections.




If there are an excessive number of Gen #1 and Gen #2 garbage collections relative to Gen #0 garbage collections, this suggests that the application may be thrashing itself due to insufficient memory causing an excessive amount of high generation garbage collections in order to scrounge up enough memory to satisfy the next operation.

The cause of this kind of situation may be due to insufficient physical memory, or due to a software aging problem, i.e., a memory leak. If we re-boot the machine and the problem goes away, this indicates a memory leak. If the problem doesn't go away, then it's time to add more memory.

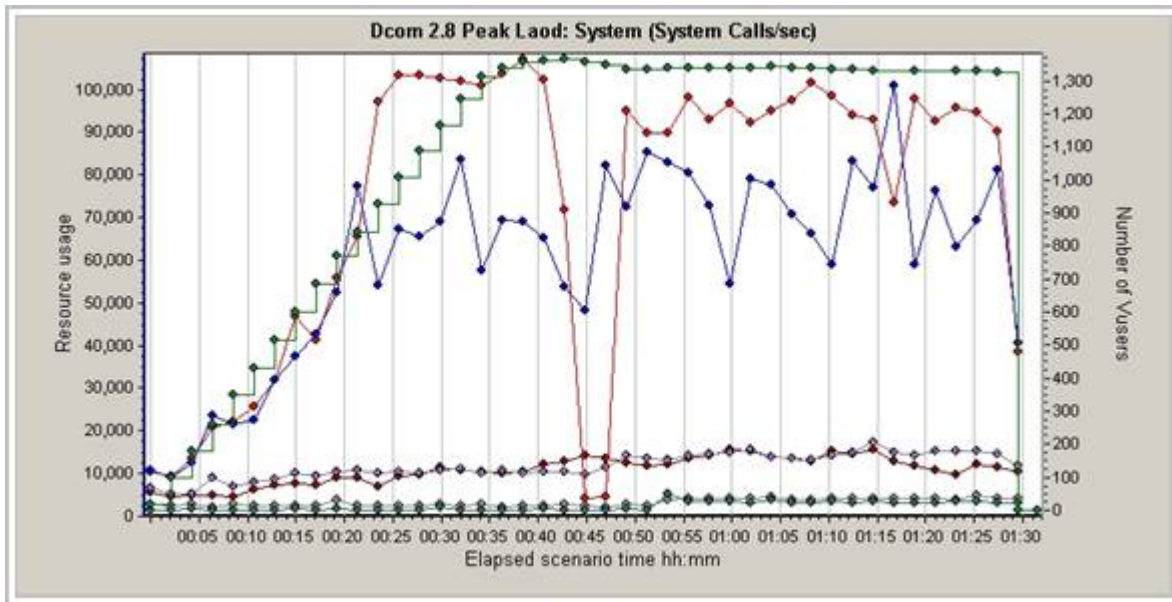
An Example of a Possible Software Bottleneck

The following graphs were taken from a load test that was conducted over a year ago:



Color	Measurement	Graph's Min.	Graph's Ave.	Graph's Max.	Graph's Median	Graph's SD
	System (Context Switches/sec):Server1	2779.679	5598.889	7632.631	5870.708	1337.386
	System (Context Switches/sec): Server2	3267.77	6938.823	10109.07	6623.861	1559.394
	System (Context Switches/sec): WebServer1	2970.588	45995.765	73156.652	56599.767	22770.71

■	System (Context Switches/sec): WebServer2	4714.073	35106.737	64624.73	39466.593	14382.049
■	System (Context Switches/sec): Server3	1070.579	1253.792	1541.155	1204.144	144.88
■	System (Context Switches/sec): Server1	835.483	988.731	1409.286	872.839	161.686
■	Run	0.0	1007.857	1366	1329	467.343



Color	Measurement	Graph's Min.	Graph's Ave.	Graph's Max.	Graph's Median	Graph's SD
■	System (System Calls/sec): Server1	4387.29	10704.997	15673.177	11356.861	3206.966
■	System (System Calls/sec): Server2	4978.951	11667.426	17446.188	11060.506	2941.709
■	System (System Calls/sec): WebServer1	4026.311	73531.874	107128.059	92788.933	34106.849
■	System (System Calls/sec): WebServer2	9106.658	59907.732	100722.501	66228.158	22376.502
■	System (System Calls/sec): Server3	2087.107	3135.356	4705.936	2697.711	833.129
■	System (System Calls/sec): Server4	1327.757	2245.728	5121.947	1534.655	971.676
■	Run	0.0	1007.857	1366	1329	467.343

In this example, the two web servers, usatrame5151 and usatrame5152 show a moderately high rate of context switches (within the Microsoft guidelines), and a very high rate of system calls.

However, available memory remained adequate throughout the test (minimum of 1.6 GB), CPU utilization was moderately high (average of 60% to 67%) and reached a maximum of only 86%. Interrupts/sec. was low. Processor queue length was within acceptable limits (1.3 to 1.9 on average and reaching a maximum of 3).

All of this combined indicates that there was no bottleneck and that the application was nearing its peak capacity at the load of 1,300 simultaneous users.

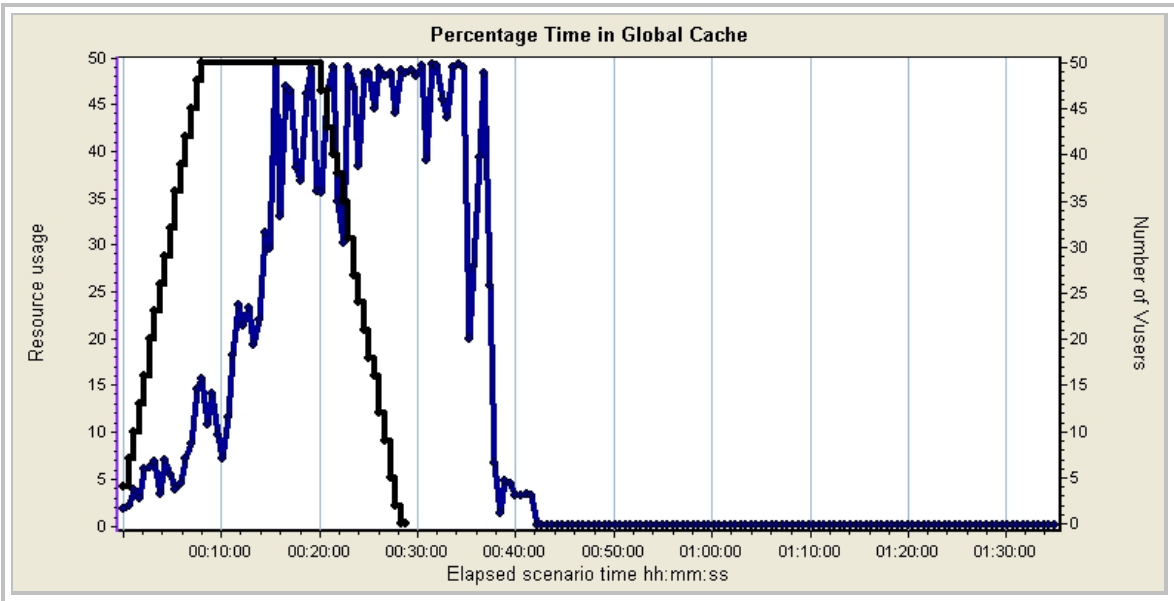
This is the closest example of a potential software bottleneck that was available, but close analysis revealed that it wasn't a bottleneck after all.

An Example of Thrashing Caused by a Memory Leak

This example shows a situation where a memory leak occurred on the IIS/Application server, causing the application to thrash. The application reaches a point where it is spending nearly 50% of its time in garbage collection trying to obtain enough memory to complete its tasks:

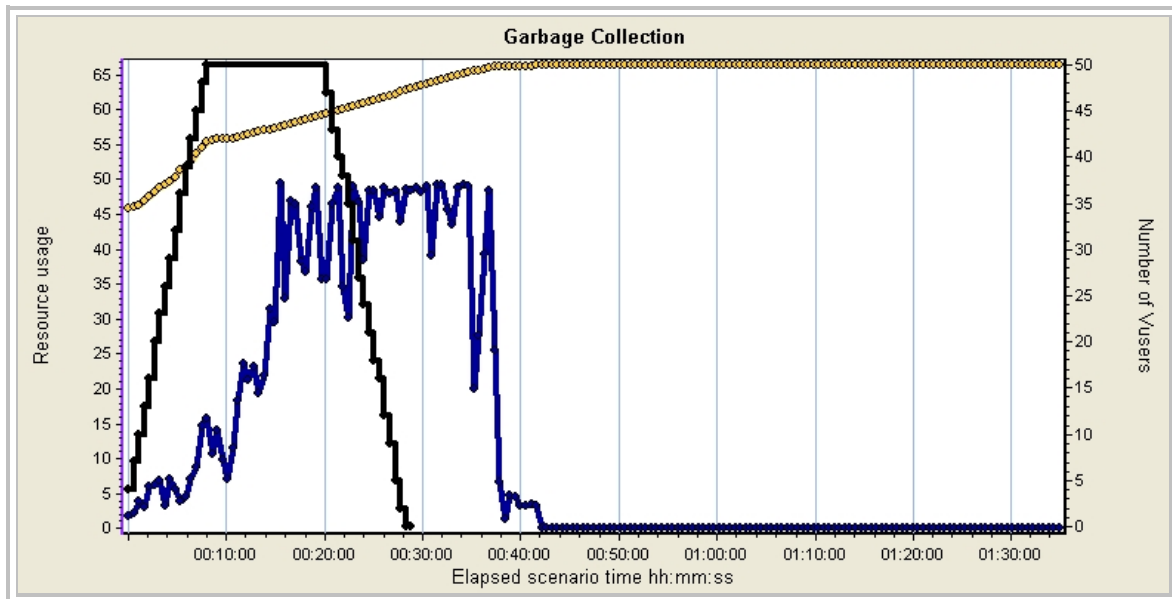
The application appeared to be reaching its size limit for the managed heap on the IIS machine. At this point, a Gen #2 garbage collection was forced, which caused the entire application to suspend until the garbage collection completed. The processor queue on the IIS machine backed up during the GC operation, and then released, flooding the database machine. The process continued to repeat itself for about 15 minutes after the load is removed and the queued transactions were worked off. At this point, the application recovered.




The time spent in global cache graph shows major cache activity at the point at which the heaps reach their maximum size.



Color	Graph	Scale	Measurement	Min.	Ave.	Max.	SD	Count
■	Windows Resources	1	.NET CLR Memory(_Global_) (% Time in GC):Server1	0.046	12.339	49.995	19.161	5712
■	Running Vusers	1	Run	N/A	N/A	N/A	N/A	N/A

The garbage collection graph shows the number of Gen #2 garbage collections rising rapidly. It also show that the time spent in garbage collection by the IIS machine reaches about 50% and remains there until about 15 minutes after the scenario ends.



Color	Graph	Scale	Measurement	Min.	Ave.	Max.	SD
	Windows Resources	0.01	.NET CLR Memory(_Global_) (# Gen 2 Collections): Server1	4592	6314.842	6642	531.444
	Windows Resources	1	.NET CLR Memory(_Global_) (% Time in GC): Server1	0.046	12.339	49.995	19.161
	Running Users	1	Run	N/A	N/A	N/A	N/A

The %CPU utilization graph shows that the IIS machine reaches a level of 25% to 30% and remains there until about 15 minutes after the scenario ends. The CPU utilization for the database server shows periodic spikes that are indicative of the back-end flooding that occurs when a Gen #2 garbage collection completes on the IIS machine and the application is re-started. The database server does not appear to be heavily loaded, except for these periods of back-end flooding. The IIS machine does not appear to be CPU bound, since the CPU utilization never gets higher than about 40%.

Once the server was re-booted, the thrashing problem disappeared.

IIS and SQL Server on the Same Machine

Generally, it's a bad idea to host IIS or SQL Server or Exchange on the same machine. The reason is that these applications can consume lots of memory. SQL Server, Exchange, and IIS by default try to allocate all available physical memory for itself, while IIS allows its .NET heap to grow until it runs out of memory. The result is that both servers can end up memory starved. Thrashing occurs as processes for each server attempt to obtain enough memory to complete their tasks. If these applications must be run on the same hardware it is best to hardcode a SQL max server memory parameter so it does not become overly aggressive with memory needed for the other applications.

This kind of situation can be spotted by examining the configuration of the application.

Diagnosing Software Aging

As software continues to run over a period of time, it tends to accumulate various types of errors, eventually causing degraded performance or complete failure. Types of errors can include: memory leaks, memory bloating, un-released file or database locks, other types of resource leaks, such as I/O handles, data corruption, un-terminated threads and processes, fragmentation of storage, and round-off errors.

This phenomenon is known as “software aging.” The aging can occur either in the application itself, a host server, such as IIS, or the operating system.

The problem can be diagnosed by restarting either the application or by rebooting. If the degraded performance disappears after either one of these actions is performed, this indicates a software aging issue. The next time this situation occurs, Task Manager can be used to look for specific processes that show signs of memory leaks, etc.

The average rate at which the application enters the degraded mode can be measured. It is important to note that this rate may or may not be dependent on the load that is applied. This can be determined through load testing, by running tests with different levels of load and plotting the results, or from operational data by calculating the average load from the time of the last restart until the application crashed or entered the degraded mode, and plotting the results. The slope of the line gives the load-dependent portion of the aging rate.

Once the aging rate is known, it is then possible to set up a schedule for restarting the aged components or rebooting.

A more thorough treatment of this topic can be found in, “Capacity Planning for Web Services,” Menasce and Almeida, Prentice-Hall PTR, 2002, section 11.8.

Diagnosing Garbage Collection Issues

During load testing, we occasionally observe spikes in the in all of the performance counters simultaneously across all of the application's servers, accompanied by temporary increases in response time. This document attempts to explain what causes this.

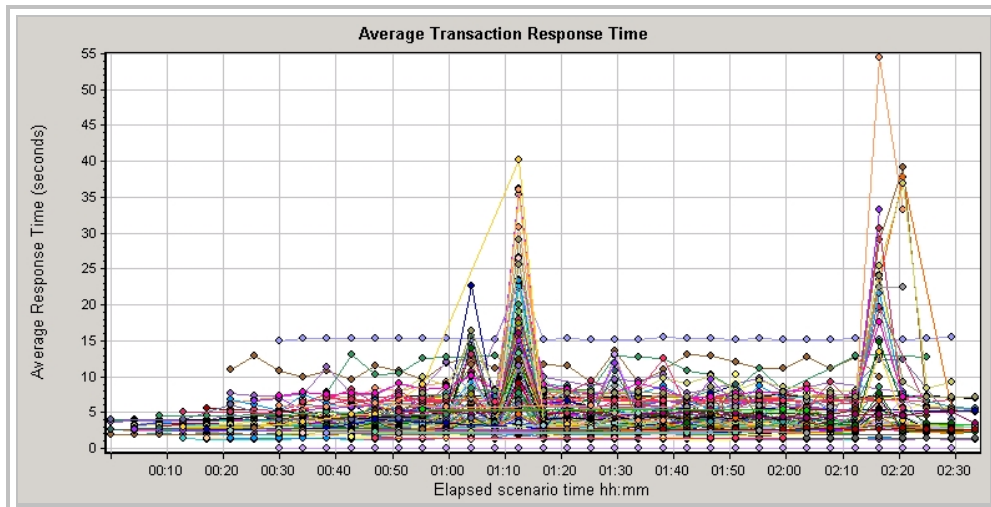
The following guidelines should prove useful. When we see:

- An occasional large spike in response times accompanied by a corresponding large spike in resource utilization (Perfmon counters), we should suspect that a Gen #2 garbage collection is occurring that is temporarily suspending the entire application, as described below.
- Gen #0, Gen #1 and Gen #2 garbage collections with a ratio of significantly less than 10:1 between lower and higher generations, it can be the result of two different conditions:
 - If the server is lightly loaded, it indicates excessive use of the 'finalize()' method, described below.
 - If the server is heavily loaded, time spent in garbage collection as well as the amount of memory available may indicate insufficient memory:
 - If the condition disappears after rebooting the server, and the ratios between garbage collections return to 10:1, this indicates a software aging problem, specifically, a memory leak. This is discussed in, "Diagnosing Thrashing in Applications."
 - If a re-boot doesn't correct the problem, try testing the application with a light load. If the ratio is still wrong, it indicates excessive use of the 'finalize()' method.
 - If a re-boot doesn't fix the problem, but the ratio returns to 10:1 under light load, it indicates that memory is a bottleneck on the server and needs to be increased.

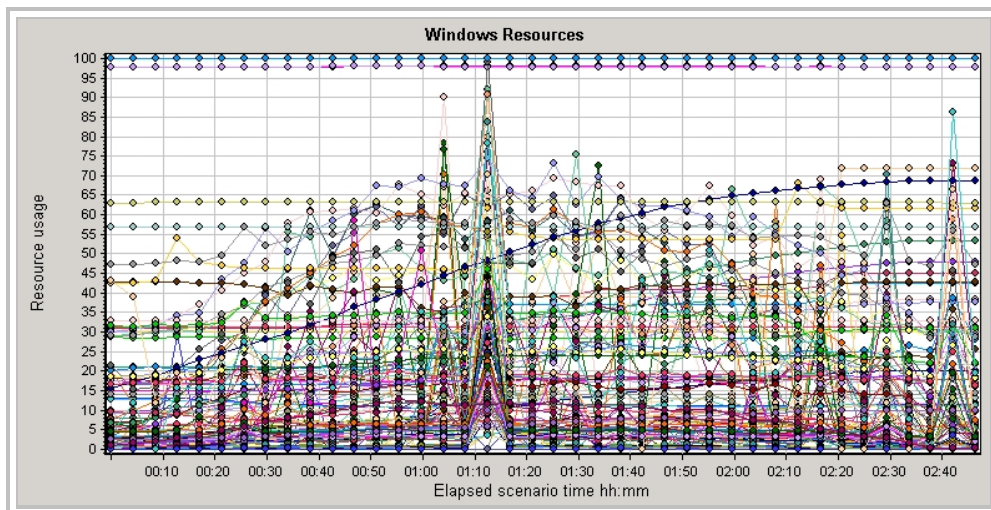
The following sections provide detailed explanations of the garbage collection spike and the reason that overuse of the 'finalize()' method causes performance problems.

The Garbage Collection Spike

The following graph shows average response times for a large number of different types of transaction types (represented by the differently colored lines) occurring simultaneously:

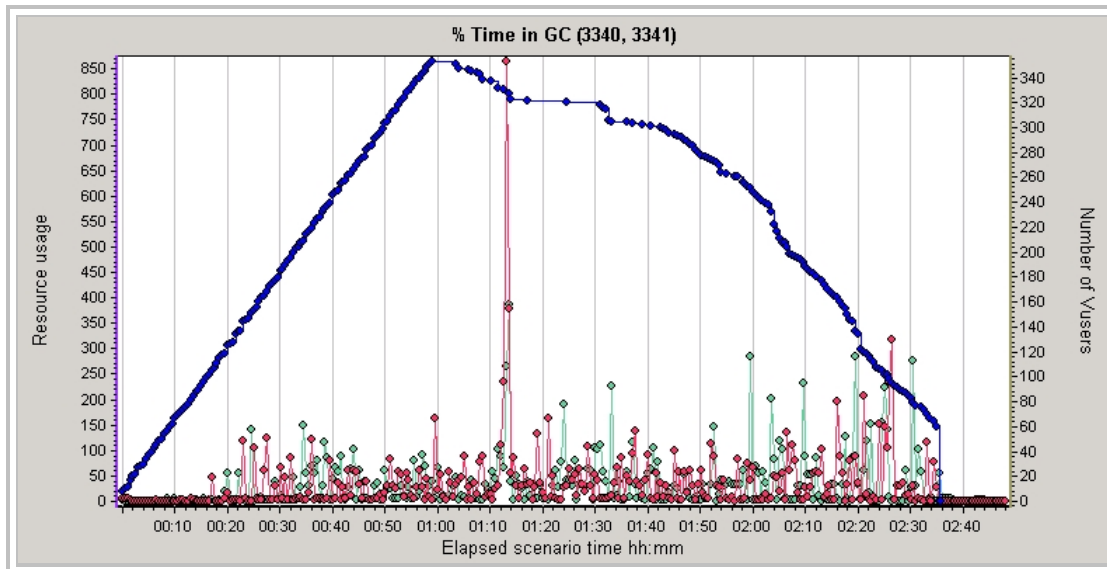


Note the spikes occurring at 1:12 hours and 2:17 hours. Now look at the graph of the combined Perfmon counters, with spikes occurring simultaneously in all counters at the same times.



Please excuse the ‘busy’ nature of the graphs. The individual lines aren’t important here, only the pattern that they form.

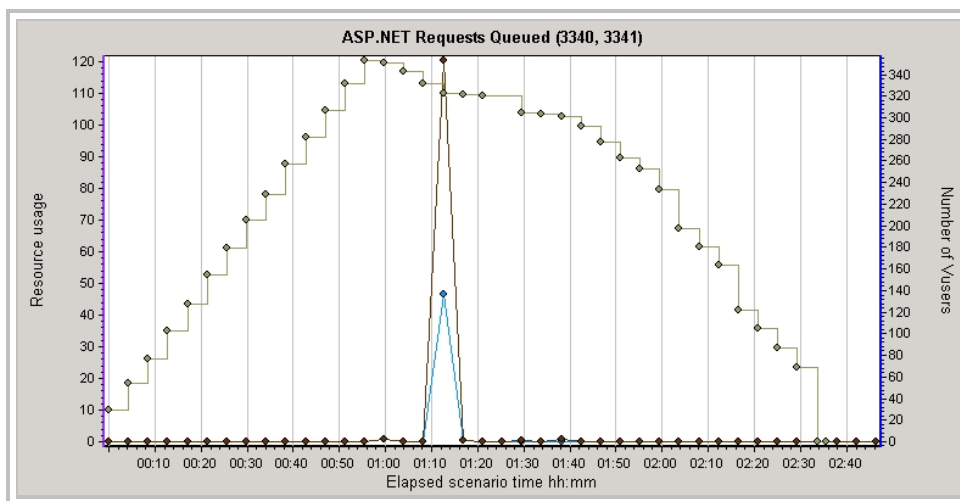
This graph shows the time spent in garbage collection on the web servers – the only servers that actually perform garbage collection, as one of the functions of the managed heap:



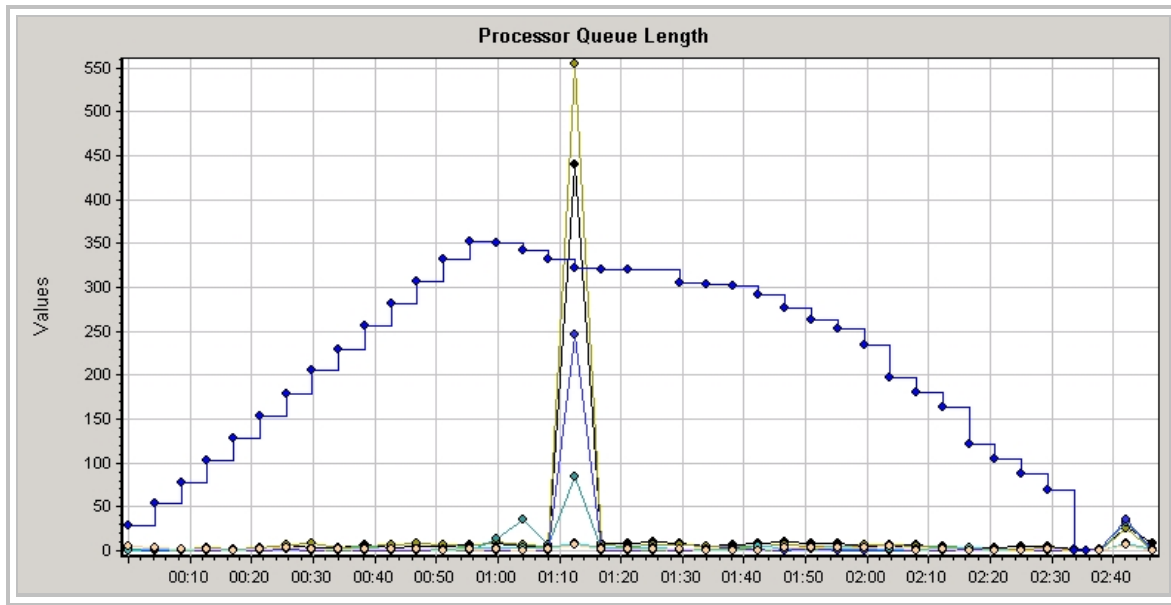
We see a substantial spike simultaneously occurring at about 1:12 into the test on both servers and a series of smaller spikes occurring simultaneously on both servers in the 2:17 to 2:30 time frame.








The spikes at 1:12 and 2:17 are reflected in the response times, and the spikes after the 2:17 spike are also reflected in the Perfmon counters.

This graph shows queuing of ASP.NET requests on the web servers:



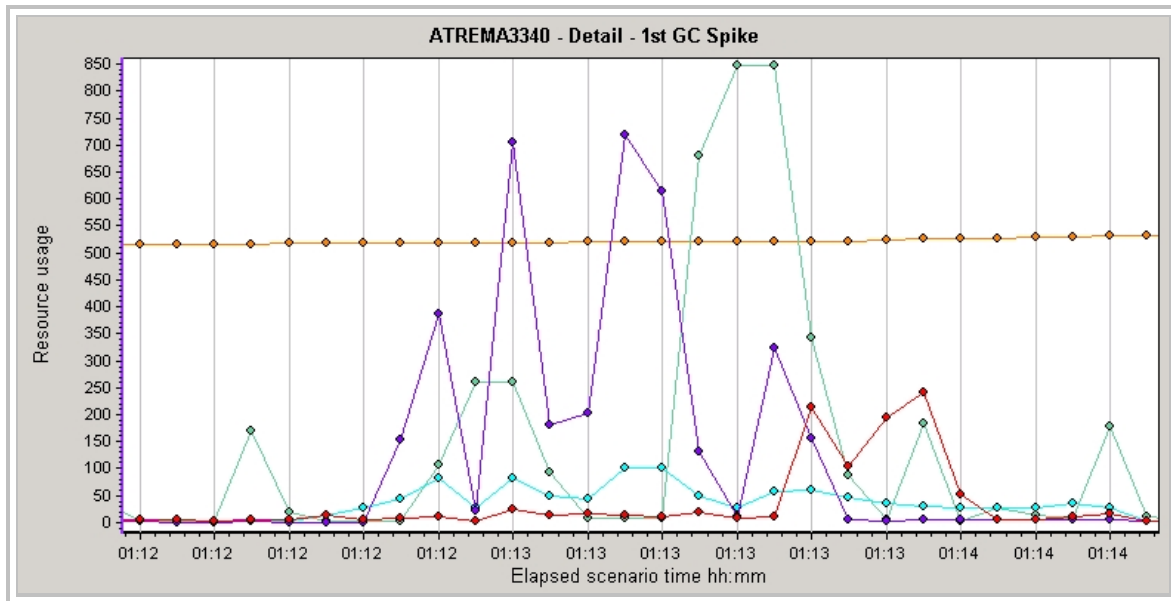
And this graph shows a spike in the processor queue length in all six servers at the time of the first spike:








Color	Scale	Measurement	Min.	Ave.	Max.
	100	System (Processor Queue Length):Web Server 1	0.0	0.161	82
	100	System (Processor Queue Length):Web Server 2	0.0	0.179	85
	100	System (Processor Queue Length):App Server 1	0.0	0.05	49
	100	System (Processor Queue Length):App Server 2	0.0	0.08	61
	100	System (Processor Queue Length):DB Server	0.0	0.026	1
	100	System (Processor Queue Length):File Server	0.0	0.019	1
	1	Run	N/A	N/A	N/A

In the above graph, the web servers are Web Server 1 and Web Server 2, the application servers are App Server 1 and App Server 2, and the database servers are DB Server and File Server. Notice how the maximum queue length decreases as we get farther from the web servers. File Server is used for storing file images in SQL Server, and is used less than DB Server.

The following graph shows the details of what is happening on the web servers. First the processor queue length begins to increase as the *entire application* is suspended for a Gen #2 garbage collection, and transaction requests continue to arrive from the clients. Next, the garbage collection activity itself is performed. Finally, the application is restarted and a flood of queued ASP.NET transaction requests are processed as rapidly as possible, indicated by the system calls/sec counter.



Color	Scale	Measurement	Min.	Ave.	Max.
	1	.NET CLR Memory(_Global_) (# Gen 2 Collections): Web Server 1	149	632.344	1151
	10	.NET CLR Memory(_Global_) (% Time in GC): Web Server 1	0.0	3.145	87.7
	1	Processor(_Total) (% Processor Time): Web Server 1	0.0	3.239	100
	10	System (Processor Queue Length): Web Server 1	0.0	0.161	82
	0.001	System (System Calls/sec): Web Server 1	1650.099	5502.171	636656.415

What is Happening

What is happening is that this application is making excessive use of the ‘finalize()’ statement, forcing de-referenced objects to go through two generations of garbage collection rather than one. As long as there is adequate memory allocated to the application’s heap, there is no impact. However, some objects are more persistent than others, and would naturally survive to a Gen #1 garbage collection anyway. When they are forced to go through two generations of garbage collection, they eventually end up as Gen #2 de-referenced objects.

Once the point is reached where Gen #1 collection cannot free up enough memory to satisfy the needs of the next transaction request, a Gen #2 garbage collection. Since a Gen #2 collection compresses the entire heap, the entire application must be suspended before the Gen #2 garbage collection can be performed.

During the period when the application is suspended, the transaction request queue builds up as new transactions arrive. Once the garbage collection activity is completed, the server processes the queued transaction requests as quickly as possible, causing a cascading effect as the sudden burst of high activity ripples out through the application servers and on to the database servers.

The spikes that occurred after the 2:17 were reaping operations. The test had already completed, and the application reaped its unused but still allocated objects after an inactivity timeout. There is no spike in response time after the 2:17 spike because the application is idle at this point.







There is no spike in processor queue length at 2:17 because the number of transactions backed up is smaller, no new transaction requests are being received, and the resulting load is insufficient to cause the processors to overload. The queuing effect is evident on the response times, however. There is no backup in the ASP.NET request queue, and no corresponding backend flood when the reaping garbage collections complete. In this example, only the initial spike at 1:12 and the spike at 2:17 had any impact on performance.

It should also be noted that the major spikes at 1:12 and 2:17 occurred when both web servers went into Gen #2 garbage collection simultaneously. This is an indicator that the application was properly load-balanced. When only one server goes into Gen #2 garbage collection, the other server continues to process transaction requests and the effect is far less pronounced.

Diagnosing Excessive Use of the ‘finalize()’ Method

Microsoft literature advises that Gen #1 garbage collections should normally occur at about 10% of the rate of Gen #0 garbage collections, and Gen #2 garbage collections should occur at about 10% of Gen #1 garbage collections.

The number of garbage collections for the example application was monitored, and the following table shows the results:

Color	Scale	Measurement	Total
	0.01	.NET CLR Memory(_Global_) (# Gen 0 Collections): Web Server 1	4798
	0.01	.NET CLR Memory(_Global_) (# Gen 0 Collections): Web Server 2	5349
	0.01	.NET CLR Memory(_Global_) (# Gen 1 Collections): Web Server 1	2027
	0.01	.NET CLR Memory(_Global_) (# Gen 1 Collections): Web Server 2	2160
	0.01	.NET CLR Memory(_Global_) (# Gen 2 Collections): Web Server 1	1151
	0.01	.NET CLR Memory(_Global_) (# Gen 2 Collections): Web Server 2	1186

The ratio is clearly way off from Microsoft’s recommendation. As stated above, this is caused by excessive use of the ‘finalize()’ method, which causes a de-referenced object to be promoted to a higher generation before it is collected.

When questioned about excessive use of the ‘finalize()’ method in the example application, the developers acknowledged that they were using it for nearly every object.

What happens is that when an object with a 'finalize()' method is de-referenced, it is placed on the 'Finalize' queue for that generation, and a reference to it is placed on the 'Freachable' queue, so that it won't be removed by the garbage collector before the 'finalize()' method is executed.

The 'finalize()' method is executed the next time a garbage collection for that generation is performed. Since the object is still reachable because it is on the 'Freachable' queue, the garbage collector promotes it to the next generation. The object is then removed from the 'Freachable' queue and ultimately removed from the heap.

The reason for this complexity is that the garbage collection activity and the execution of 'finalize()' methods are performed by separate, asynchronous processes, and there is no way to know which action is performed first.

Evaluating Stability Errors

Many times, when performing load tests, random errors are encountered of the type, "Internal Server Error" with type code 500. This is the generic, "something went wrong on the servers" error, and doesn't reveal much about the cause of the error. It is important to provide some insight into the potential causes of these errors so that the development team can diagnose and correct them.

If the errors appear to be random in nature, with no apparent pattern, then they can be classified as stability errors. Stability errors generally fall into five categories:

- Race conditions
- Boundary/limit problems (e.g., buffer overruns)
- Resource leaks
- Deadlocks
- Timeouts (but not timeouts caused by deadlocks)

It must be remembered that the load test acts as an error detector. When random stability errors such as race conditions or deadlocks occur, increasing the load increases the probability of detecting these errors – in proportion to the increase in the load.

Resource leaks tend to manifest themselves as slow buildups over time in such things as memory utilized or open file handles, etc. This type of stability problem tends to manifest itself as a catastrophic failure, rather than as randomly occurring errors.

Boundary/limit errors and timeouts (not caused by deadlocks) tend to be *caused* by higher loads, so with these types of errors, the load test not only acts as an error detector but as an error inducer. These differences aid in diagnosing the cause of the errors.

The question arises, "Are the errors caused by the increased load, or are they simply randomly occurring stability errors caused by such things as race conditions, which are merely detected with greater frequency at higher loads?"

Diagnosing performance problems is typically 3 parts pattern recognition and 1 part intuition based on experience and understanding of how applications are typically implemented. This paper provides some insight into the pattern recognition part of the exercise, with respect to the types of things that can cause application thrashing and how to recognize these situations when they occur.

For this reason, the following guidelines should prove useful:

- Evidence of "Poisson Clumping" of errors under heavy loads, as described below, is a indication that the errors are load-induced.
- Vary the test load by +/- 10%. If the error rate scales linearly, this indicates that the errors are not load-induced. If the error rate accelerates with increased load, this indicates that the errors are load-induced.

The following sections explain all of this in detail.

Diagnosing Load-Induced Errors

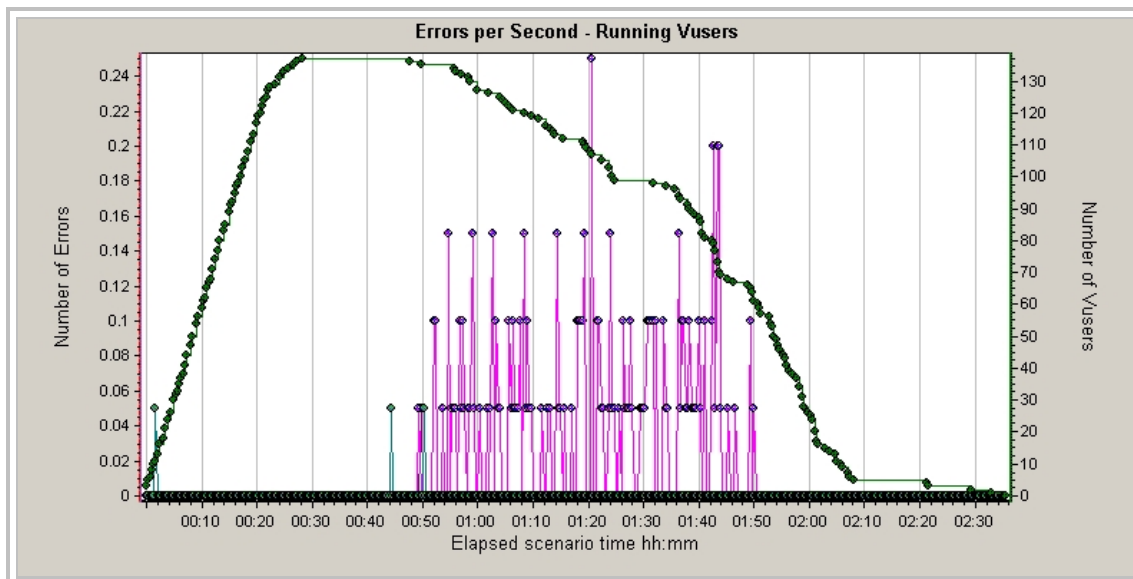
There are two ways to diagnose load-induced errors. The first way is to vary the load by +10% and -10% around the point where the errors were observed. If the number of errors appears to increase linearly with increased load, then the errors are most likely caused by race conditions or deadlocks.

If, on the other hand, the slope of the errors vs. load line increases significantly when going from the reference load to reference plus 10%, as opposed to the slope of the errors vs. load line when going from the reference load to reference minus 10%, this would indicate that the errors are load-induced. This would place them in the category of either boundary/limit problems (e.g., buffer overruns) or timeouts not caused by random deadlocks.

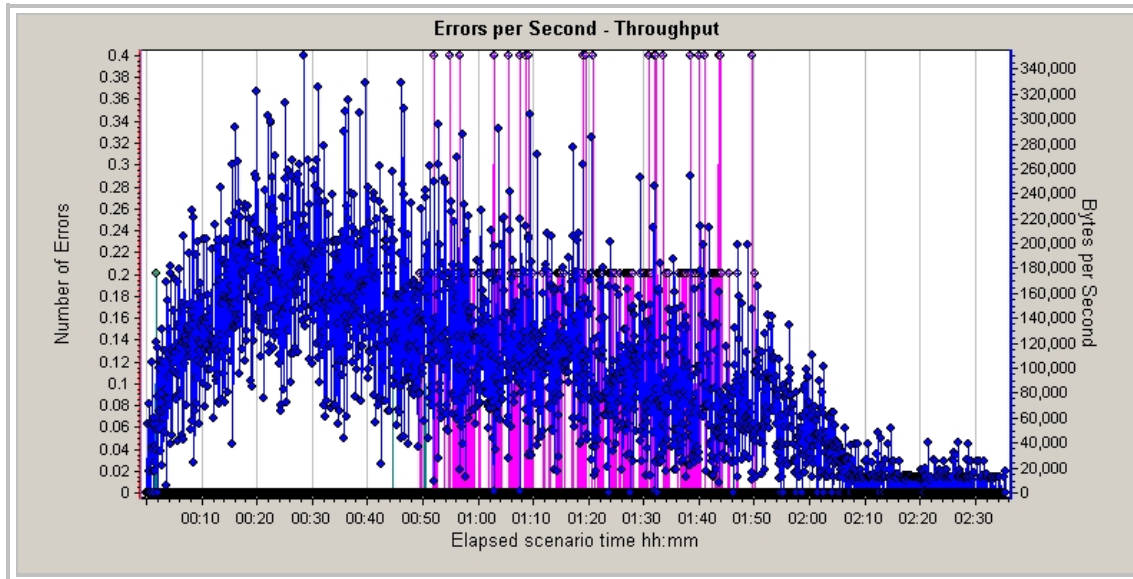
The second way to diagnose load-induced errors is to observe the pattern of the errors (if any). As long as the load test is designed in such a way that the load generators behave like Poisson processes, i.e., the time between iterations is exponentially distributed, then it should be possible to observe a pattern in the errors known as “Poisson clumping.”

This phenomenon is a characteristic of all queuing systems when they get overloaded, and is frequently observed in network appliances.

With sufficiently high loads, the peaks in the error rate should correspond fairly closely with the peaks in the throughput, when these two values are plotted together. The first graph below illustrates Poisson clumping observed in one of our load tests:



This next graph shows this same clumping plotted against throughput for the same load test:



The reason that the peaks in the error rate correlate with peaks in the throughput is that up until a level of load is reached where significant queuing occurs, throughput increases with load.

In the charts above, the errors didn't start until 50 minutes into the test because the problem was related to the design of a third party application that was retrieving an excessive amount of data from the database for an intermediate processing step of each transaction. The test was designed in such a way that new data was being added to the database throughout the duration of the test, and it took 50 minutes for the size of the database to increase to the point where the application could no longer handle the load.

The reason for varying the load by only 10% around the reference load is to avoid raising the load to such a level that load-induced errors start occurring in addition to the errors that were already occurring.

Footnotes to the Flowcharts

1.1 When an application saturates at higher loads, its response time degrades as more and more transaction requests are queued. Simultaneously, application throughput reaches its maximum value. When response time is plotted against load, or throughput is plotted against load, the resulting curves resemble ‘hockey sticks’ – hence their name. Response time can be plotted against load in LoadRunner with the ‘Response Time Under Load’ graph. A more complete explanation, as well as examples of the ‘hockey stick curves’ are provided in, “Diagnosing Application Saturation.doc.”

1.2 By much, much greater, we generally mean 5 or more times as great.

2.1 A ‘large number of processes/threads’ is a subjective evaluation, and depends on the capacity of the host. For a DL380, it might be 10,000 or more.

2.2 This is a subjective evaluation, but significantly more than 2% can be considered to be an indication of problems.

3.1 This is a subjective evaluation, but if more than half of the machine’s memory is freed up by a re-boot, it may indicate a software aging problem.

5.1 This is a subjective evaluation, but there are two key points to observe. First, the errors must be ‘random’, i.e., they don’t appear to be related in any way to any particular operation. Second, ‘a lot’ would be interpreted as a significant portion of the total transactions being in error. For example, if a test had a total of 30,000 transactions, 5,000 errors would be a lot, 10 errors wouldn’t. For anything in between, it’s up to the analyst to determine whether or not there is a serious enough problem to warrant further investigation.

5.2 ‘Poisson Clumping’ is a characteristic type of error distribution common to any type of queuing system where the arrival rate of requests follows a Poisson distribution. Examples of such systems include network routers, IIS, application servers and database servers. An entire application hosted on multiple servers is also a queuing system. When errors are plotted against throughput in LoadRunner, using fine granularity, this will appear obvious. An example of this is given in, “Evaluating Stability Errors with LoadRunner.doc.”