Perceived Performance

Tuning a System for What Really Matters

White Paper by Tim Mangan
June 2004

Introduction

This white paper explains the subtle difference between tuning a computer for maximum computational performance and tuning it for what we call "perceived performance." Perceived performance is concerned with human perception of system performance – and ultimately organizational productivity. Specifically, this paper addresses perceived performance associated with multi-user systems based on Microsoft Windows Terminal Server – and optionally Citrix MetaFrame.

In working with IT professionals, we have found the concept of perceived performance to be poorly understood. Often when dealing with system performance, they focus on things they have read and heard in the past. It is rare that an IT professional actually has the time (or budget) to become truly trained in performance tuning. The techniques and objectives picked up along the way may or may not be appropriate for what they are trying to accomplish today. While advances in operating systems and hardware account for much of this misguided effort, the different challenges associated with Terminal Services (and the tendency for otherwise excellent sources to provide only passing reference to the difference when operating in this environment) require a different mindset, and a somewhat different set of goals.

This paper does not represent an end goal, but part of the education process involved in expanding your understanding of performance tuning. Curious IT professionals may investigate more practical texts, which help to organize your efforts and describe tools you can use to measure cause and effect. In addition there are countless Web sites offering tricks and tips on this subject. With an understanding of perceived performance, you will better understand why and when specific techniques are appropriate for enhancing your Terminal Server environment.

Computational Performance

Computational performance is the term we use to describe a methodology where one analyzes the system with a goal of improving the peak capacity by eliminating unnecessary actions that reduce the overall computational capacity of the system.

Computational capacity is the total amount of useful work that can be accomplished on a system in a define, fixed period of time (e.g., spreadsheets re-calculated, reports generated, print jobs printed, 3d-images edited – or games of solitaire if the system isn't locked down).

In focusing on computational performance, one analyzes where CPU cycles are spent and where more (or occasionally less) hardware or software resources can increase a system's computation capacity.

Most reference books on performance use computational performance as the methodology for helping organize your performance-related efforts. These books, the methodologies they teach, the tools they train you to use, and the tricks they teach, can be of immense value.

A simple example of computational performance follows.

- **Subject:** memory.
- **Tools**: Use the performance monitor to look at the memory utilization and paging activity on a system.
- **Solution:** add more physical ram to the system.
- **Result:** less paging activity, which frees up the system to perform more useful work, rather than wasting all that effort (CPU Cycles, Bus, and Disk Activity) just sliding things in and out of physical memory.

Computation performance is an important technique. However, if you are optimizing a multi-user system such as a Terminal Server, then you need also consider (and temper your solutions with) *Perceived Performance* thinking.

Perceived Performance

Perceived performance is our term for describing a methodology where one analyzes the system with a goal of improving user productivity. It focuses on issues that affect the performance of the system as perceived by the people using it.

Few IT professionals run their server farms at 100 percent CPU loading (and none should). Typically, the farms are configured so that average loading is at a much lower value – 60 or 30 percent. Much depends upon the environment and budget.

Quite often, IT starts out planning and deploying to such a figure, however more servers are added not because of some magic utilization number (although the number may be used in purchase justification), but because people are complaining about how long it takes to do their jobs.

A discipline that uses perceived performance achieves optimal user productivity. While accomplished using primarily the tools and techniques of computational performance, the thought process and analysis under perceived performance is geared toward different objectives to reach a different goal. Table 1, below, summarizes the key difference in goals and objectives of these two methodologies.

	Computational Performance	**Perceived Performance**
Goal	Improve peak capacity	Improve user productivity
Objectives	Eliminate unnecessary actions that reduce the overall computational capacity of the system.	Modifications aimed at reducing delays in responsiveness to user actions.

Table 1 - Goals and Objectives of Performance Methodologies

Examples of Perceived Performance

Example 1: Context Switching

Context switching is the overhead that occurs when the operating systems switches between different tasks[i]. Each time a CPU switches from working on one thread to another the CPU must save information necessary so that it can later resume processing of that thread exactly where it left off. Typical information includes the contents of CPU registers, including instruction address and stack pointers.

Computational performance dictates that by reducing the number of context switches, you reduce the overhead associated with the switching, and thus increase the overall computational capacity of the system. Discussions about overhead have a deep history. In years gone by, a class of operating systems known as "Real-time Operating Systems" was rated almost entirely by their context switch time. Today IT professionals attempting to tune their system often use the Performance Monitor to see the number of context switches per second appearing on their server as a measure of system performance.

Perceived performance thinking is not concerned with "reasonable" levels of context switching. Why? Because the ratio of CPU cycles per context switch cycle have been reduced significantly on our systems over the years.

In part, OS/compiler vendors have learned how to produce an absolute minimum context switch time. In part, chip manufacturers like Intel have made processors more capable as well. And finally, the constant doubling of processor speed has massively increased the number of CPU instructions that take place in a given period of time, while the average thread run time (the average clock time that a thread runs without being interrupted) is going down.

In his paper, High-*Performance Programming Techniques on Linux and Windows*, Dr. Edward Bradford[ii] shows a method that attempts to measure the context switch time on a Windows 2000 Server. Bradford indicates that it runs about 1.8 usecs on a 650 MHz processor. (NOTE: a usec is 1/1000 of an msec, or one millionth of a second). I analyzed his test and made a few minor changes, both to reduce potential unnecessary paging overhead and to account for start/end thread swaps that were not accounted for in the calculations. We tested this on Windows 2000 and 2003 systems, ranging from 666MHz to 2.4GHz, and generally came up with figures in the 1.6 usecs/context switch range[1].

The bottom line: if a change is made to a modern server that even doubled the number of context switches per second you do reduce the computational capacity by a small amount. For example, by lowering the maximum run time of a thread (under Microsoft Windows this is referred to as a number of "quantum") you potentially increase the number of context switches that would added in the case of a CPU bound thread that would normally run for the full quantum limit. Therefore, moving the quantum value from 36

[1] This value still contains much more than the context switch time because it captures other OS activities including interrupts, APCs and DPCs, but it is a "good enough number". The actual value would be lower.

(roughly 180ms on a multi-processor) to 6 (30ms)[2] in the presence of a CPU-bound thread would add a context switch overhead of 0.005% (see Equation 1).

$$(6cs * 1.6u\,\text{sec}/cs)/180m\,\text{sec} = 0.005333\%$$

Equation 1 - Added context switch overhead (worst case scenario)

However we run our servers with large amounts of spare capacity. Sixty percent loading plus 0.005 percent loading still looks like 60 percent loading. The flip side of reducing the quantum (which increased the overhead ever so slightly) is that responsiveness is improved. Instead of Task-A waiting up to 180ms for a competing Task-B to complete its quanta before Task-A can begin its work, it waits for 30 to 60ms.

A blink of an eye is about 100ms. When the user clicks on a button and has to wait 500ms to see the menu appear they feel it. After 750ms they might click on it again, thinking they missed the target they tried to click.

Can the user tell the difference between a 60 and a 180ms delay? Yes, because Task-A will likely swap out multiple times to complete the work of a very simple request. **The responsiveness the user feels is the result of multiple small delays that add up, what we call *serialized delay***.

Perceived performance is about tuning the system to reduce the delays that decrease responsiveness to user actions. And the effect of responsiveness can be measured. Either empirically (how many users can you log onto the system before they complain), or experimentally (measuring responsiveness to a particular action).

An Experiment in Responsiveness

We have developed a test that repeatedly measures the time required for launching and then shutting down a small, Win32-based GUI executable. This involves the system allocating a new process, the loader loading up the executable from the disk, a few threads are spawned and a window is presented. Then the GUI sleeps (waits on a timer) for a short period and terminates itself. Even on an idle system this process will cause hundreds of context switches. We know, from running on an idle system for thousands of passes, what the best possible time to complete this process is. The test is run under a given load with certain system parameters to determine an average completion time. From this value we subtract the minimum time and we have a measure of the current responsiveness of the system – the lower the value (the closer to the minimum time) the more responsive the system. Change system parameters and run the same test. Did responsiveness improve or get worse?

Our test allows us to accurately measure whether a change to the system improves perceived performance or not. As is often the case when it comes to performance, the logical result may not be what you find if you actually run the test.

[2] Changing from "Background Services" to "Application" mode.

Ultimately, however, it is the empirical test that proves whether the change is important. Does this change allow more users to share this system? Changes that individually (or as a group) do not increase the user capacity of the system, even if the experimental tests show some improvement, are in the long run probably a bad idea. They make the system more complex to set up and more complex to maintain. The problem is in devising an empirical test that can be measured.

We can devise a measurable empirical test using the experimental test we described above. In the experimental test, we run our test program on a system while varying the number of users, until the responsiveness exceeds some magic value. And what is the magic value? It is not one-size fits all, rather, it must be determined based on your specific situation.

To determine your response rate, you start by using a given system configuration. You need only add more users until you subjectively determine that the performance is no longer acceptable in actual tasks. Once you find the maximum number of users, run the responsiveness test with that load and you have your magic value.

Now you can make system changes and measure the number of users that can run until the magic value is exceeded.

Example 2: Delay Profile Graph

Since publishing the original version of the paper, some additional work has gone into measuring the effect of serialized delay imposed on an application by others sharing the same system. This work was done in the context of investigating how a tool, such as triCerat Simplify Resources, can be used to improve performance on a Terminal Server, as perceived by the users. The impact of adding such software to a server is encouraging. Publishing an updated paper allows us the opportunity to present some of these results, as well as a new way that we have developed to visualize the performance information..

In the previous example we describe a test to measure the delay of a very simple program that goes through a few hundred context switches to perform a task. Each time there is a context switch there is a chance that another program thread will run instead of our intended application. When this occurs a small, incremental, delay occurs – what we called *Serialized Delay.*

Running the test thousands of times on a lightly loaded system allows us to determine the minimum period possible for the test. This minimum period is designated as a serialized delay value of zero. The serialized delay of any other run of the test is determined as the test time minus the minimum value.

Calculating the serialized delay for each run allows us to create what we call a *Delay Profile*. Represented graphically, this provides a visualization of the perceived performance effect that a user feels when other software is running on the server.

Figure 1 (below) illustrates how a Delay Profile of our given test application on an unloaded system. The test application was run 600 times and the serialized delay calculated for each test. The height of the chart indicates the number of times that this serialized delay was measured to be the delay indicated (in 100ths of a second on the horizontal axis). This allows us a quick visualization of not only the average delay (the peak), but of the variation in delay experienced by the user.

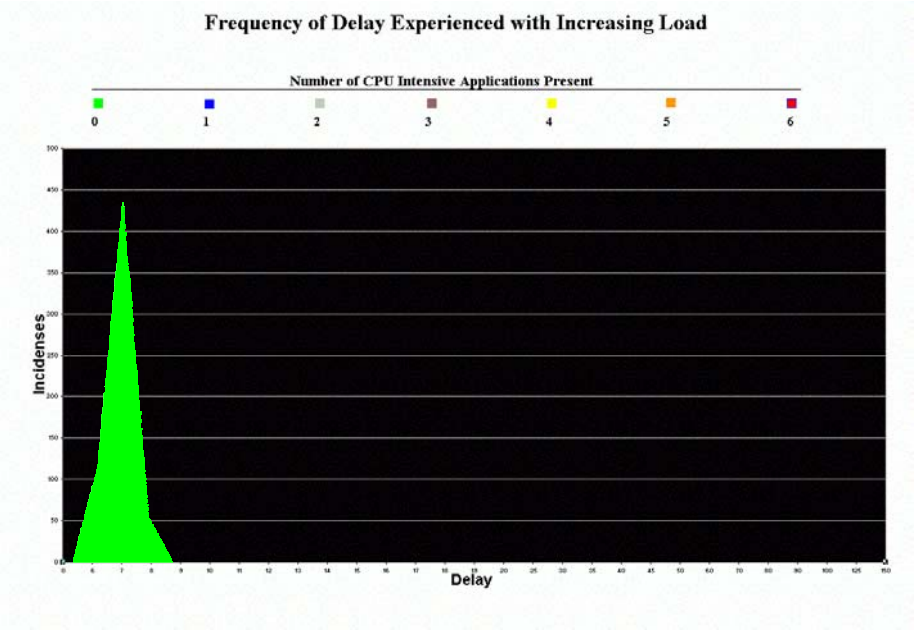

Figure 1 – A Delay Profile with no load

As can be seen in the graph, even on a stable and virtually idle system there is a small variation in the serialized delay due to background system software and events. Both the average delay and variation in delay are key parameters in determining the performance that a typical user will perceive. A user can adjust to a 50 millisecond delay between typing a key and seeing it display on the screen when the delay is consistent. But the performance seems so much worse when the delay varies between 20 and 200, and their typing speed suffers. The variation shown in Figure 1 is small enough that the user would not notice.

Figure 2 (below) illustrates the Delay Profile of the same system with additional loads.

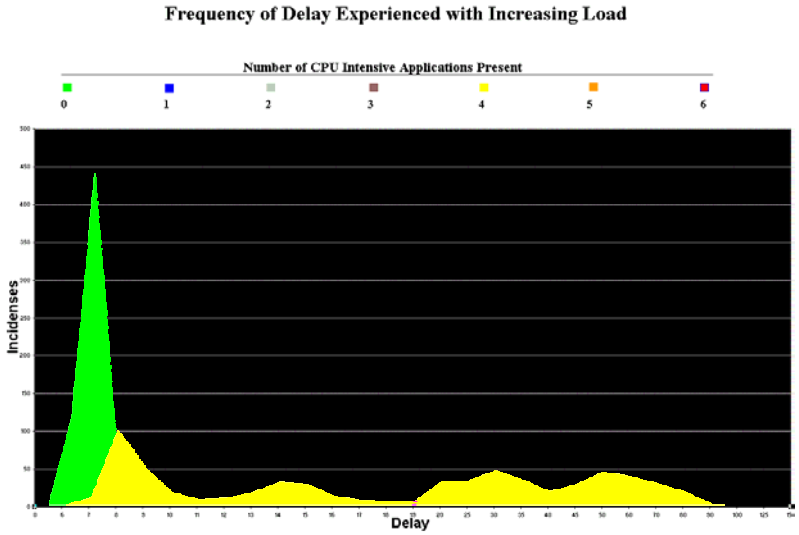

Figure 2 - Delay Profile with and without load

Each added load attempts to consume a complete CPU on the system – in this case a dual processor with Hyper-Threading enabled on Windows Server 2003 (effectively having four processors). As is shown in yellow, not only is the average serialized delay larger when load is added, it is also quite variable. Numerically, this variability can be described by calculating a standard deviation; however, I happen to like a nice picture instead. Figure 3 (below) represents the complete results from a series of tests. In this Delay Profile Graph one can see how both the average delay and variability in delay tend to increase. This illustrates the performance, as perceived by end users performing typical tasks such as word processing.

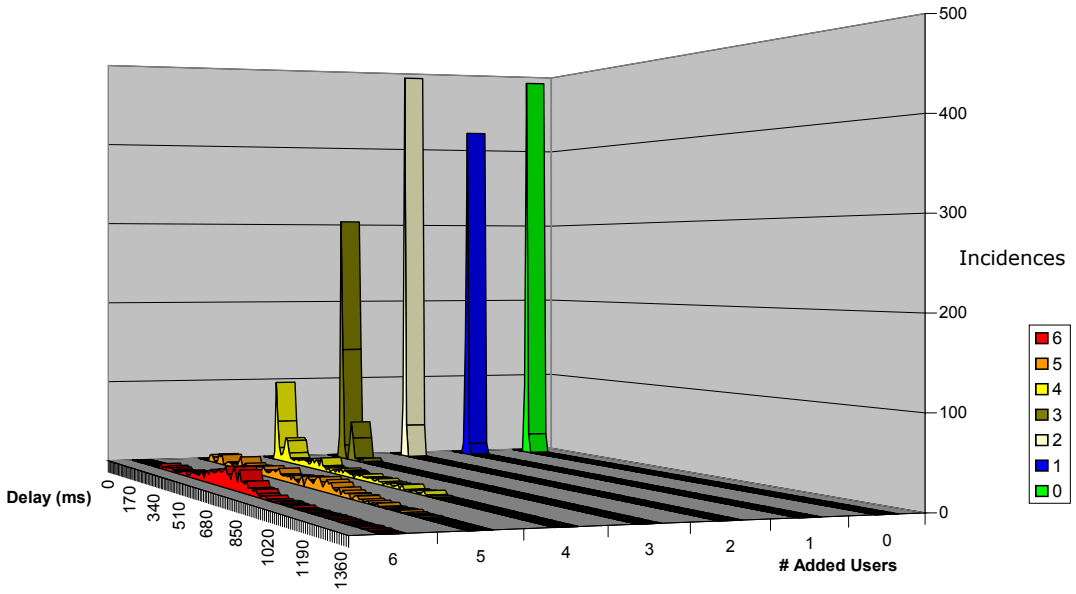

Figure 3 - Delay Profile Graph under loads

In figure 4 (below), we present the same tests after implementing triCerat Simplify Resources version 3. The application modifies Windows thread scheduling algorithms, applying a Quality of Service (QoS) model to thread scheduling.

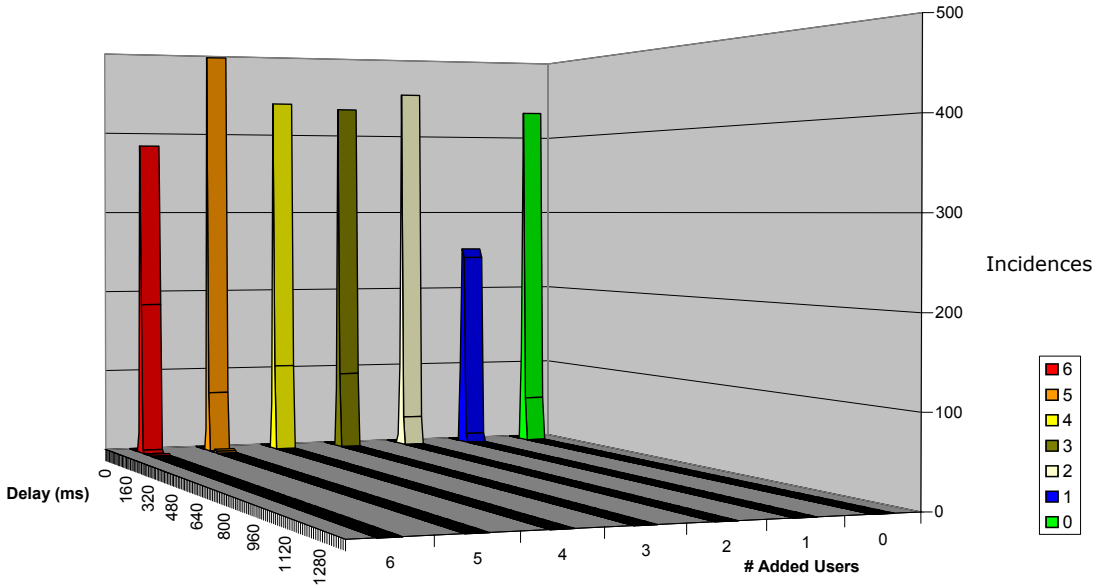

Figure 4 - Delay Profile Graph with QoS Added

Simplify Resources improves the average serialized delay, as well as the variation, resulting in improved performance as perceived by the users even under extraordinary loading conditions. In cases where user performance limits scalability of a Terminal Server, this can allow more users to be supported on a given system before users begin to complain.

Conclusion

In this paper, we make a distinction between two methodologies for improving the performance of Terminal Server systems. In doing so, we are in no way implying that addressing computation performance is a mistake. Rather, we hope readers will evaluate and implement approaches to performance tuning based on the unique characteristics of their systems and business conditions, instead of using a cookie-cutter approach that was developed as a one-size fits all solution for Terminal Server performance problems.

References and Resources

There is a great deal of information available on the subject of performance tuning Windows and Terminal Server. Some useful references and resources follow:

Windows 2000 Performance Guide, Mark Friedman & Odysseas Pentakalos, O'Reilly & Associates 2002

*Terminal Server Performance Tuning***,** Brian Madden, September 2003, http://www.brianmadden.com/papers

---

[i] *Scheduling Priorities: Everything You Never Wanted To Know About OS Multi-Tasking*, TMurgent Technologies, July 2003, http://www.tmurgent.com/SchedulingWP.htm

[ii] *Context Switching Part 1: High Performance Programming Techniques on Linux and Windows*, Dr. Edward Bradford IBM DeveloperWorks, July 2002, http://www-106.ibm.com/developerworks/linux/library/l-rt9/?t=gr,lnxw02=RTCS