Measuring Processor Utilization in Windows and Windows applications

Mark B. Friedman
Demand Technology Software

Introduction.

This paper discusses the legacy technique for measuring processor utilization in Windows that is based on sampling. This technique for measuring processor utilization is efficient and generally adequate for capacity planning. However, it lacks the precision performance engineers require for application optimization and tuning, particularly over small measurement intervals. The paper then introduces newer techniques for measuring processor utilization in Windows that are event-driven. The event-driven approaches are distinguished by far greater accuracy, enabling the reconstruction of the precise path that threads, processes and processors take when they execute. Gathering event-driven measurements entails significantly higher overhead, but measurements indicate this overhead is well within acceptable bounds on today's high powered server machines.

As of this writing, Windows continues to report measurements of processor utilization based on the legacy sampling technique. The more accurate measurements that are derived using events are gaining ground, however, and can be expected to supplant the legacy measurements in the not too distant future.

While computer performance junkies like me relish the prospect of obtaining more reliable and more precise processor busy metrics, the event-driven measurements do leave several very important issues in measuring CPU utilization unresolved. These include validity and reliability issues that arise when Windows is running as a guest virtual machine under VMware, Zen, or Hyper-V that impact the accuracy of most timer-based measurements. (In an aside, mitigation techniques for avoiding some of the worst measurement anomalies associated with virtualization are discussed.)

A final topic concerns characteristics of current Intel-compatible processors that undermine the rationale for using measurements of CPU busy based solely on thread execution time. We discuss the value of using internal hardware measurements of the processor's instruction execution rate to understand and improve application performance. While we make the case for using internal hardware measurements of the processor's instruction execution rate to augment more conventional measures of CPU busy, we also acknowledge some of the current barriers that advocates of this approach encounter when they attempt to put it into practice today.

Sampling processor utilization.

The technique used to calculate processor utilization in Windows is based on gathering periodic samples of the processor's execution state. This legacy technique is characterized by low overhead, yielding measurements with a reasonable degree of accuracy over the kinds of time intervals that computer capacity planning, for example, requires. The sampling methodology was originally devised 20 years ago for Windows NT. Since one of the original design goals of Windows NT was to achieve a high degree of

hardware independence, the measurement methodology was also designed so that it was not dependent on any specific set of processor hardware measurement features.

The familiar % Processor Time counters in Perfmon are the measurements derived using this sampling technique. The measurement procedure uses an OS Scheduler periodic clock interrupt to sample the execution state of the processor once per interval. The periodic clock interrupt is a high priority, timer-based, hardware clock interrupt that is programmed to fire 64 times per second, once approximately every 15.6 ms. This clock interrupt is used to calibrate the system's Time of Day clock, which can then be retrieved by calling the GetSystemTime function.

The operating system's clock interrupt routine performs additional functions, in addition to updating the current system clock value. One of those other functions is CPU accounting, which is performed by recording the current execution state of each processor, immediately prior to the occurrence of clock interrupt. If the processor was running the Idle loop when the OS's periodic interrupt occurs, it is recorded as an Idle Time sample. If the processor was running some application thread, that is recorded as a CPU busy sample. Busy samples are then accumulated continuously at both the thread and process level. Since roughly 64 clock interrupts occur each second, the % Processor Time measurements are based on samples of the processor execution state gathered 64 times per second.[1]

This periodic sampling of the execution state of each processor is the source of the processor utilization measurements at the processor, process and thread level in both Perfmon and TaskMan, as well any number of Windows API calls that allow applications to retrieve that measurement data. Figure 1

[1] The periodic clock interrupt advances the system clock value by a 15.6 ms "tick." You can access the precise value that the OS uses between timer interrupts by calling the GetSystemTimeAdjustment() function.

This difference between the *precision* with which the system reports clock values and the actual *granularity* of system Time of Day clock value updates is the source of endless confusion in Windows. See, for example, Raymond Chen's "The Old New Thing" blog posting "Precision is not the same as accuracy" on this subject. Unfortunately, Chen's discussion of the issue on his popular Microsoft insider blog probably raises as many questions as it answers. The range of comments from his readers further illustrates some of the confusion around this topic.

Clock and timer values in Windows are reported in a standardized hh:mm:ss format, with fractional seconds reported to seven decimal digits. Thus, each logical "tick" of the system clock denotes 100 nanoseconds of elapsed time. However, if you write a program that spins in a loop, checking the value of the Windows Time of Day clock continuously, you will observe the clock value remains stationary until it is updated during the periodic clock interrupt. When your program resumes execution following the clock interrupt, you will then observe a clock "tick" of about 15.6 milliseconds added to the previous value of the system clock.

Another source of confusion is the bewildering array of system Time functions available in Windows. See the "Time Functions" article in the official MSDN library documentation for details.

A high resolution timer facility called QueryPerformanceCounter was introduced in Windows 2000. The names of the QueryPerformanceCounter and QueryPerformanceFrequency time functions that are used in obtaining high resolution clock values in Windows reflect their origin in solving the clock resolution problem specifically for the purpose of performance measurement. Unfortunately, the API names serve to obscure their key role in Windows in obtaining more precise measurements of elapsed time. Another source of confusion is that the official documentation for these API calls does not give developers an example of how to use them to obtain elapsed time measurements. Also, largely undocumented is the fact that since their original introduction, the implementation of the QueryPerformanceCounter and QueryPerformanceFrequency functions varies significantly from OS release to release, a topic that it will be necessary to return to later in this article.

illustrates the calculation of CPU time based on this sampling of the processor execution state as reported in the Performance tab of the Windows Task Manager.

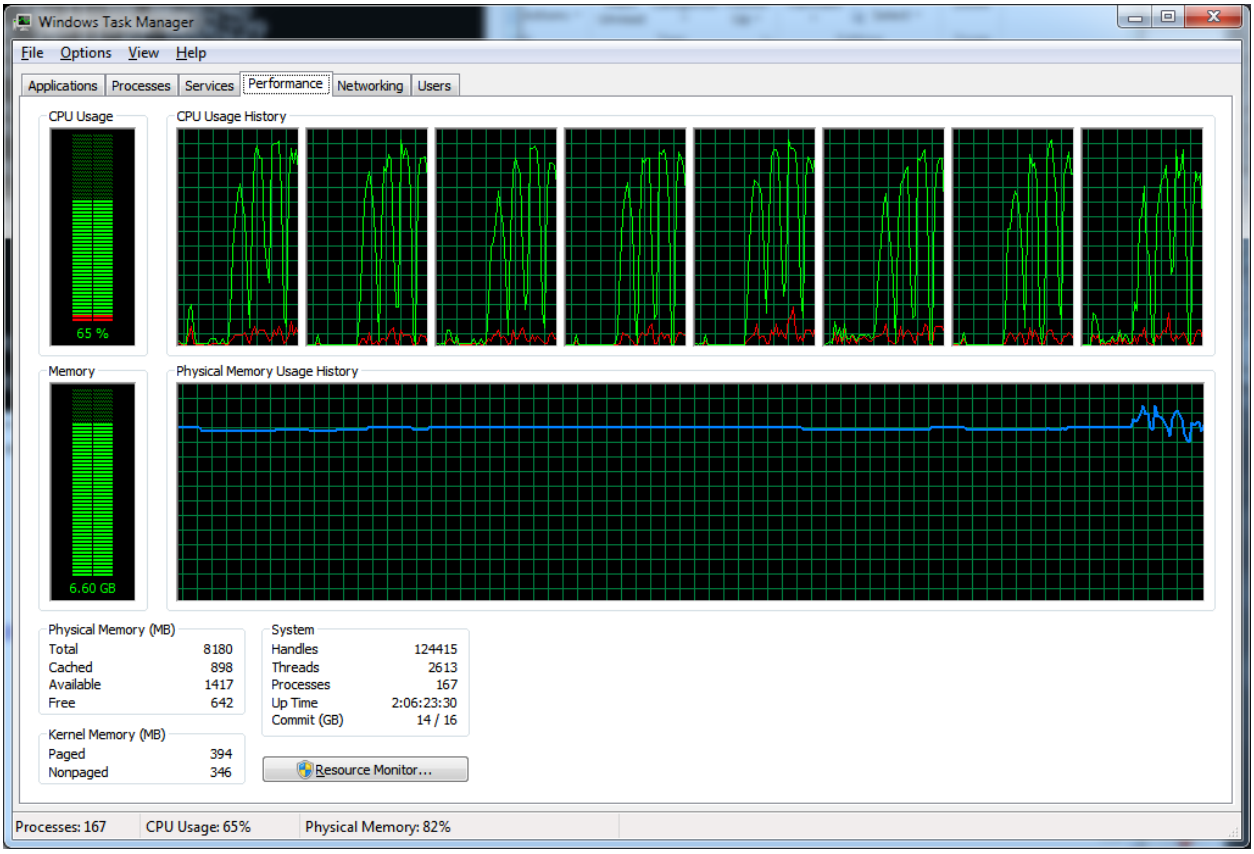


FIGURE 1. THE PERFORMANCE TAB OF THE WINDOWS TASK MANAGER REPORTS PROCESSOR UTILIZATION BASED ON SAMPLING THE PROCESSOR EXECUTION STATE ONCE EVERY *QUANTUM*, APPROXIMATELY 64 TIMES PER SECOND.

When the periodic clock interrupt occurs, the OS Scheduler performs various tasks, including adjusting the dispatching priority of threads that are currently executing with the intention of stopping the progress of any thread that has exceeded its time slice. Using the same high priority OS Scheduler clock interrupt that is used for CPU accounting to implement processor time-slicing is the reason the interval between Scheduler interrupts is often known as the *quantum*. At one time in Windows NT, the quantum between clock interrupts was set based on the speed of the processor; the faster the processor the shorter the quantum interval and the more frequently the OS Scheduler would gain control. Today, however, the quantum value is constant across processor hardware.

Another measurement function that is performed by the OS Scheduler's clock interrupt is to take a sample of the length of the processor Ready queue that contains threads that are queued for execution. The *System\Processor Queue Length* counter in Perfmon is an *instantaneous* counter that reflects the last measurement taken by the OS Scheduler's clock interrupt service routine of the current number of Ready threads waiting in the OS Scheduler queue. Thus, the *System\Processor Queue Length* counter represents a singleton observation, and needs to be interpreted with that in mind.

The processor Queue Length metric is sometimes subject to anomalies due to the kind of phased behavior you can often see on an otherwise idle system.[2] Even on a mostly idle Windows system, a sizable number of threads can be observed effectively waiting on the same clock interrupt (typically, waking up once per second to look for some changed state). When Perfmon is running, one of these periodically awaking threads happens to be the Perfmon measurement thread, often also set to cycle once per second. This situation is depicted in Figure 2a, showing the state of the machine at the time the OS Scheduler's periodic clock interval fires.

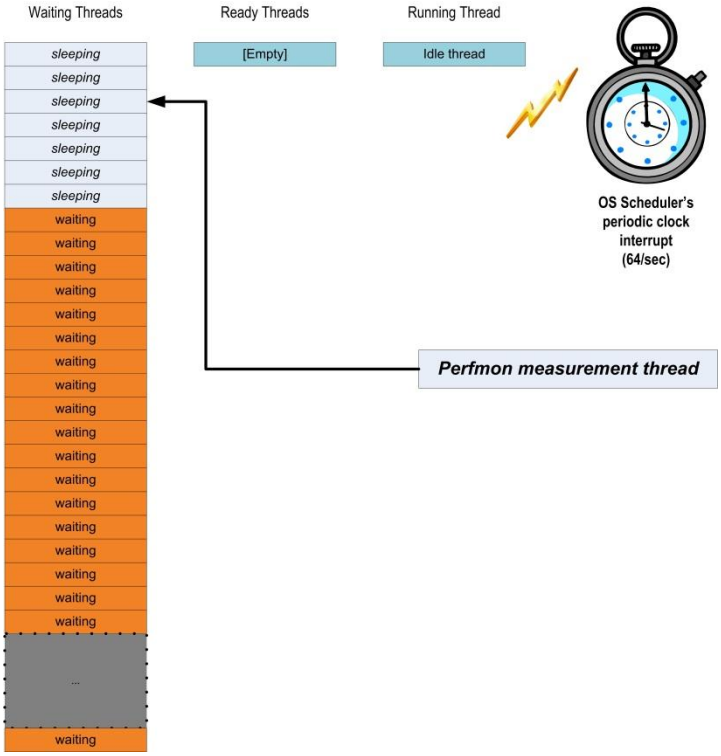


FIGURE 2A. PROCESSOR QUEUE LENGTH MEASUREMENTS IN WINDOWS ARE SUBJECT TO AN ANOMALY DUE TO THE "CLUMPING" BEHAVIOR OF THREADS WAITING ON TIMER INTERRUPTS OFTEN OBSERVED IN AN OTHERWISE IDLE MACHINE. A SIZABLE NUMBER OF THREADS CAN OFTEN BE FOUND WAITING ON THE SAME TIMER INTERRUPT. TYPICALLY, THESE ARE WORKER THREADS DESIGNED TO WAKE UP ONCE PER SECOND TO LOOK FOR SOME CHANGE IN MACHINE OR APPLICATION STATE. AS ILLUSTRATED, THE PERFMON MEASUREMENT THREAD, CYCLING ONCE PER SECOND, IS OFTEN ONE OF THESE SLEEPING THREADS. THE DRAWING DEPICTS THE OS SCHEDULER'S PERIODIC CLOCK INTERVAL FIRING, WHICH SERVES TO UPDATE THE SYSTEM CLOCK, WHICH WILL THEN WAKE UP ANY SLEEPING THREAD WHOSE SLEEP TIMER HAS EXPIRED.

When the clock interrupt updates the current Windows system clock value, the OS transitions any waiting threads whose elapsed sleep timer has expired to the Ready state, as depicted in Figure 2b. On an idle system, sleeping threads tend to clump together, such that a bunch of them are awakened by the same timer interrupt. These timer-activated threads wake up, discover rather quickly that the state change they are checking for has not occurred, and then quickly go back to sleep. The OS Scheduler's Ready queue is ordered by priority, so the high priority Perfmon measurement thread sorts to the top of the Ready queue, as illustrated in Figure 2b.

[2] These anomalies were first reported in a paper entitled "Interpreting Windows NT Processor Queue Length Measurements" by Ding, et. al., published in CMG *Proceedings*, 2002.

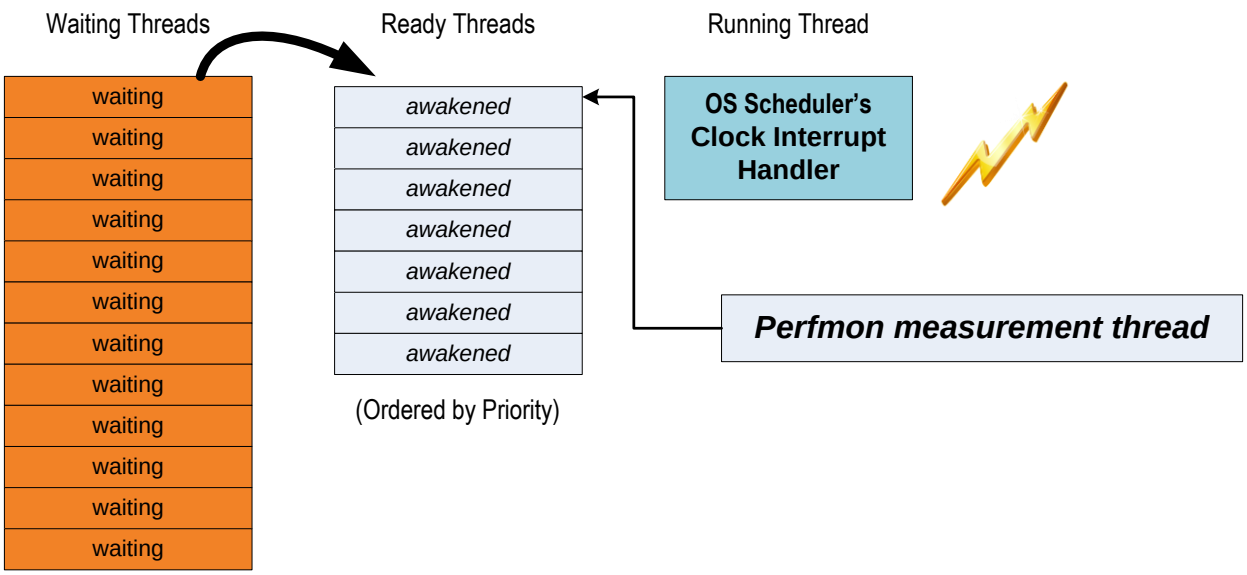


FIGURE 2B. AFTER THE OS SCHEDULER'S CLOCK INTERRUPT HANDLER UPDATES THE SYSTEM CLOCK VALUE, THE PERFMON MEASUREMENT THREAD TRANSITIONS TO THE READY STATE BECAUSE ITS SLEEP TIMER HAS EXPIRED. BECAUSE THE PERFMON MEASUREMENT THREAD EXECUTES AT A HIGH PRIORITY, IT SORTS TO THE TOP OF THE SCHEDULER'S QUEUE OF READY THREADS. AS ILLUSTRATED, THE CLOCK INTERRUPT MAY ALSO SERVE TO AWAKEN A NUMBER OF OTHER SLEEPING THREADS AT THE EXACT SAME TIME.

The Perfmon measurement thread executes at a high priority level, so it is scheduled for execution ahead of any other User mode threads that were also awakened by the same Scheduler clock tick, as illustrated in Figure 2b. When the clock interrupt handler completes its processing, including performing its CPU usage accounting functions, the Perfmon measurement thread is ready to execute next. The effect is that at the time the Processor ready queue length is measured, there are likely to be a disproportionately high number of Ready Threads, as depicted in Figure 2c.

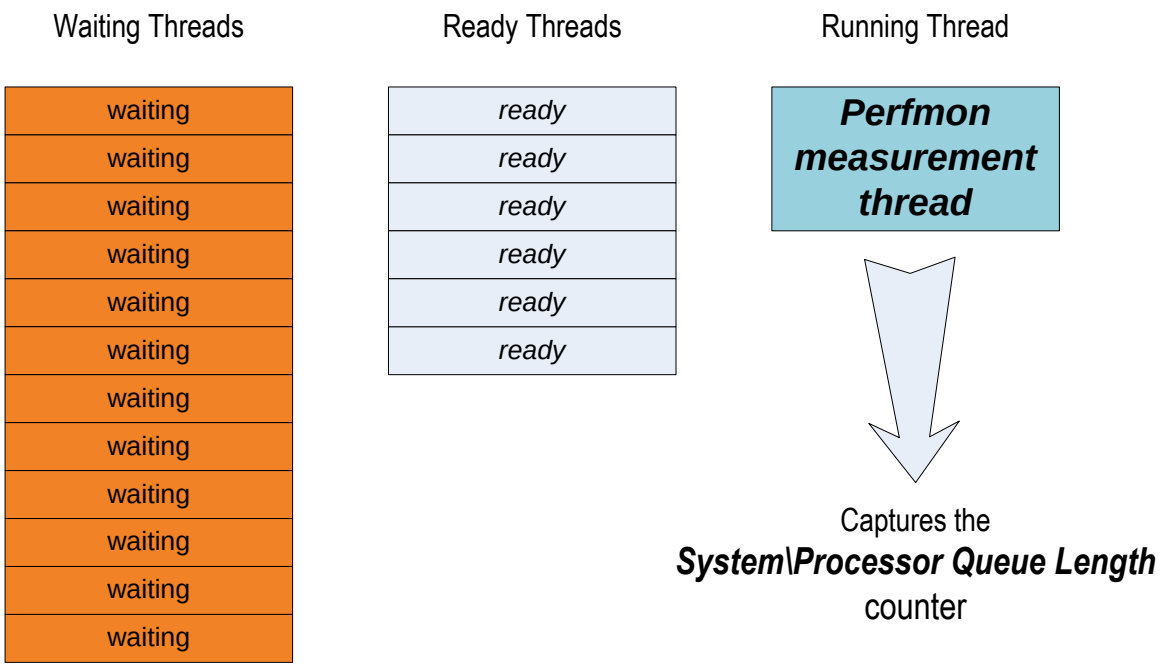


FIGURE 2C. WHEN THE CLOCK INTERRUPT HANDLER COMPLETES ITS PROCESSING, INCLUDING PERFORMING ITS CPU USAGE ACCOUNTING FUNCTIONS, THE PERFMON MEASUREMENT THREAD EXECUTES NEXT. IT CAPTURES A VALUE FOR THE SYSTEM\PROCESSOR QUEUE LENGTH COUNTER THAT IS DISPROPORTIONATELY HIGH DUE THE "CLUMPING" BEHAVIOR OBSERVED ON RELATIVELY IDLE MACHINES.

The result of this "clumping" behavior is that the periodic OS Scheduler interrupt that updates the system Time of Day clock, has a tendency to wake a bunch of sleeping threads up at the exact same time. The awakened threads then flood the OS dispatching queue. If one of these threads is the Perfmon measurement thread that is responsible for gathering the Processor Queue Length measurement, it sees an elongated queue. This "clumping" behavior can easily distort the measurements Perfmon gathers. Compared to the modeling assumption where processor scheduling is subject to random arrivals, one observes a disproportionate number of Ready Threads waiting for service, even (or especially) when the processor itself is not very busy overall.

This anomaly is best characterized as a *low-utilization effect* that perturbs the measurement when the machine is loafing. It generally ceases to be an issue when processor utilization climbs or there are more processors available on the machine. But this bunching of timer-based interrupts remains a serious concern, for instance, whenever Windows is running as a guest virtual machine under VMware or Hyper-V. Another interesting side discussion is how this clumping of timer-based interrupts interacts with power management, but I do not intend to venture further into that subject here.

Sampling. To summarize, the CPU utilization measurements at the system, process and thread level in Windows are based on a sampling methodology. Similarly, the processor queue length is also sampled. Like any sampling approach, the data gathered is subject to typical sampling errors, including

- accumulating a sufficient number of sample observations to be able to make a reliable statistical inference about the underlying population, and

- ensuring that there aren't sources of sampling error that causes sub-classes of the underlying population to be under or over-sampled systematically

So, these CPU measurements face familiar issues with regard to sampling size and the potential for systematic sampling bias, as well as the usual difficulty in ensuring that the sample data is actually *representative* of the underlying population (something known as *non-sampling error*). For example, the interpretation of the CPU utilization data that Perfmon gathers at the process and thread level is subject to limitations based on a small sample size for collection intervals less than, say, 15 seconds. At one minute intervals, there are enough samples to expect accuracy within 1-2%, a reasonable trade-off of precision against overhead. Over even longer measurement intervals, say 5 or 10 minutes, the current sampling approach leads to minimal sampling error, except in anomalous cases where there is some other source of systematic under-sampling of the processor's execution state.

Small sample size is also the reason that Windows does not currently permit Perfmon to gather performance data at intervals more frequent than once per second. Running performance data collection at intervals of 0.1 seconds, for example, the impact of relying on a very small number of processor execution state samples is quite evident. At 0.1 second intervals, processor times are calculated based on just 5 or 6 samples per interval. If you are running a micro-benchmark and want to access the same Thread\% Processor Time counters that Perfmon uses at 0.1 second intervals, you are looking for trouble. Under these circumstances, the % Processor Time measurements lose their resemblance to a continuous function over time.

An event-driven approach to measuring processor execution state.

The limitations of the legacy approach to measuring CPU busy in Windows and the need for more precise measurements of CPU utilization are recognized in many quarters across the Windows development organization at Microsoft. The legacy sampling approach is doubtless very efficient, and this measurement facility was deeply embedded in the OS kernel's Scheduler facility, a chunk of code that is very risky to tamper with. But, for example, more efficient power management, something that is crucial for battery-powered Windows devices, strongly argues for an event-driven alternative. You do not want the OS to wake up from a low power state regularly on an idle machine just to perform its CPU usage accounting duties, for example.

A straightforward alternative to periodically sampling the processor execution state is to measure the time spent in each processor state directly. This is accomplished by instrumenting the phase state transitions themselves. Processor state transitions in Windows are known as *context switches*. A context switch occurs in Windows whenever the processor switches the processor execution context to run a different thread. Processor state transitions also occur as a result of high priority Interrupt Service Routines (ISRs) gaining control following a device interrupt, as well as the Deferred Procedure Calls (DPCs) that ISRs schedule to complete the interrupt processing. By recording the time that each context switch occurs, it is possible to construct a complete and an accurate picture of CPU consumption.[3]

[3] See a two-part article in MSDN Magazine, entitled "Core OS Events in Windows 7," written by Insung Park and Alex Bendetovers and published beginning in September 2009. The authors are, respectively, the architect and lead developer of the

It helps to have a good, general understanding of thread scheduling in the OS in order to interpret this stream of events. Figure 3 is a diagram depicting the state machine associated with thread execution. At any point in time, a thread can be in only one of the three states indicated: Waiting, Ready, or Running. The state transition diagram shows the changes in execution state that can occur. A Waiting thread is usually waiting for some event to occur, perhaps a Wait timer to expire, an IO operation to complete, a mouse or keyboard click that signals user interaction with the application, or a synchronization event from another thread that indicates it is OK to continue processing.

A thread that is Ready to run is placed in the Dispatcher's Ready Queue, which is ordered by priority. When a processor becomes available, the OS Scheduler selects the highest priority thread on the Ready Queue and schedules it for execution on that processor. Once it is running, a thread remains in the Running state until it completes its execution cycle and transitions back to the Wait state. An executing thread can also be *interrupted* because a higher priority execution unit needs to run (this is known as *preemptive scheduling*) or it is interrupted by the OS Scheduler because its *time-slice* has expired. A Running thread can also be delayed because of a page fault, accessing data or an instruction in virtual memory that is not currently resident in physical memory. These thread execution time delays are often referred to as *involuntary waits*.

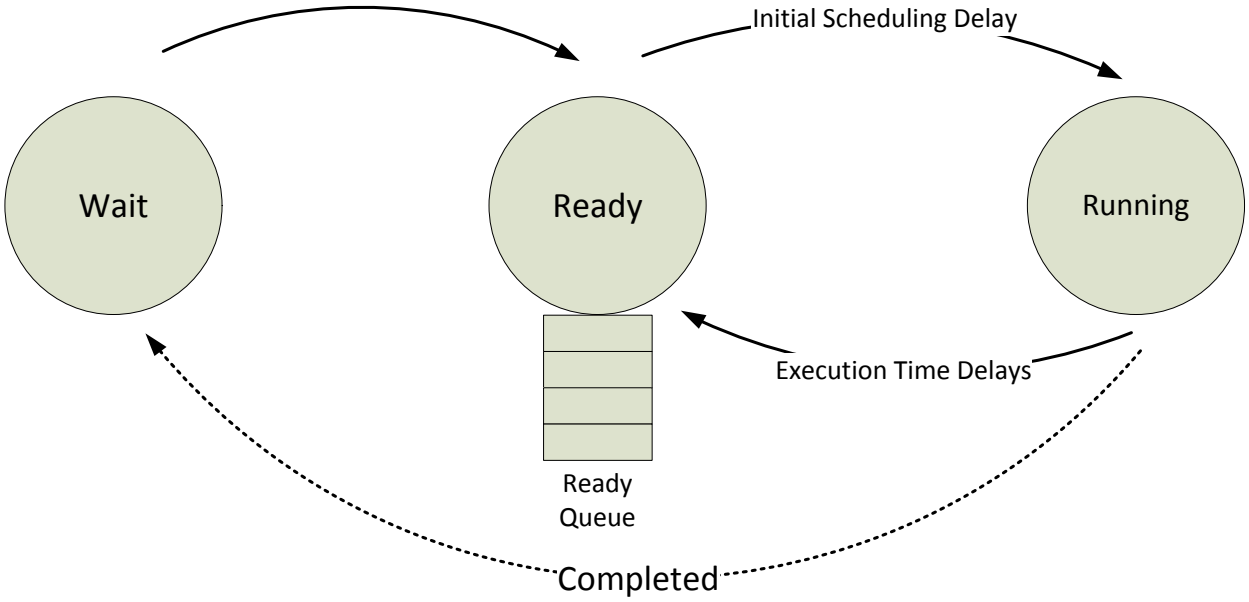


FIGURE 3. A STATE MACHINE FOR THREAD EXECUTION.

Figure 4 associates these thread execution state transitions with the ETW events that record when these transitions occur. The most important of these is the CSwitch event record that is written on every processor context switch. The CSwitch event record indicates the thread ID of the thread that is entering

ETW infrastructure. The article provides a conceptual overview describing how to use the various OS kernel events to reconstruct a state machine for processor execution, along with other diagnostic scenarios. Park and Bendetovers report, "In state machine construction, combining Context Switch, DPC and ISR events enables a very accurate accounting of CPU utilization."

the Running state (the new thread id), the thread ID that was displaced (the old thread ID), and provides the Wait Reason code associated with an old thread ID that is transitioning from Running back to the Wait state. The processor number indicating which logical CPU has undergone this state change is provided in an ETW_Buffer_Context structure associated with the ETW standard record header. Thread 0 from Process 0 indicates the Idle thread, which is dispatched on a processor whenever there are no Ready threads waiting for execution. While a thread other than the Idle thread is "active," the CPU is considered busy.

Conceptually, a context switch event adheres to a state **switch** pattern, with a time stamp identifying when the context switch occurred. The CPU time of a thread is the amount of time it spends in the Running state. It is measured using the CSwitch events that show the thread transitioning from Ready to the Running state and the CSwitch events that show that thread transitioning back from the Running state to Waiting. To calculate processor busy, you summarize the amount of time each processor spends when the Idle thread is active and subtract from 100% over the measurement interval.

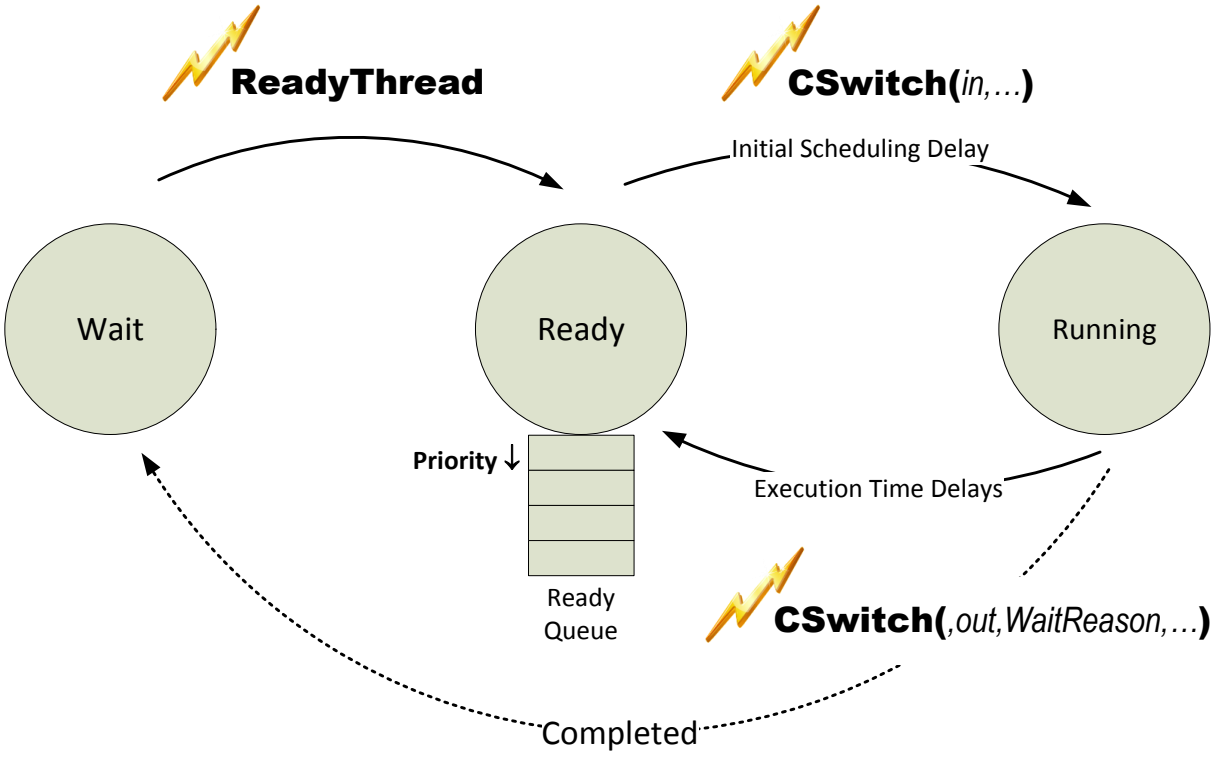


FIGURE 4. THE STATE TRANSITION DIAGRAM FOR THREAD EXECUTION, INDICATING THE ETW TRACE EVENTS THAT MARK THREAD STATE TRANSITIONS.

One complication in this approach is that the ETW infrastructure does not guarantee delivery of every event to a Listener application. If the Listener application cannot keep up with the stream of events, then ETW will drop memory-resident buffers filled with events rather than queue them for delivery later. CSwitch events can occur at very high rates, 20,000-40,000 times per second per CPU are not unusual on busy machines, so there is definitely potential to miss enough of the context switch events to bias the calculations that result. In practice, handling the events efficiently in the ETW Listener

application and making appropriate adjustments to the ETW record buffering options can be used to minimize the potential for missing events.

To see this event-driven processor execution state measurement facility at work, access the Resource Monitor application (resmon.exe) that is available beginning in Vista and Windows Server 2008. Resource Monitor can be launched directly from the command line, or from either Performance Monitor plug-in or Task Manager Performance tab. Figure 5 displays a screen shot that shows Resource Monitor in action on a Windows 7 machine, calculating CPU utilization over the last 60 seconds of operation, breaking out that utilization by process. The CPU utilization measurements that ResMon calculates are based on the context switch events. These measurements are very accurate, about as good as it gets from a vantage point inside the OS.

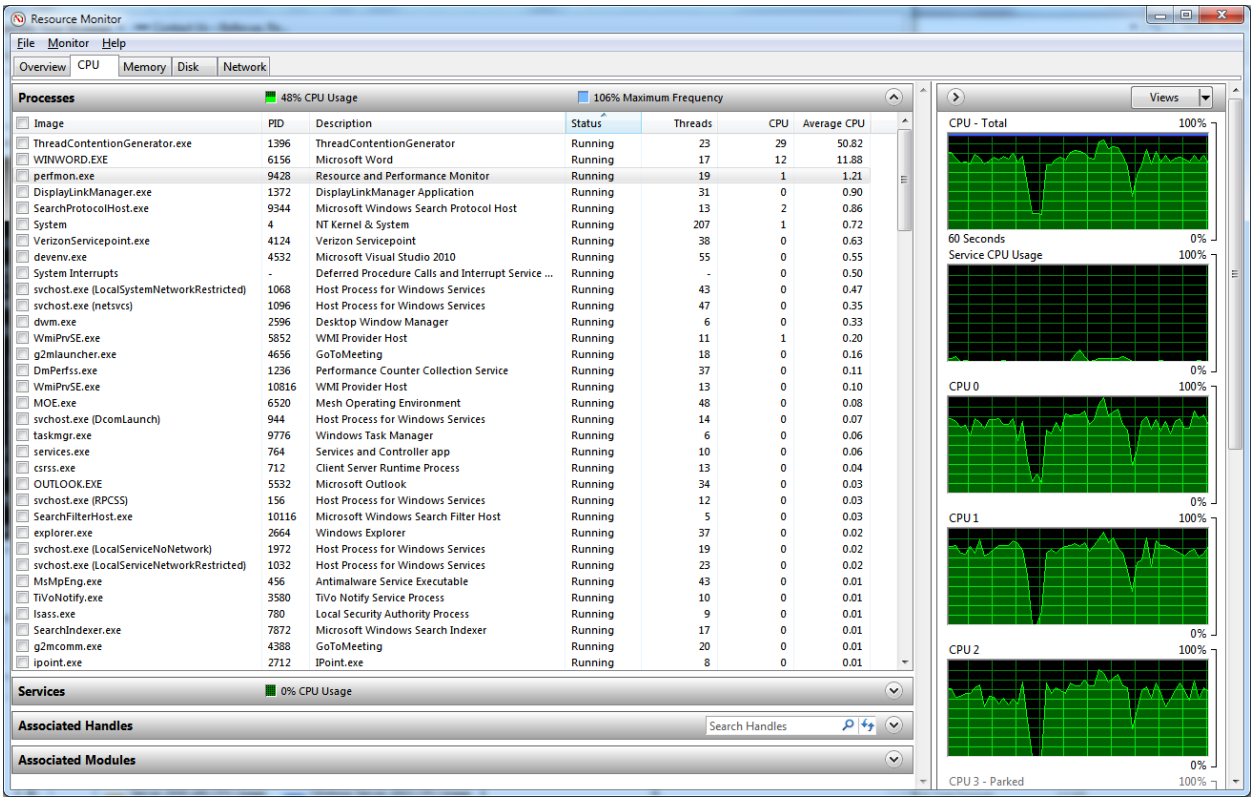


FIGURE 5. THE WINDOWS 7 RESOURCE MANAGER APPLICATION.

The Resource Monitor measures CPU busy in real time by listening to the ETW event stream that generates an event every time a context switch occurs. It also produces similar reports from memory, disk, and network events.

To summarize these developments, this trace-driven measurement source positions the Windows OS so it could replace its legacy CPU measurement facility with something more reliable and accurate sometime in the near future. Unfortunately, converting all existing features in Windows, including Perfmon and Task Manager, to support the new measurements is a big job, not without its complications and not always as straightforward as one would hope. But we anticipate future versions

of the Windows OS will adopt an accurate, event-driven approach to measuring processor utilization, ultimately replacing the legacy sampling approach that Task Manager and Perfmon rely on today.

Using xperf to analyze CSwitch events

The same CPU busy calculations that the Resource Manager in Windows 7 makes can also be performed after the fact using the event data from ETW. This is the technique used in the Windows Performance Toolkit (WPT, but which is better known around Microsoft as xperf), for example, to calculate CPU usage metrics.

Once you have downloaded and installed the Windows Performance Toolkit, you can launch a basic ETW collection session using the following xperf command:

```
xperf -on DiagEasy
```

Then, after you have accumulated enough data, issue another command to stop tracing and capture the event stream to a file:

```
xperf -d cputrace.etl
```

Next, process the cputrace.etl file using the xperfview app. After the trace file is loaded, xperfview provides visualizations that are very similar to ResMon. See Figure 6 for an example.

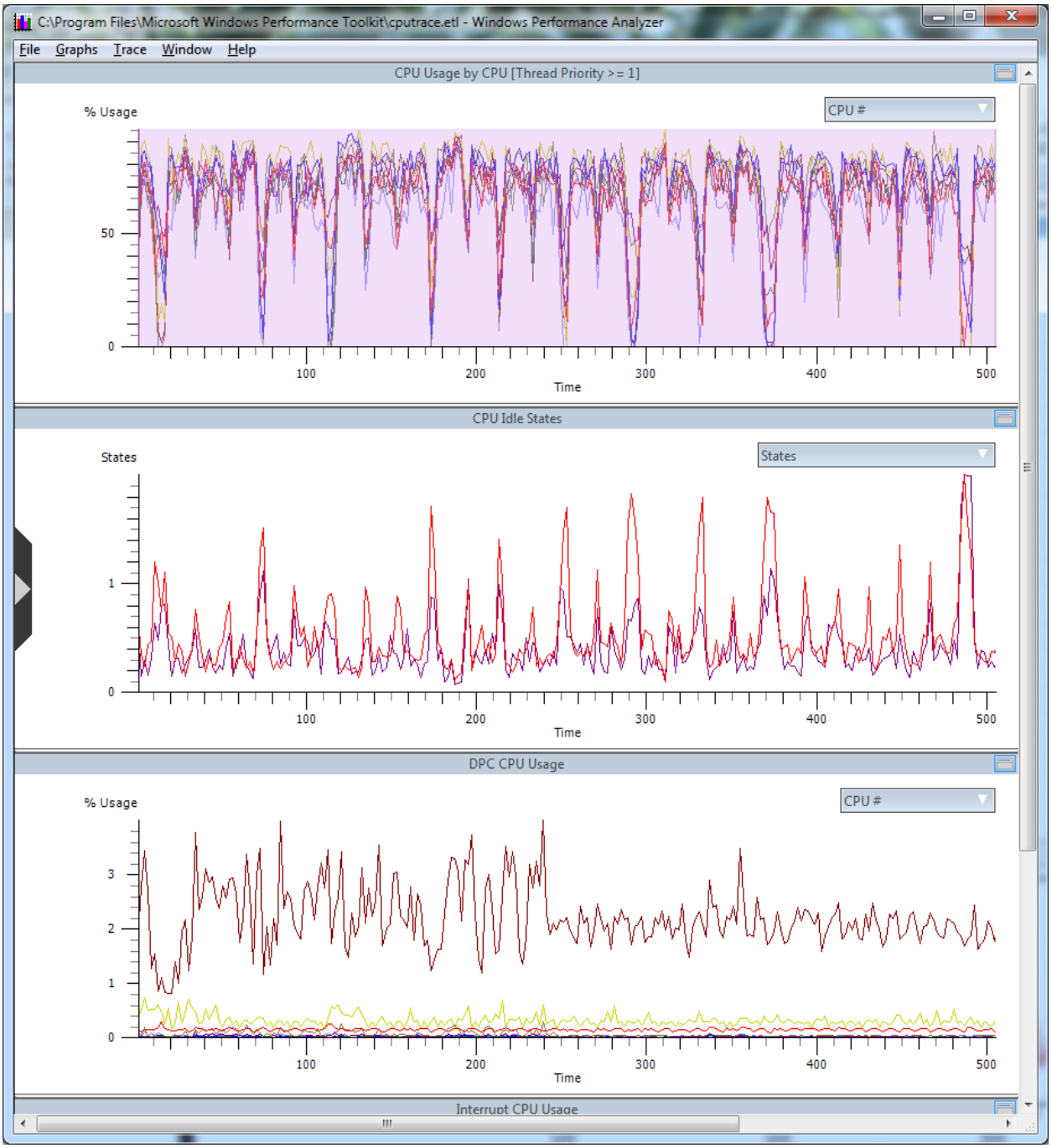
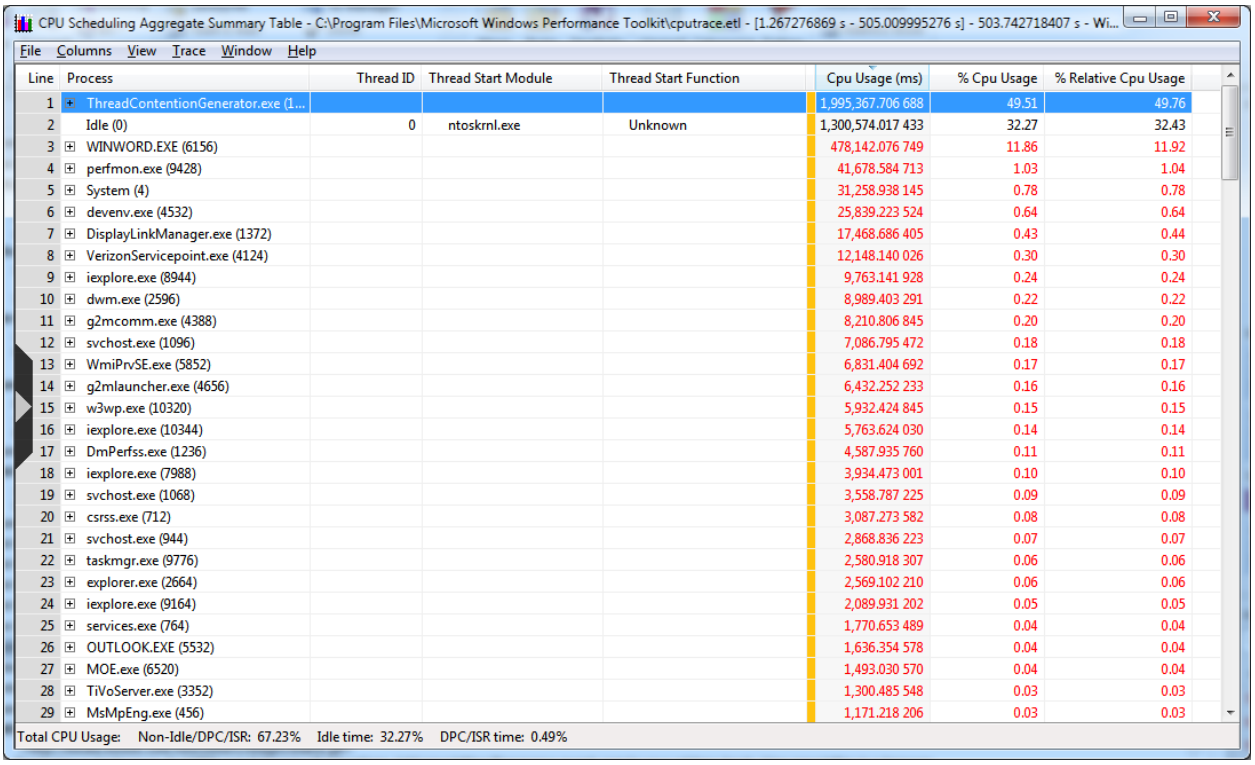


FIGURE 6. CPU UTILIZATION GRAPHS IN XPERFVIEW, BASED ON ETW CONTEXT SWITCH EVENT DATA GATHERED WITH THE XPERF UTILITY. ADDITIONAL ISR AND DPC EVENTS ARE USED TO CALCULATE THE AMOUNT OF TIME DEVICE DRIVERS SPEND PROCESSING INTERRUPTS.

Figure 6 illustrates several of the CPU utilization graphs that xperfview creates from the context switch event stream. To help make the graph more readable, I filtered out Idle time calculation for all but two of the logical processors on this machine. (The machine illustrated has 8 logical CPUs.) To gain insight into what high priority Interrupt Service Routines (ISRs) and DPCs are running, ISR and DPC events should also be gathered, which the DiagEasy event profile in xperf does automatically. (Windows device driver developers are very interested in them. For an example, see this blog posting discussing using the xperf ETW utility to capture CPU consumption by the TCP/IP network driver stack.)

With xperfview, you can also request a Summary Table which displays CPU usage by process (and thread) by right-clicking on the CPU Usage graph and accessing the pop-up context menu. An example of the CPU Scheduling Aggregate Summary Table is illustrated in Figure 6. It is similar to the one ResMon produces. The data here was gathered while running a multi-threaded CPU soaker program called ThreadContentionGenerator while ResMon was also active. You can see that the calculation in Figure 7 roughly mirrors the results shown in Figure 5 for ResMon, allowing for some variation that is to be expected since the intervals themselves are not identical. The xperf interval shown in Figure 6 is approximately 500 seconds long, while ResMon maintains a rolling window that is only 60 seconds in duration. The ResMon screen shot was taken somewhere in the middle of the longer xperf tracing session.



Line	Process	Thread ID	Thread Start Module	Thread Start Function	Cpu Usage (ms)	% Cpu Usage	% Relative Cpu Usage
1	ThreadContentionGenerator.exe (1...				1,995,367.706 688	49.51	49.76
2	Idle (0)	0	ntoskrnl.exe	Unknown	1,300,574.017 433	32.27	32.43
3	WINWORD.EXE (6156)				478,142.076 749	11.86	11.92
4	perfmon.exe (9428)				41,678.584 713	1.03	1.04
5	System (4)				31,258.938 145	0.78	0.78
6	devenv.exe (4532)				25,839.223 524	0.64	0.64
7	DisplayLinkManager.exe (1372)				17,468.686 405	0.43	0.44
8	VerizonServicepoint.exe (4124)				12,148.140 026	0.30	0.30
9	iexplore.exe (8944)				9,763.141 928	0.24	0.24
10	dwm.exe (2596)				8,989.403 291	0.22	0.22
11	g2mcomm.exe (4388)				8,210.806 845	0.20	0.20
12	svchost.exe (1096)				7,086.795 472	0.18	0.18
13	WmiPrvSE.exe (5852)				6,831.404 692	0.17	0.17
14	g2mlauncher.exe (4656)				6,432.252 233	0.16	0.16
15	w3wp.exe (10320)				5,932.424 845	0.15	0.15
16	iexplore.exe (10344)				5,763.624 030	0.14	0.14
17	DmPerfss.exe (1236)				4,587.935 760	0.11	0.11
18	iexplore.exe (7988)				3,934.473 001	0.10	0.10
19	svchost.exe (1068)				3,558.787 225	0.09	0.09
20	csrss.exe (712)				3,087.273 582	0.08	0.08
21	svchost.exe (944)				2,868.836 223	0.07	0.07
22	taskmgr.exe (9776)				2,580.918 307	0.06	0.06
23	explorer.exe (2664)				2,569.102 210	0.06	0.06
24	iexplore.exe (9164)				2,089.931 202	0.05	0.05
25	services.exe (764)				1,770.653 489	0.04	0.04
26	OUTLOOK.EXE (5532)				1,636.354 578	0.04	0.04
27	MOE.exe (6520)				1,493.030 570	0.04	0.04
28	TiVoServer.exe (3352)				1,300.485 548	0.03	0.03
29	MsMpEng.exe (456)				1,171.218 206	0.03	0.03

FIGURE 7. THE CPU SCHEDULING AGGREGATE SUMMARY TABLE CALCULATED BY XPERFVIEW. THE RESULTS OF THESE CALCULATIONS CLOSELY RESEMBLES THE ROLLING ONE-MINUTE CALCULATION REPORTED BY THE RESOURCE MONITOR IN FIGURE 5.

For some perspective on the volume of trace events that can be generated, the binary .etl trace file produced in this example was approximately 325 MB on disk for a DiagEasy trace session that ran for more than ten minutes. Running with the xperf defaults, I received a notification when the trace session closed that three ETW 64K buffers of data were dropped during the trace because xperf was unable to keep pace with processing the event stream in real-time.

The Context Switch event also provides the old thread's Wait Reason code, which helps you to understand why the sequence of thread scheduling events occurred. For reference, a Windows context switch is defined here, while the contents of the ETW (Event Tracing for Windows) context switch event record are defined here, including a list of the current thread WaitReason codes.

Note that you can measure CPU queue time accurately from the ETW events, an important indicator of processor contention when the processor becomes saturated. As illustrated in Figure 4, the transition from the Wait state to the Ready state is marked by a ReadyThread event record. The time between the ReadyThread event and a subsequent CSwitch event marking its transition to Running is one form of CPU Queue time. A second form of CPU queue time is the time between a **CSwitch**(*…,out,WaitReason,…*) where the WaitReason is either a Preempt or time-slice quantum expiration and a subsequent re-dispatch. Both forms of CPU queue time can be measured accurately using ETW.

When precision in understanding patterns of CPU consumption is required, post-processing the ETW context switching event stream is a much better way to proceed than attempting to use the Windows % Processor Time performance counters. Measuring CPU consumption from the context switch events is considerably more precise than, for example, the Windows performance counter data available in Perfmon that report processor utilization at the system and process level based on processor state sampling. Such high precision measurements are not always required, of course, and processing the ETW context switching event stream is relatively expensive due to the extremely high volume of trace data that you must deal with.

Measuring thread execution state.

Besides measuring processor utilization at the system level, the stream of context switch events can also be re-constructed to drill into CPU consumption at the process and thread level. An exemplary example of this approach is the Visual Studio Profiler's Concurrency Visualizer, available in Visual Studio 2010. (For reference, see "Performance Tuning with the Concurrency Visualizer in Visual Studio 2010 in the Visual Studio 2010 Profiler," an MSDN Magazine article written by the tool's principal architect, Hazim Shafi.) The Concurrency Visualizer gathers Context Switch events to calculate processor utilization for the application being profiled.

The VS Concurrency Visualizer creates a system-level CPU Utilization View with an interesting twist – the view pivots based on the application you are profiling, a perspective that matches that of a software performance engineer engaged in a performance investigation. Based on the sequence of context switch trace events, the Concurrency Visualizer calculates processor utilization by the process, aggregates it for the current selection window, and displays it in the CPU Utilization View. In the CPU Utilization View, all other processor activity for processes (other than one being profiled) is lumped together under a category called "Other Processes." System-processes and the "Idle process," which is a bookkeeping mechanism, not an actual process that is dispatched, are also broken out separately. See Dr. Shafi's article for more details. (For reference, Figure 12 below illustrates the CPU Utilization View.)

The Concurrency Visualizer's primary focus is on being able to reconstruct the sequence of events that impact an application's execution progress. The Concurrency Visualizer's Threads View is the main display showing an application's execution path. The view here is of execution progress on a thread by thread basis. For each thread in your application, the Concurrency Visualizer shows the precise sequence of context switch events that occurred. These OS Scheduler events reflect that thread's execution state over time. See Figure 8 for an example of this view.

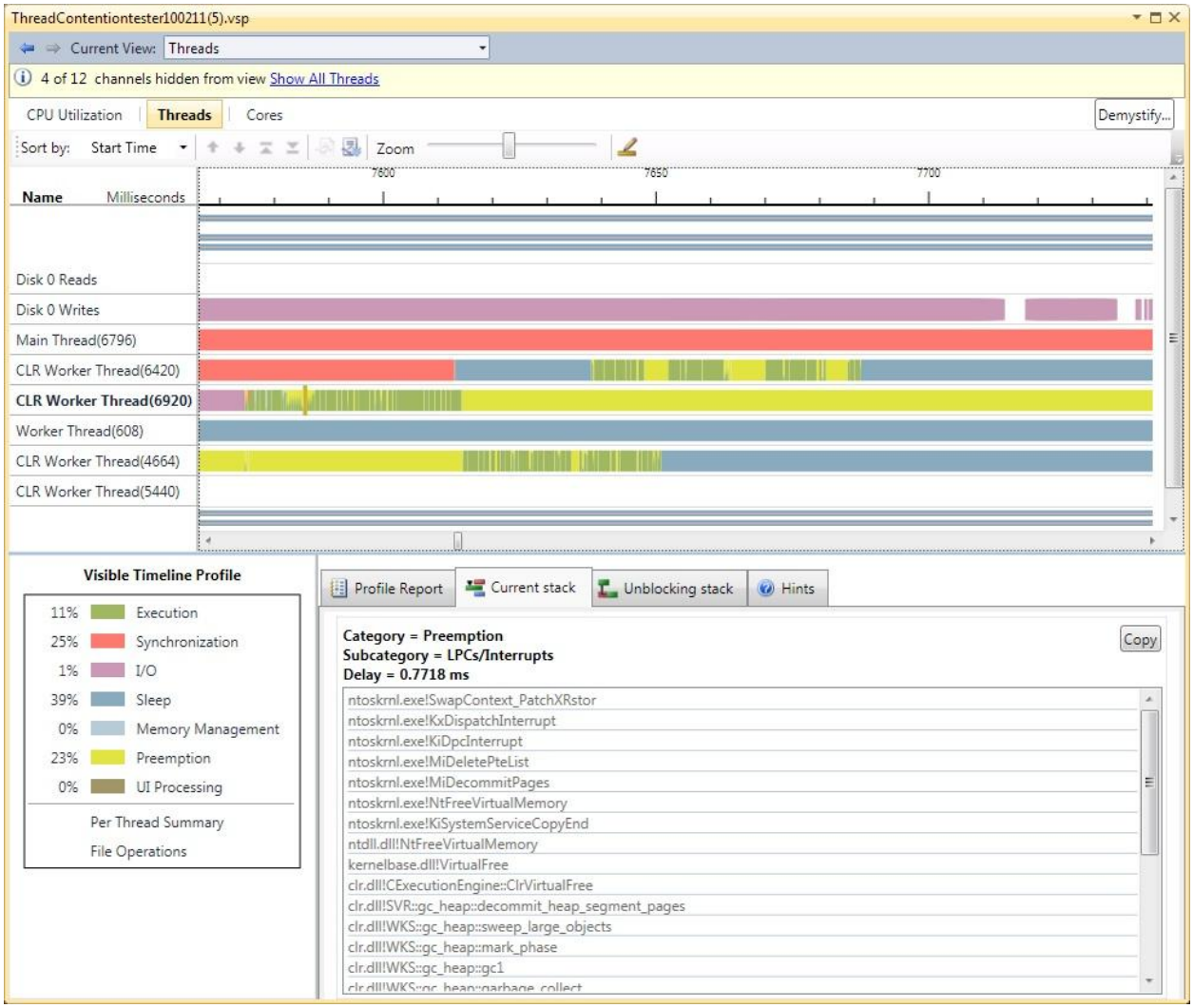


FIGURE 8. SCREEN SHOT OF THE CONCURRENCY VISUALIZER ILLUSTRATING THREAD PREEMPTION BY A HIGHER PRIORITY SYSTEM ROUTINE.

Figure 8 shows the execution path of six application threads, a Main thread, a generic worker thread, and 4 CLR (the Common Language Runtime for .NET languages) worker threads that the application created by instantiating a .NET ThreadPool object. (There were originally more threads than this, but I chose to hide those that were inactive over the entire run.) For each thread, the execution state of the thread – whether it is running or whether it is blocked – is indicated over time.

The upper half of the display is a timeline that shows the execution state of each thread over time. The execution progress of each thread display is constructed horizontally from left to right from rectangles that indicate the start and end of a particular thread state. An interval when the thread was running shows as green. An interval where the thread is sleeping is shown in blue. A ready thread that is blocked from executing because a higher priority thread is running is shown in yellow. (This state is labeled "preemption.") A thread in a synchronization delay waiting on a lock is visualized as red.

On the lower left of the display is a Visible Timeline Profile. This summarizes the state of all threads that are visible within the selected time window. In the screen shot in Figure 8, I have zoomed into a time

window that is approximately 150 milliseconds wide. During that interval, the threads shown were in a state where they were actively executing instruction only 11% of the time. For 25% of the time interval, threads were blocked waiting on a lock. Finally, there is a tabbed display at the lower right. If you click on the "Profile Report" tab, a histogram displays that summarizes the execution state of each individual thread over the time window. In the screen shot, I have clicked on the "Current stack" tab that displays the call stack associated with the ETW context switch event. If the thread is blocked, the call stack indicates where in the code the thread will resume execution once it unblocks. We will drill into that call stack in a moment.

Note: The Threads View also displays call stacks from processor utilization samples that ETW gathers on a system-wide basis once per millisecond. Call-stacks samples are visible during any periods when the thread is executing instructions (and ETW execution sampling is active).[4]

The Concurrency Visualizer screen shot in Figure 8 illustrates the calculation of a running thread's CPU queuing delay. Thread 6920, which happens to be a CLR thread pool worker thread, is shown at a point in time where it was preempted by a higher priority task. The specific delay that I zoomed in on in the screen shot is preemption due to the scheduling of a high priority LPC or ISR – note this category in the Concurrency Visualizer also encompasses assorted APCs and DPCs. In this specific example, execution of Thread 6920 was delayed for 0.7718 milliseconds. According to the trace, that is the amount of time between Thread 6920 being preempted by a high priority system routine and a subsequent context switch when the ready thread was again re-dispatched.

The tool also displays the call stack of the preempted thread. The call stack indicates that the CLR's garbage collector (GC) was running at the time that thread execution was preempted. From the call stack, it looks like the GC is sweeping the Large Object Heap (LOH), trying to free up some previously allocated virtual memory. This is not an opportune time to get preempted. You can see that one of the other CLR worker threads, Thread 6420, is also delayed. Notice from the color coding that Thread 6420 is delayed waiting on a lock. Presumably, one of the other active CLR worker threads in the parent process holds the lock that Thread 6420 is waiting for.

This is one of those "Aha" moments. If you click on the synchronization delay that Thread 6420 is experiencing, as illustrated in Figure 9, you can see that the lock that Thread 6420 is trying to acquire is, in fact, currently held by Thread 6920, the one that was preempted somewhere in the midst of running garbage collection. Clicking on the tab that says "Current Stack" (not illustrated) indicates that the duration of the synchronization delay that Thread 6420 suffered in this specific instance of lock contention was about 250 milliseconds.

[4] One of the ETW OS kernel events that the Concurrency Visualizer does not analyze is the ReadyThread event. The interval between a ReadyThread event and a subsequent Context Switch that signals that a ready Thread is being dispatched measures CPU queue time delay directly. Using event data, it is possible to measure CPU queuing delays precisely. Analysis of the ETW kernel event stream far exceeds anything that can be done using Windows performance counters to try to estimate the impact of CPU queuing delays.

The scenario here shows one CLR worker thread blocked on a lock that is held by another CLR worker thread, which in turn finds itself being delayed due to preemptions from higher priority Interrupt processing. We can see that whatever high priority work preempted Thread 6920 has the side effect of also delaying Thread 6420, since 6420 was waiting on a lock that Thread 6920 happened to be holding at the time. The tool in Figure 9 displays the Unblocking stack from Thread 6920 which shows the original memory allocation from the Dictionary.Resize() method call being satisfied, releasing a global GC lock. When Thread 6920 resumed execution following its preemption, the GC operation completes, releasing the global GC lock. Thread 6920 continues to execute for another 25 microseconds or so, before it is preempted because its time slice expired. Even as Thread 6920 blocks, Thread 6420 continues to wait while a different CLR thread pool thread (4664) begins to execute instead. Finally, after another 25 microseconds delay, Thread 6420 resumes execution. For a brief period both 6420 and 4664 execute in parallel from approximately the 7640 to 7650 microsecond milestones. (However, they are subject to frequent preemptions during that period of overlapped execution.)[5]

[5] Welcome to the *indeterminacy* associated with parallel programming.

I won't take the time here to go into what this little concurrent CLR (Common Language Runtime) thread pool application is doing. Suffice to say that it instantiates and references a very large Dictionary object in .NET, and I wrote it to illustrate some of the performance issues developers can face trying to do parallel programming, which is a topic I was blogging about at the time. (I should also note that the test program puts the worker threads to sleep periodically to simulate synchronous I/O waits to create an execution profile similar to what one could expect in processing a typical ASP.NET web request that needs to access an external database, an excellent idea I appropriated from a colleague, Joe Hellerstein.)

When I first began to profile this test app using the VS Concurrency Visualizer, I was able to see blocking issues like the one described here where the CLR introduced synchronization and locking considerations that are otherwise opaque to the developer. Well, *caveat emptor*, I suppose, when it comes to utilizing someone else's code framework in your application. (See Rico Mariani's Performance Tidbits blog for a singular discussion of his intriguing proposal that a .NET Framework method provide a *performance signature* that would allow a developer to make an informed decision before ever calling into some 3rd party's code. Alas, static code analysis cannot be used to predict the performance of some arbitrarily complex method call embedded in your application, something Rico was eventually forced to concede.)

It turns out that .NET Framework collection classes do use locks to ensure *thread-safe* operation in a multi-threaded program, whether it is necessary or not. See the MSDN "Thread-Safe Collections" Help topic for more information. Each worker thread in my test program instantiated and accessed a dedicated instance of the Dictionary class during processing, so locking wasn't necessary in this little test application. Because I had taken steps to ensure thread-safety issues would never arise in my test program, I was unpleasantly surprised when the tool uncovered lock contention for these Dictionary objects. Unfortunately, there is no way for the developer to explicitly signal the runtime that locking is not necessary. Some of the popular .NET Framework collection classes – like the HashTable – do provide a Synchronized method that exposes a lock created implicitly. But the Synchronized method is designed to support more complex multi-threaded access patterns, such as a multiple readers and writers scenario, for example. To assist in parallel programming tasks, several newer collection classes were introduced in the System.Collections.Concurrent Namespace that use "lock-free" and optimistic locking approaches that promise better scalability for parallel programs.

I eventually tweaked the test app into an especially ghoulish version I call the LockNestMonster program that uses explicit global locks to shine an even brighter light on these issues.

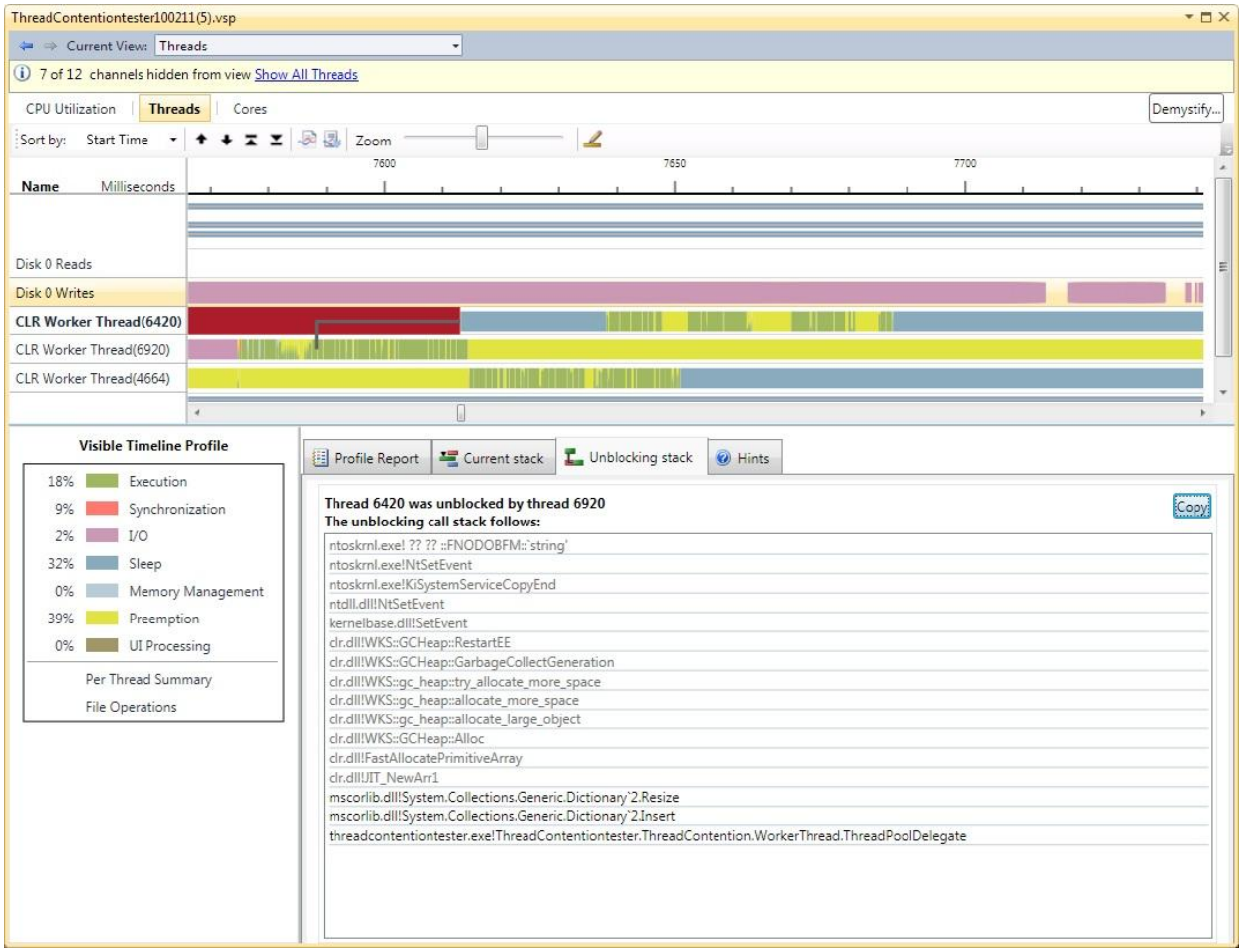


FIGURE 9. CLR WORKER THREAD 6420 BLOCKED BECAUSE IT IS WAITING ON A GC LOCK THAT HAPPENS TO BE HELD BY THREAD 6920, WHICH IS SUBJECT TO PREEMPTION BY HIGHER PRIORITY SYSTEM ROUTINES.

Time-slicing.

The Concurrency Visualizer also utilizes context switch events to calculate the delays a thread encounters during execution due to *preemption*, as a result of the expiration of a thread's time-slice.[6] In Figure 10, I clicked on the large yellow block on the right hand side of the execution time bar graph for Thread 6920 indicating another long delay. As in Figure 9, I have hidden all but the three active CLR thread pool threads. Using a combination of zooming to a point of interest in the event stream and filtering out extraneous threads, as illustrated in Figure 10, the Concurrency Visualizer is able to construct an execution time profile using just those events that are visible in the current time-window.

[6] The duration of the OS Scheduler time slice being one of the few tuning adjustments available in the OS. For the record, I normally recommend that system administrators not fiddle with this tuning knob, unless they lots of extra time on their hands. This older KB article provides some flavor for what is involved. For someone that cannot resist the temptation to fiddle with the time-slicing tuning parameters, the Concurrency Visualizer Threads View is the first Windows performance tool that can help you determine if changing the OS default time-slice value is doing your application any good, or harm, for that matter.

Overall, the three active CLR worker threads are only able to execute 18% of the time, while they are delayed by synchronization 9% of the time and subject to preemption 39% of the time. (You can click on the Profile Report tab in the middle right portion of the display and see a profile report by thread.)

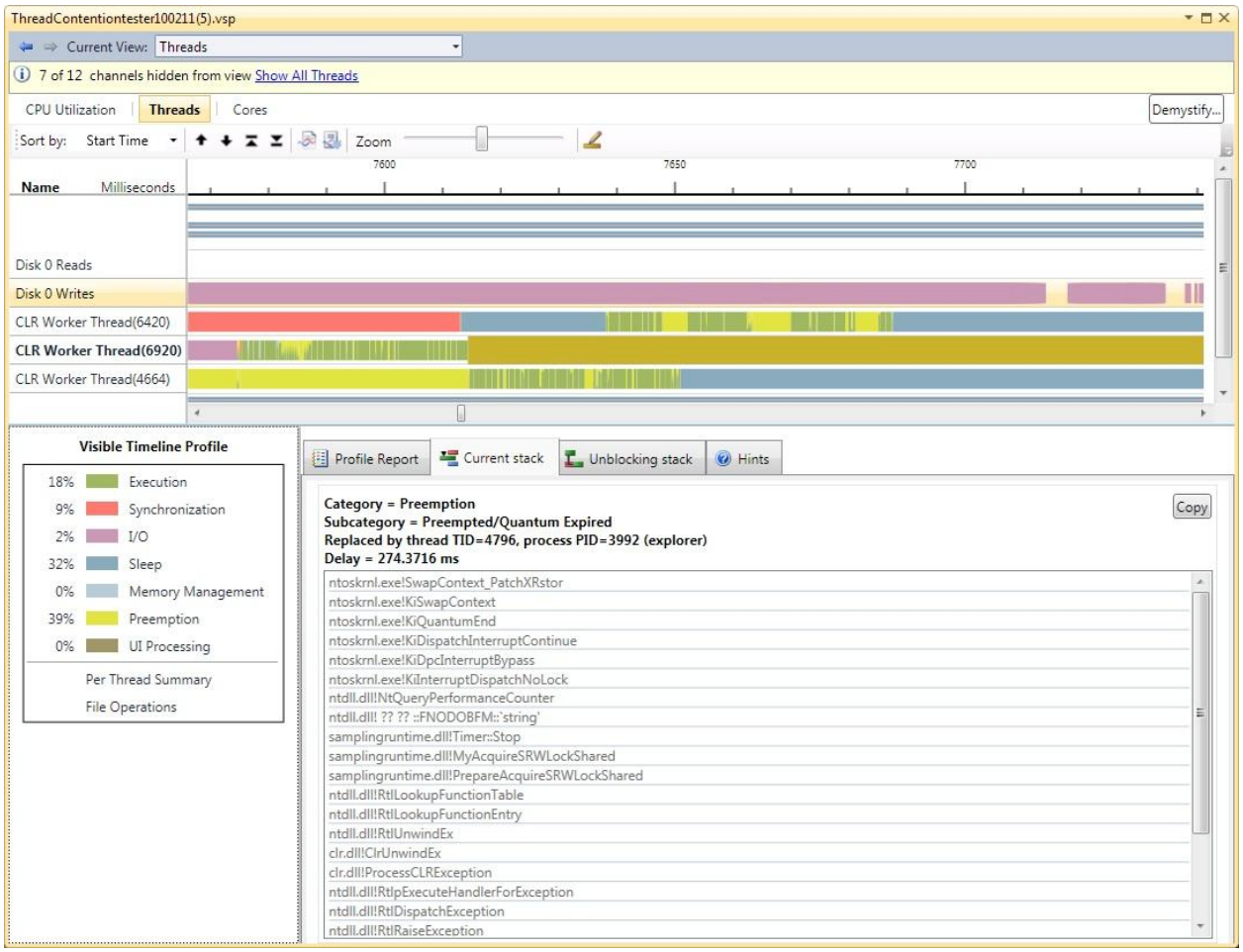


FIGURE 10. USING THE CONCURRENCY VISUALIZER TO DRILL INTO THREAD PREEMPTION DELAYS.

At the point indicated by the selection, the time-slice quantum for Thread 6920 expired and the Scheduler preempted the executing thread in favor of some other ready thread. Looking at the visualization, it should be apparent that the ready thread the Scheduler chose to execute next was another CLR thread pool worker thread, namely Thread 4664, which then blocked Thread 6920 from continuing. The tool reports that a *context switch*(6920, 4664) occurred, and that Thread 6920 was delayed for about 275 milliseconds before it resumed execution after being preempted.

As illustrated in this example, the Concurrency Visualizer uses the ETW-based event data from a profiling run to construct a state machine that reflects the precise execution state of each application thread over the time interval being monitored. It goes considerably beyond calculating processor queue time at the thread level. It understands how to weave the sequence of Ready Thread and Context switch events together to create this execution time profile. It summarizes the profiling data, calculating the precise amount time of time each thread is delayed by synchronous IO, page faults (i.e., involuntary waits due

to memory management overhead[7]), processor contention, preemption by higher priority work, and lock contention over the profiling interval. Furthermore, it analyzes the call stacks gathered at each Context Switch event, looking for signatures that identify the specific blocking reason. And, specifically, to help with lock contention issues, which are otherwise often very difficult to identify, it also identifies the thread that ultimately unblocks the thread that was found waiting to acquire a lock.

Application monitoring using the Scenario instrumentation library.

Within the discipline of software performance engineering (SPE), application response time monitoring refers to the capability of instrumenting application requests, transactions and other vital interaction scenarios in order to measure their response times. In addition, performance engineers usually want to be able to break down application response time into its component parts, one of which is CPU usage. Other than the Concurrency Visualizer that is packaged with the Visual Studio Profiler that was discussed above, there are few professional-grade, application response time monitoring and profiling tools that exploit the ETW facilities in Windows. As we have seen illustrated, the Concurrency Visualizer demonstrates the capability to reconstruct an application's execution time profile from kernel trace events with great precision and detail. An application response time monitoring tool that generates ETW response time events that can then be correlated with the Windows OS thread scheduling events has potential value in performance investigations of many sorts.

There is such an application response time monitoring tool for both Windows C++ native and .NET applications that is integrated with ETW is called the Scenario class.[8] The Scenario instrumentation library is a free download available from the MSDN Code Gallery site here.

The Scenario class is an ETW wrapper built around an extended version of the .NET Stopwatch class. The standard .NET Stopwatch class is in the System.Diagnostics Namespace and is used to measure elapsed time in a .NET Framework application. The .NET Stopwatch class itself is a managed wrapper around the native Windows APIs called QueryPerformanceFrequency and QueryPerformanceCounter (QPC) that access a high resolution timer and are used to calculate elapsed time. A straightforward extension of the .NET Stopwatch class adds a call to QueryThreadCycleTime (QTCT), a Windows API that provides a measure of CPU usage at the thread level, beginning in Window version 6.

Prior to discussing the use of the application-oriented Scenario instrumentation library, however, we should first take a deeper look at the Windows APIs it utilizes, namely QueryPerformanceCounter (QPC), QueryPerformanceFrequency, and the newer QueryThreadCycleTime (QTCT). Using the Scenario library properly and interpreting the measurement data it produces will benefit from a deeper understanding of how these Windows APIs work.

---

[7] In the Concurrency Visualizer, memory management waits that are resolved very quickly, usually in less than 1 μ-second, correspond to soft page faults. When hard pages faults occur, the tool will show a corresponding disk IO, and the delay is ordinarily several milliseconds in duration, depending on the speed of the paging disk.

[8] The name "Scenario" is a nod to the popular practice of scenario-based development, associated primarily with Agile software development methodologies. See, for example, *Scenario-Based Design of Human-Computer Interactions,* by John Carroll and *User Stories Applied: For Agile Software Development* by Mike Cohen.

QueryPerformanceCounter. The QueryPerformanceCounter and QueryPerformanceFrequency APIs were added to Windows beginning with Windows 2000 when the Performance Monitor developers noticed that the granularity of the Windows system clock was inadequate for measuring disk IO response time. As discussed in footnote 1, the Windows system's time of day clock contains time values that display precision to 100 nanosecond timer units. However, the current time of day clock value maintained by the OS is only updated once per *quantum*, effectively about once every 15.6 milliseconds in current versions of Windows. Using values from the system clock to time disk IOs in early versions of Windows NT, the Logical and Physical Disk\Avg Disk sec/Transfer counters in Perfmon often reported zero values whenever an IO operation started and ran to completion within a single tick of the system clock.

The solution in Windows 2000 was the addition of the QueryPerformanceCounter (QPC) and QueryPerformanceFrequency APIs. In Windows 2000, the QueryPerformanceCounter API returned a value from an **rdtsc** (Read TimeStampCounter) instruction. The TSC is a special hardware performance monitoring counter on Intel-compatible processors that is incremented once per processor clock cycle. On a processor clocked at 2 GHz, for example, one expects two TSC clock cycle ticks to occur every nanosecond.[9] Issuing an **rdtsc** instruction on a processor clocked at 2 GHz returns a clock value considerably more precise than standard Windows timer values delineated in 100 nanosecond units. Since the processor clock speed is hardware-dependent, an additional API call, QueryPerformanceFrequency, was provided to supply the processor clock speed so that the output from successive **rdtsc** instructions could be translated into elapsed wall clock time.

Once the QueryPerformanceCounter and QueryPerformanceFrequency APIs became generally available in Windows, they were rapidly adopted by other applications in need of a more precise timer facility than the Windows system clock. However, developers using QPC() soon began to notice discrepancies in time measurements taken using the **rdtsc** instruction due to the way in which the TSC was implemented in the hardware. There were two major discrepancies that were encountered using the **rdtsc** instruction on Intel-compatible processors, namely:

(1) lack of synchronization of the TSC across processors, and
(2) dynamic changes to the TSC clock update interval as a result of the processor entering a lower power state, slowing *both* the clock rate and the TSC update interval in tandem.

The effect of these TSC anomalies was quickly evident in the disk driver routine responsible for timing the duration of disk operations when running on a multiprocessor, which was especially ironic since QPC was originally built in order to measure disk IO operations accurately. It was possible on a multi-processor for a disk IO that was initiated on CPU 0 and completed on CPU 1 to retrieve a TSC value from

[9] As will be discussed further, elapsed time measurements that are based on successive **rdtsc** instructions are far less precise than the processor's actual instruction execution cycle time. See, for example, this FAQ from the Intel Software Network for an official explanation from the CPU manufacturer.

CPU 1 for the IO completion that was *before* the TSC value retrieved when the IO operation was initiated on CPU 0.[10]

Of less serious concern, the latency of an **rdtsc** instruction was considerably larger than expected for a hardware instruction, on the order of several hundred clock cycles on older Intel hardware. That, and the fact that the **rdtsc** instruction does not serialize the machine, made QPC() unsuitable for timing, say, micro-benchmarks of less than several thousand instructions.

To deal with the drift in TSC values across multiple processors, in Windows 6 (Vista and Windows Server 2008), the Windows QueryPerformanceCounter function was changed to use one of several external, high resolution timer facilities that are usually available on the machine. These include the High Precision Event Timer (HPET), often also referred to as the Multimedia Timer in Windows, and the external ACPI Power Management Timer, another high resolution timer facility that is independent of the main processor hardware. Because these timer facilities are external to the processors, they are capable of supplying uniform values that are consistent across CPUs. (At the same time, QueryPerformanceFrequency was also re-written to return the frequency of the external timer source.) This change effectively fixed the problems associated with accurate measurements of disk IO response time that were evident in Windows 2000 and Windows XP.

However, using an external timer in the QPC implementation does have one major disadvantage, namely latency. If you wrap **rdtsc** instructions around QPC() calls in Windows Vista, you can typically measure latency on the order of 800 nanoseconds to call the QPC API, or roughly 1 μ-second per call. This latency is particularly problematic given how frequently QPC is called. In ETW tracing, for instance, QPC is the default timer routine[11] that is used to generate the event timestamps. When gathering high volume events such as the CSwitch, ReadyThread, ISR and DPC, using QPC for timestamps in ETW generates significant overhead. If one is expecting, say, 20,000 ETW events to be generated per second per processor on a Vista or Windows Server 2008 machine, calling QPC that frequently adds about 2% additional CPU utilization per processor.[12]

---

[10] The flavor of some of the difficulties in dealing with **rdtsc** instructions across multiple CPUs on multiprocessors are discussed in this blog entry written by Microsoft SQL Server customer support representatives.

[11] The clock facility used by default in ETW is QPC. Alternatively, one can specify either the low resolution system timer (an option that should only be used in rare instances where the low resolution 15.6 ms clock ticks suffice), or that an **rdtsc** instruction be issued directly. The **rdtsc** option is termed the "CPU cycle counter" clock resolution option. See this ETW documentation entry on the MSDN Library for details.

[12] ETW is not the only application that routinely relies on QPC-based measurements of elapsed time. When the ETW tracing infrastructure is used to gather OS kernel scheduling events, it is probably the most frequent caller of the API on the machine. Other frequent callers of QPC include the disk driver routines mentioned earlier that measure disk IO response time – reported in Perfmon as the Avg. Disk Secs/Transfer counters – that were re-written in Windows 200 to use QPC. The TCP protocol, which needs to estimate the Round Trip Time (RTT) of packets sent to remote TCP sessions, utilizes QPC for high resolution timing, also. As mentioned earlier, the .NET Framework Stopwatch class allows an application to issue calls to QPC and QPF as an alternative to using the low resolution DateTime.Now() method that access the Windows system clock.

The hardware manufacturers were meanwhile at work making improvements in the TSC hardware to allow it to serve as an efficient and accurate elapsed time counter. Changing the behavior of the TSC when there was a power state change that adjusted the processor's clock cycle time was the first fix. Newer machines now contain a TSC with a *constant tick rate* across power management states. (Which is what I like to call it, instead of what Intel officially calls an *invariant* TSC, terminology I find a little awkward).[13]

The second problem related to the TSC clocks not being synchronized across processors still exists, but the TSCs for all the processor cores resident on a single multi-core socket do run off the same underlying hardware source, at least. Clock drift *across* processor cores remains an issue on multi-socket NUMA hardware, but built-in NUMA node thread and interrupt affinity in Windows minimizes some of these concerns, while not eliminating them completely. Finally, the hardware vendors also report that the latency of the **rdtsc** instruction has improved significantly in current hardware.[14]

The long latency associated with accessing an external clock facility combined, with the **rdtsc** hardware improvements described in footnotes 13 and 14, prompted another round of changes in QueryPerformanceCounter for Windows 7 (and Windows Server 2008 R2). During system initialization, Windows 7 attempts to figure out if the current hardware supports a TSC tick rate that is constant across power state changes. When Windows 7 determines that the processor's TSC tick rate is constant, the QPC routine is set to issue **rdtsc** instructions. If it appears that the TSC is not invariant across processor core frequency changes, then QPC will be resolved as in Windows 6 by calling the machine's external timer. In this fashion, QPC in Windows 7 automatically provides a well-behaved, high resolution hardware clock timer service that uses the low latency **rdtsc** instruction whenever it is well-behaved.

This ability in Windows 7 to resolve the QPC timer service dynamically based on the current hardware is the reason Windows application developers are advised to stay away from using **rdtsc** – unless you absolutely know what you are doing – and to use QPC instead. Coding an inline **rdtsc** instruction is still going to be faster than calling QPC to access the TSC, but using **rdtsc** directly is not for the faint of heart.

QueryThreadCycleTime. Beginning in Windows 6 (which refers to both Vista and Windows Server 2008), there is a new, event-driven mechanism for measuring processor utilization at the thread level. This measurement facility relies on the OS Scheduler issuing an **rdtsc** (Read Time-Stamp Counter) instruction

---

[13] For details, see the Intel hardware manual entitled *Intel® 64 and IA-32 Architectures, Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*. Section 16.9 of this manual discusses the Time-Stamp Counter on Intel hardware and its processor-dependent behavior. The manual reports, "Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer *even if the processor core changes frequency*. This is the architectural behavior moving forward." [Emphasis added.]

[14] Reportedly, the latency of an **rdtsc** instruction has improved by an order of magnitude on current hardware. Unfortunately, the hardware vendors are reluctant to disclose the specific latency of their **rdtsc** instructions due to competitive concerns. The **rdtsc** instruction still does not serialize the processor instruction execution engine, so **rdtsc** continues to return timer values that are subject to some processor core jitter and imprecision. The need to maintain a TSC with a constant tick rate across power state changes also results in some loss of precision in rdtsc return values, affecting just one or two of the least significant clock resolution timer bits.

at the beginning and end of each thread execution dispatch. By accumulating these CPU time measurements at the thread and process level each time a context switch occurs, the OS can maintain an accurate running total of the amount of time on the processor an executing thread consumes. Application programs can then access these accumulated CPU time measurements by calling a new API, QueryThreadCycleTime() and specifying a Thread Id.

QueryThreadCycleTime(), or QTCT, provides measurements of CPU time gathered by issuing an **rdtsc** instruction each time a context switch occurs. Using the same mechanism, the OS Scheduler also keeps track of processor idle time, which can be retrieved by calling either QueryIdleProcessorCycleTime or QueryIdleProcessorCycleTimeEx(), depending on whether multiple processor groups are defined in Windows 7. (Overall CPU utilization is calculated from Idle time for any given interval by subtracting Idle time from the interval duration: *CPU % Busy = (IntervalDuration – Idle Time) * 100 / IntervalDuration*)

While using the **rdtsc** instruction isn't quite as straightforward as most measurement people would like, the OS Scheduler, at least, handles some of the vagaries automatically. In QTCT, CPU time is kept in units of clock ticks, which is model-dependent. If you are running on an older Intel processor that does not have a constant tick rate across power management states, it is going to be a very difficult number to interpret. That is because on older CPUs, whenever there is a power state change that changes the clock frequency, the frequency of the associated Time Stamp Counter (the TSC register) is adjusted in tandem. However, the OS Scheduler does not attempt to adjust for this change in the time between clock ticks. That means that on one of these machines, it is possible for QTCT() to return accumulated clock ticks for an interval in which one or more p-state changes have occurred such that clock ticks of different lengths are aggregated together. Obviously, this creates a problem in interpretation, but, of course, only to the extent that power management changes are actually occurring during thread execution, and it is a problem that only occurs on older hardware.

Given that set of concerns with an **rdtsc**-based measurement mechanism, **QTCT()** remains a major step forward in measuring CPU usage in Windows. Instrumenting the OS Scheduler directly to measure processor usage is the first step towards replacing the legacy sampling technique. It has all the advantages of accuracy and precision that accrue to an event-oriented methodology. Plus, the OS Scheduler issuing an **rdtsc** instruction inline during a context switch is much more efficient than generating an ETW event that must be post-processed subsequently, the approach that xperf uses.

As of Window 7, the OS Scheduler measurements are only exposed through QTCT at the thread level. I suspect that the **rdtsc** anomalies due to the variable rate of the clock on older machines are probably one factor holding up wider adoption, while the scope of retrofitting all the services in & around the OS that currently rely on the sampling-based processor utilization data is probably another.

The QTCT API that gives access to these timings at the thread level does have one other serious design limitation. QTCT currently returns the number of processor cycles a thread has consumed up until the last time a context switch occurred. There is no method that allows a running thread to get an up-to-date, point-in-time measurement that includes the cycle time accumulated up to the present time. A

serializing and synchronization method along those lines would make QTCT suitable for explicitly accounting for CPU usage at a thread level between two phases in the execution of a Windows program.

Calling QTCT inline is exactly what the Scenario instrumentation library, discussed at the beginning of this section, attempts to do, using QPC and QTCT together to gather elapsed and CPU times between two explicit application-designated code markers. The thread-level CPU times that the Scenario instrumentation library returns are subject to this current limitation in QTCT. With that background in hand, we can now take a look at the Scenario class itself.

Using the Scenario instrumentation library. The Scenario class wraps calls to an internal ExtendedStopwatch object that returns both the elapsed time and CPU time of a demarcated application scenario. Once a Scenario object is instantiated by an application, calls to Scenario.Begin() and Scenario.End() are used to mark the beginning and end of a specific application scenario. After the Scenario.End() method executes, the program can access the object's Elapsed and ElapsedCpu time properties. In addition, the Scenario.Begin() and Scenario.End() methods generate ETW events that can be post-processed. The payload of the ETW trace event that is issued by the Scenario.End() method reports the elapsed time and CPU time measurements that are generated internally by the class.

To support more elaborate application response time monitoring scenarios, there is a Scenario.Step method that provides intermediate CPU and wall clock timings. The Scenario class also provides a correlation ID for use in grouping logically related requests. Nested parent-child relationships among scenarios that are part of larger scenarios are also explicitly supported. For details, see the API and ETW payload documentation pages on MSDN Code Gallery.

In summary, the Scenario instrumentation class library provides a convenient way for a developer to indicate in the application the beginning and end of a particular usage scenario. Internally, the Scenario instance uses an **ExtendedStopwatch()** object to gather both wall clock time (using QueryPerformanceCounter) and the CPU ticks from QTCT() for the Scenario when it executes. The Scenario class can then output these measurements in an ETW trace record that records for posterity the elapsed time and CPU time of the designated block of code. Another way to think about the Scenario instrumentation library is that it supports a simple, common measurement pattern, using events to delineate the beginning and end of a sequence. The pattern is similar to the Open Source Application Response Measurement (ARM) standard, but adapted specifically to the Windows diagnostic environment and tailored for the .NET Framework developer.

There is even a capability in the Concurrency Visualizer in the Visual Studio profiling tools to integrate application-oriented Scenario measurements. Hazim Shafi's MSDN article on the Concurrency Visualizer cited earlier illustrates using this facility. In his article, Dr. Shafi describes the Scenario markers as a way to link the visualization of thread execution progress that the Concurrency Visualizer provides to specific phases of the application being profiled. Separately, Shafi discusses specific usage scenarios for the Scenario class in a blog entry here.

Experience with the Scenario instrumentation library, which embeds calls to QTCT() in your application, confirms that QTCT is not a wholly satisfactory solution for obtaining the CPU time of a micro-

benchmark. The one serious limitation is that QTCT returns the CPU cycle count from the most recent thread context switch. QTCT works just fine if you specify a thread id that is currently blocked, but when you call QTCT inline the way the Scenario class does, you get the situation illustrated in Figure 11.

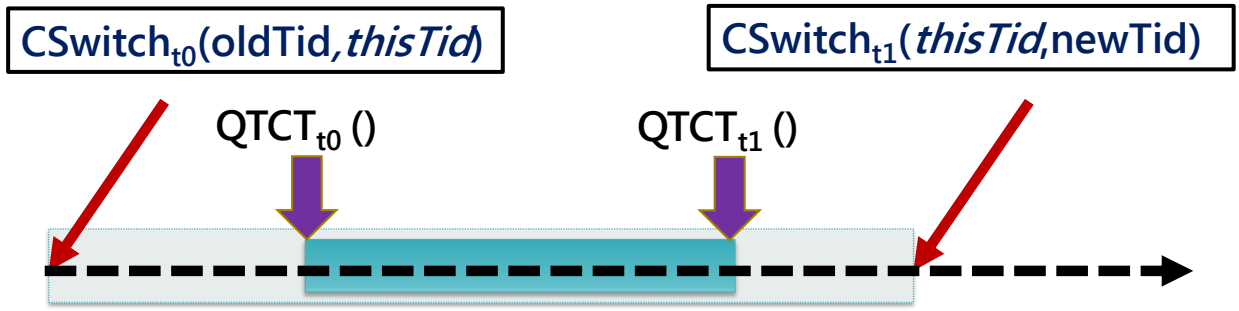


FIGURE 11. QTCT RETURNS THE CPU CYCLE COUNT FROM THE MOST RECENT THREAD CONTEXT SWITCH. WHEN CALLING QUERYTHREADCYCLETIME INLINE FOR THE THREAD YOU WANT TO GATHER THE CYCLE TIME MEASUREMENTS FOR LEADS TO DISCREPANCY WHEN THE GAP BETWEEN THE LAST CONTEXT SWITCH AND THE INLINE CALL TO QTCT IS LARGE. TO FIX THIS, QTCT SHOULD SYNCHRONIZE ITS MEASUREMENT WHEN THE CALLING THREAD IS REQUESTING CYCLE TIME DATA FOR ITSELF.

Here, the value of QTCT returned at t_0 reflects the most recent context switch that occurred. If the last context switch happens to occur long before the block of code you want to measure begins to execute, then the amount of CPU time reported by successive calls to QTCT at the scenario Begin and End includes the CPU time accumulated prior to the Scenario.Begin method call. For inline measurements, there needs to be a way to measure CPU cycles just for the duration of the time between QTCT$_{t0}$ and QTCT$_{t1}$, shaded in blue in the drawing.

This happens to be a scenario that can be seen quite clearly in the VS Concurrency Visualizer, which also suggests a straightforward extension to QTCT that would address the problem. To address this limitation, QTCT needs an option to call the OS Scheduler to update the CPU cycle time inline or to synchronize automatically its cycle time accumulator upon recognizing that the caller is requesting CPU time data on itself. This option would then issue an **rdtsc** instruction inline, driving the same CPU clock cycles consumed update mechanism used when the thread context switches out. This would allow the calling thread to access its current cycle time synchronized to the current point in time.

Comparing Processor Utilization measurements from different sources

Given the use of different CPU measurement techniques in Windows, it is instructive to compare them head-to-head and take their measure. We need not look much further than the Visual Studio 2010 Profiler to compare and contrast the sampling and the event-driven measurement techniques described here in order to understand their trade-offs, primarily with regard to accuracy and efficiency.

The VS Profiler currently provides three different views of an application's processor consumption that we can compare. Most prominently, the Profiler calculates CPU usage by the application being profiled using *sampling* of the processor execution state, the type of profiling that is most frequently used. Meanwhile, the new Rules engine in the Visual Studio 2010 Profiler conveniently gathers the Windows *Process\% Processor Utilization* performance counter for the process being profiled for the duration of a profiling run. Finally, as described earlier, the Concurrency Visualizer calculates CPU consumption of the

process being profiled from the ETW context switch events. First, let's see how the event-oriented approach adopted in the Concurrency Visualizer compares to the other, more familiar process level measures of CPU usage that Windows performance counters provide.

Windows performance counters. The data gathered by the Rules engine is the familiar Windows performance counter called *Process(*n*)\% Processor Time*, where *n* is the name of the process being profiled. This and every other Windows counter analyzed by the Rules engine is gathered by the Profiler Rules engine every 500 milliseconds, by default.[15] Comparing the Windows performance counter measurements to the values calculated by the Concurrency Visualizer from the ETW stream, you will typically notice some discrepancies. The Windows performance counter data is subject to sampling error, which is especially apparent over smaller interval measurement intervals. If you narrow the collection interval even further, gathering performance counters every 100 milliseconds, for example, anomalies due to the discrete nature of the sampling mechanism the CPU performance counters rely on become even more evident.

Figure 12 is a screen shot of the Concurrency Profile's CPU Utilization report. This is the easiest way to compare Windows performance counters (gathered by the Rules engine) to the event-oriented approach that relies on the sequence of ETW CSwitch events. The Concurrency Profile's CPU Utilization report is primarily designed to help orient you to the sequence of events associated with the process being profiled, especially if you believe that process is – or should be – CPU bound. It is a histogram showing CPU utilization over time for the specific process being profiled as a stacked area chart. It also shows the impact of processor utilization by other active processes, as well as the system process (just in case). What remains is the Idle process, essentially a bookkeeping mechanism that allows the OS to measure CPU busy.[16]

[15] As noted earlier, the smallest data collection interval that Windows Performance Monitor application, Perfmon, supports is once every second. This limitation recognizes the data anomalies that can affect the % Processor Time measurements, generally the most frequently used measurement among all the performance counter values available, due to small sample sizes. The % Processor Time measurements reflect samples of the processor's execution state taken roughly 64 times each second. An application like the Visual Studio Profiler calls the performance monitoring interfaces directly using the Performance Data Helper library. (See the Performance Data Helper library documentation for details. Also, see this example that documents using the PDH library functions, to gather one of the % Processor Time counters.) The collection interval that the VS Profiler uses by default is 500 milliseconds, but you can specify any value in milliseconds for the collection interval using the Profiler's Windows Counter Property Page.

[16] When a thread completes its execution cycle and the OS Scheduler see no other threads in the system in the Ready state, Windows "dispatches" the Idle thread, actually a hardware-dependent routine that executes when there is no other real work to do.

Out of a reluctance to suspend the processor entirely using the hardware's HALT instruction, something considered too expensive to do routinely, the Idle thread was initially a branch to a series of No Op instructions executing in a continuous loop. Power management considerations make that expediency very inefficient on today's machines, and Windows has evolved more a conscientious approach to idling the processor, gradually quiescing its power consumption using a power management module that is hardware-dependent.

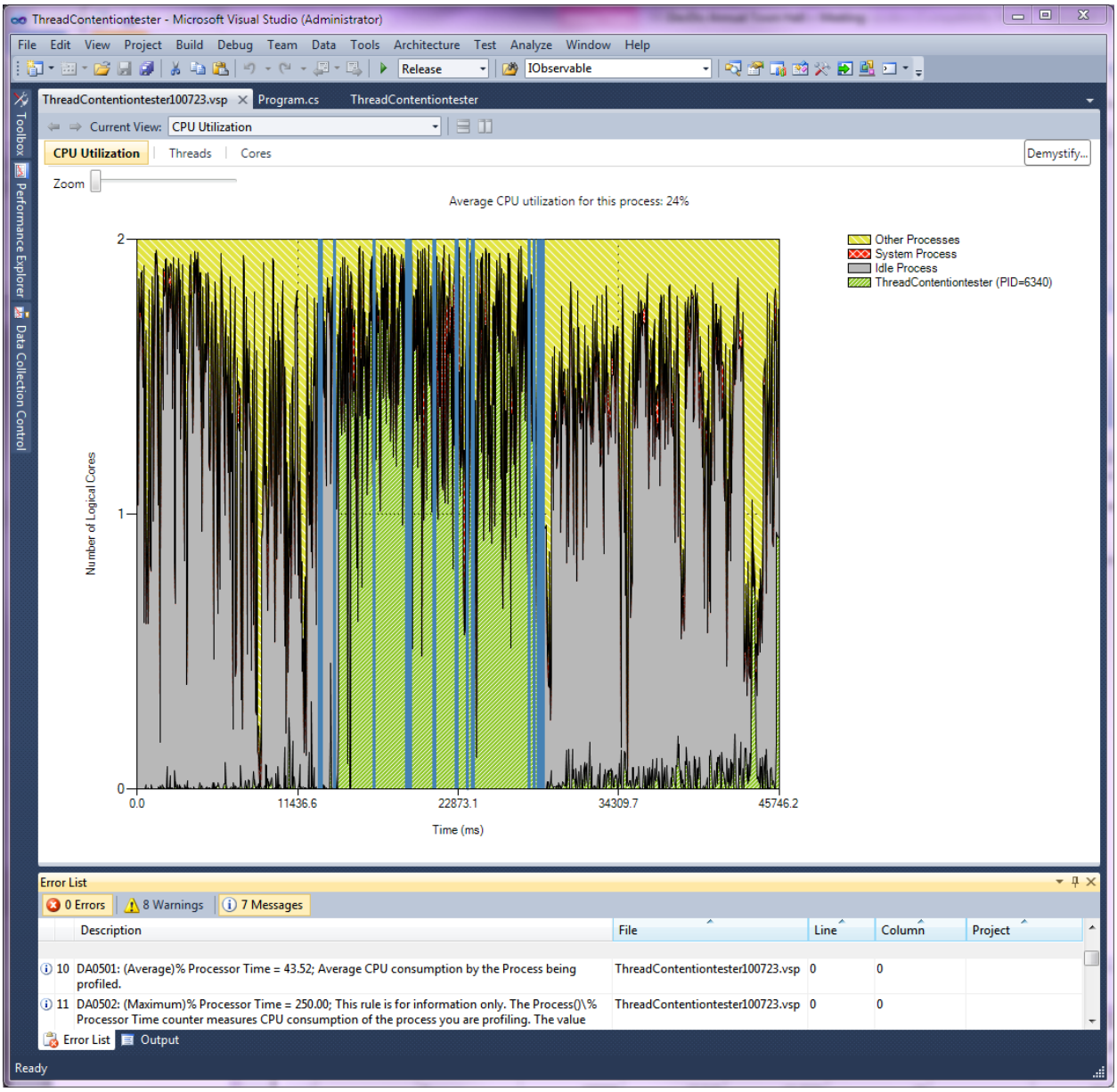


FIGURE 12. A SCREEN SHOT OF THE CONCURRENCY VISUALIZER'S CPU UTILIZATION REPORT, ALSO SHOWING OUTPUT FROM THE PROFILER RULES ENGINE THAT GATHERS THE *PROCESS(N)\% PROCESSOR UTILIZATION* WINDOWS PERFORMANCE COUNTER. THE CONCURRENCY PROFILE REPORTS THAT THE AVERAGE CPU UTILIZATION FOR THE PROCESS BEING PROFILED IS 24%. BASED ON WINDOWS PERFORMANCE COUNTERS, HOWEVER, THE PROCESS WAS REPORTED AS 43.5% BUSY OVERALL.

The Concurrency Profile's CPU Utilization report is a visualization of the event-oriented state machine associated with thread scheduling, this time from the point of view of the processor itself. At any point in time, each available processor is in one of four possible states: Idle, executing a thread from the specific process being profiled, executing a thread from some other process, or executing a system thread.

Visible at the bottom of the screen shot is output from the Profiler Rules engine, two Information-level messages that report the average and maximum CPU utilization of the specific process being profiled,

based on the *Process\% Processor Time* performance counter. Reconstructing the processor scheduling state machine from ETW CSwitch events, the Concurrency profile reports the average CPU Utilization for the process being profiled was 24%. By comparison, using the Windows performance counters gathered over the same execution interval, the process was reported to be 43.5% busy. On the face of it, that is a significant difference. One minor source of discrepancy is the difficulty reconciling the boundaries of the measurement interval using the two different measurement techniques. But that is probably too big a difference, as we will see, to be due to boundary conditions not lining up exactly.

You will also notice a serious data anomaly in the counter data reported by the Rules engine in Figure 12 – the maximum CPU utilization recorded for the process was an interval where the raw *Process(*n*)\% Processor Utilization* performance counter reported 250%, a physical impossibility on this two-way machine. This discrepancy is worth drilling into.

If you click on the Rules engine message, you will navigate to the Profiler's Marks report, which shows the values of the individual performance counters gathered each interval during execution of the process being profiled. I created Figure 13 by copying the raw processor utilization counter data from PDH in the Marks view into an Excel spreadsheet and generating a line chart. (Technically, it is an x-y scatter plot, where the x-axis shows time, the y-axis shows processor utilization, and an Excel spline function is used to connect the dots.) The raw *Process(*n*)\% Processor Utilization* performance counter values are shown as a red dotted line. There are evidently two measurement intervals where the value of the performance counter exceeded 200%. This, of course, is a physical impossibility on a two-way machine, an inconsistency that a knowledgeable performance counter reporting program like Perfmon will normally smooth over. That is certainly large factor in the discrepancy.

Figure 13 also shows the *_Total* instance of the Processor object for comparison. The *_Total* instance of the Processor object, calculated as the inverse of the "utilization" reported by the Idle thread, is effectively purged of the measurement anomalies in the counter data reported at the individual process level. It appears that PDH itself takes care of capping the _Total instance of the Processor\% Processor Time counter value to ensure that it does not exceed 100% per processor for any reporting interval.

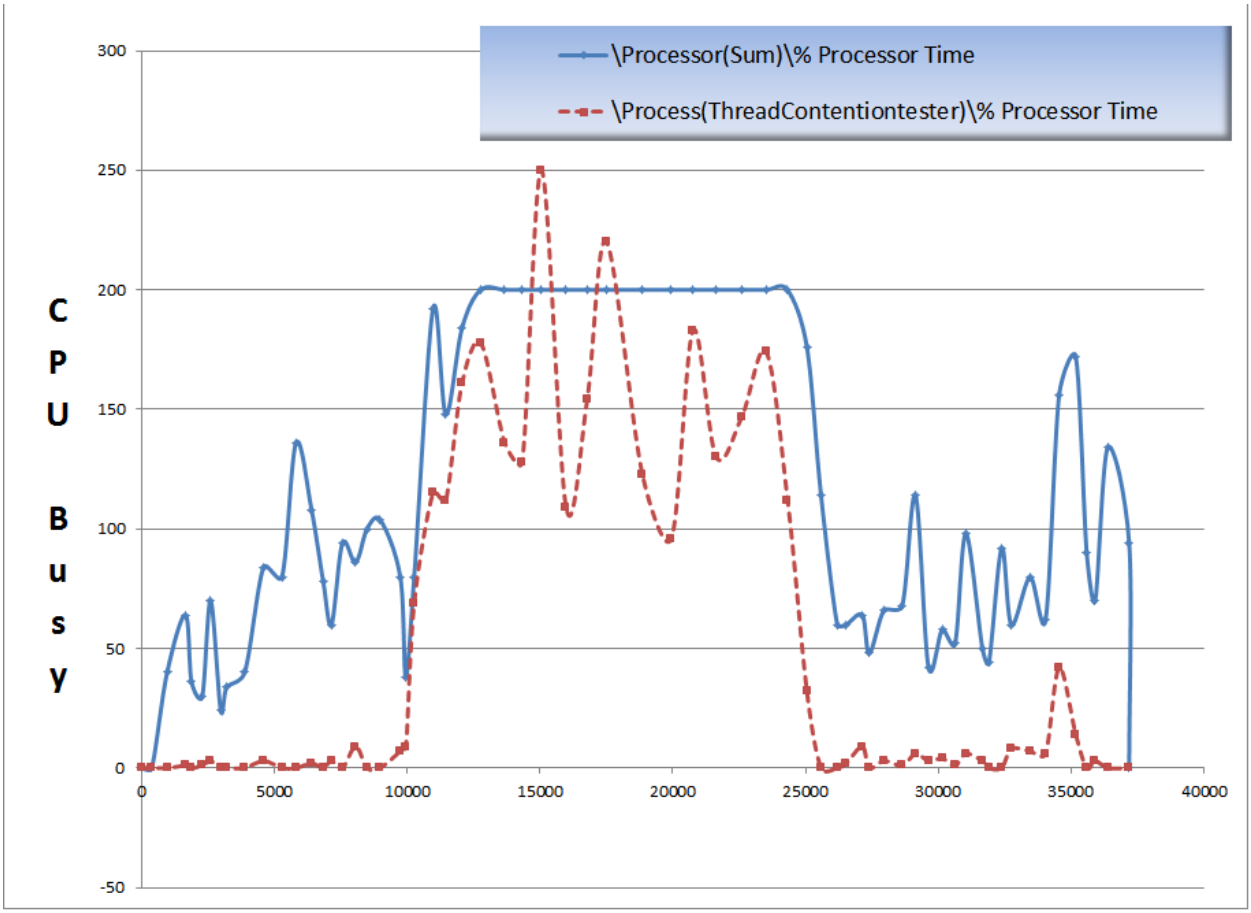


FIGURE 12. PROCESS UTILIZATION AS REPORTED BY WINDOWS PERFORMANCE COUNTERS OVER THE SAME EXECUTION TIME INTERVAL THAT WAS SHOWN IN THE CONCURRENCY VISUALIZER'S CPU UTILIZATION REPORT.

The first observation to be made about the raw Windows performance counter data measurements of processor utilization is that the very small number of samples gathered each collection interval introduces irregularities in the data. Using the Profiler's default 500 millisecond collection interval, the process level % Processor Time counter values are calibrated on only about 30 samples on the processor execution per interval.

Examining a typical thread's execution path, as reconstructed by the Concurrency Visualizer from the event trace, yields some additional insight. What is apparent on modern processors is how small the duration of most thread execution time intervals are – application threads frequently execute continuously for 1 millisecond or less – although, of course, your mileage may vary. Sampling processor execution state 64 times per second as the OS Scheduler does may be too coarse a way to capture these fine-grained thread state transitions.

Another possible explanation that I didn't personally investigate here is that the ETW logger the Concurrency Profiler uses could not keep up with the event stream, effectively like turning on a fire hose, causing some of the CSwitch events to be dropped. On this test machine, I have seen the rate of

context switches exceed 20,000 per CPU, so it wouldn't be all that surprising if the disk logging function didn't quite keep up with the pace.[17]

Those of us that drill into detailed performance measurements for a living are accustomed to dealing with some uncertainty and imprecision in our measurements. So, ignoring this measurement anomaly for the moment, we can satisfy ourselves that the overall shape of the distribution of the raw *Process(*n*)\% Processor Utilization* performance counter values plotted over time is consistent with the Concurrency Profiler's reconstruction of the thread context switch event data. We are evidently measuring the same phenomenon, a consistency that is somewhat comforting.

Sampling processor execution state. A second comparison to be made is between the sample data gathered by the Visual Studio Sampling profiler and the Windows performance counters, which are also based on sampling. A Visual Studio Sampling profile gathers call stacks on a system-wide basis every 10 million instruction cycles, by default. (This collection interval is also adjustable.) On a 2 GHz processor, for example, this means you can expect the profiler to gather about 200 call stack samples each second, or about one every 5 milliseconds. For comparison, this is about three times the sampling rate used by the OS to gather and maintain the Windows performance counter values. The Visual Studio Sampling profiler only analyzes call stacks that reference the specific process profiled – all other call stacks are discarded during its Analysis phase. Analysis of the Visual Studio Sampling profiler call stack data can also suffer if too few samples for process under consideration are gathered. The Performance Rules engine generates a Warning message

Gathering call stacks, of course, allows the Visual Studio Sampling profiler to attribute CPU consumption to specific Modules and their Methods running inside the application. By design, the Sampling profile aggregates all the sampled call stacks accumulated during the profiling run. This is because it is concerned with identifying where in your process CPU time is being consumed, not how much processor time your process is consuming overall. (If you have chosen a Sampling profile to begin with, the working assumption is that the process is using excessive amounts of CPU time since call stack sampling is the profiling technique that allows you to investigate CPU usage most effectively.) In performing this call stack aggregation, however, what is lost is a sense of the *sequence* of the processor utilization events. This is only a serious limitation when the sequence matters, of course.

[17] The Concurrency Visualizer produces a Warning message under these circumstances that displays in the VS Output window and is easily missed: "Warning: your trace missed user mode events, most likely due to excessive context switches or I/O." I might have missed it in this instance, but I don't think so.

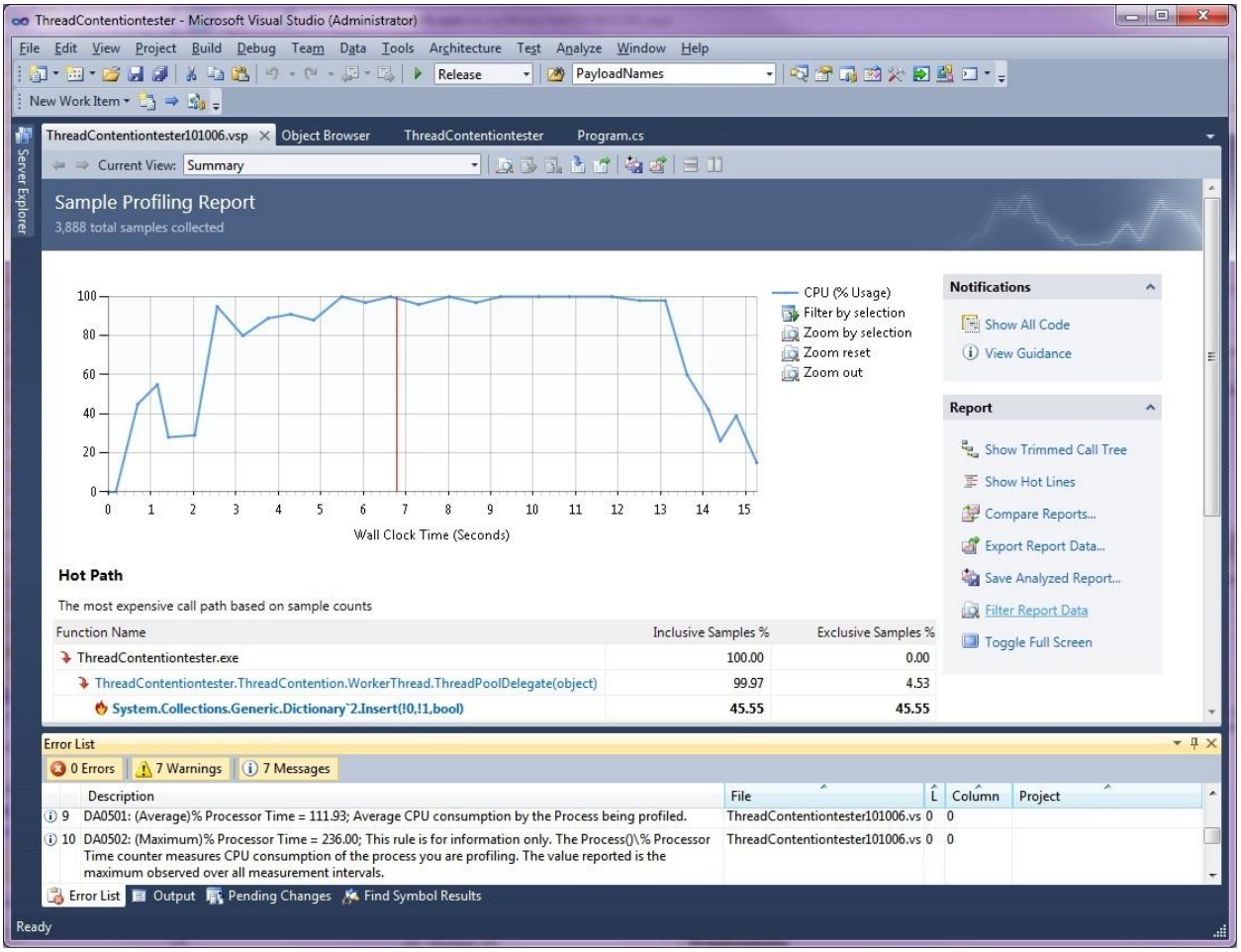


FIGURE 14. A SUMMARY PROFILER SAMPLING REPORT FOR THE SAME PROCESS.

Figure 14 shows a Summary Profiler report for a different run of the same test process. (A separate run is required because the Visual Studio Profiler, by design, can only gather one type of profiling data at a time.) I made the profiling run on a dual-processor machine running at 2.2 GHz. Taking one call stack sample every 10 million clocks, the profiler gathers about 220 call stack samples every second. The process being profiled executed for about 15 seconds, for a total of approximately 220 * 15 samples, 3300 samples per CPU. The report indicates that 3,888 of those sample call stacks originated in the process being profiled. We estimate the CPU utilization as 3,888 / 3300, or about 118%. The raw *Process(*n*)\% Processor Utilization* performance counter, gathered and reported by the Rules engine, reports the process was 112% busy over the same interval. These two measurements are roughly comparable, which should not be too surprising since they both use a similar sampling methodology. (Note that the raw *Process(*n*)\% Processor Utilization* performance counter continues to report intervals where process utilization measurements exceeded 200%, so the anomaly we saw before isn't an isolated incident.)

So there you have three basic ways to measure processor utilization by a process and its threads: (1) using familiar Windows performance counters, (2) a sampling Profile, and (3) by aggregating context switch event data. Overall, all three collection methods provide roughly similar results, assuming there is

enough time to accumulate a sufficient number of samples. The event-oriented technique should provide the most accurate measurements, but the very idea that the Windows event stream is so fast and thick that disk logging may not be able to keep pace warrants some consideration.

When the accuracy of the data is the prime concern, the event-oriented approach is definitely preferred. However, sampling techniques have advantages, too, especially with regard to overhead. As I mentioned, on the machine where I made these profiling runs, I can observe context switch events occurring at rates in excess of 20,000 context switches per CPU per second. Consequently, the overhead of gathering CPU busy measurements by processing context switch events varies as a function of the rate at which context switch events occur. With sampling, the overhead is constant, independent of whatever else may be active on the machine at the time. Over time, the accuracy of sample-based measurements also improves as more and more samples are accumulated. If you need to measure the CPU impact of micro-benchmarks, the sampling-based data on CPU usage are subject to significant error. Over very small measurement intervals, for example, scenarios that execute for 500 milliseconds or less, the event-oriented approach is the only way to ensure accurate measurement results.

Longer term, we see the beginning of a switch to a more accurate, event-oriented approach using instrumentation built into the OS Scheduler. Several new performance tools, including the Resource Manager, the Concurrency Visualizer, and the ETW-based Windows Performance Tools, are currently available that exploit the newer event-oriented measurements. Legacy tools like the Performance Monitor Processor counters and the Task Manager real-time displays, however, continue to support the legacy sampling mechanism in Windows 7.

To conclude this section, I should mention my view that some of the concern over the accuracy of the processor utilization measurements is misplaced. Measurements we are able take of processor activity at an OS level from the outside looking in obscure much of what is actually happening at the hardware level. Relying exclusively on OS software measurements of CPU time consumed by running processes and threads misses aspects of the processor hardware that are crucial to performance. For example, hyper-threading, NUMA effects, and virtualization all impact thread execution time, but these impacts cannot be understood without access to hardware internals. There are very serious performance issues that a naïve reliance on measurements of CPU elapsed time ignores. As a case in point, we will look briefly at the measurement impact of running Windows as a guest machine on a VMware virtualization Host.

Measuring CPU usage on Windows virtual machine guests

Given the widespread use of virtualization technology today, it is appropriate to ask about the impact virtualization has on the measurement of CPU utilization when Windows is running as a guest virtual machine. Without attempting to catalog all the ways in which virtualization influences the measurement data, suffice to say, virtualization has a major impact on performance monitoring of Windows guest machines. While the discussion here will focus on the narrow issue of measuring CPU utilization, we will also touch upon some of the more general measurement issues that arise under virtualization.

To acquire a full and accurate accounting of what is happening to the CPUs in the virtualization environment, it is necessary to pull together *both* the internal measurements from the guest machines and the VM Host measurements. The pervasive effects virtualization introduces into the measurement data gathered from within the virtual machine guest requires us to also consider *external* measurements of processor utilization taken by the VM Host. Fortunately, with the VMHost measurements in hand, it is possible to make sense of the internal measurements.[18] The external measurement data provides a perspective on CPU usage that is quite limited, however. The virtualization host cannot see inside its guest machines or understand the workloads they are running. We provide an example of using internal and external CPU usage measurements in the case study that follows.

From a resource management standpoint, virtualization implies sharing. Virtualizing some resource like the CPU provides the means of sharing it among guest machines. This sharing allows for fuller and more effective utilization of the resource, but potentially leads to contention. Under virtualization, any virtualized resource – which includes processors, RAM, disks, and network interface cards (NICs) – can potentially overload the physical resource that is backing the virtual request. Contention for physical processors, for example, will overload those processors, which then causes delays in scheduling the virtual machine guests for execution. Identifying the nature and extent of these scheduling delays is of fundamental importance whenever you are dealing with performance problems affecting one or more of the guest machines that are sharing some VM Host.

Queuing for virtual processors. Contention for virtual processors leads to queuing delays. To quantify the delays associated with queuing for virtual processors, it is necessary to look at external measures of processor busy and processor queuing gathered by the VM Host. Fortunately, the VMware Host system that is responsible for scheduling guest machines for execution is also capable of measuring processor usage by guest machines very accurately. When there is contention for physical processors, the VMHost measures that as well. The VMware measure that is most important to understanding the extent to which processor contention is occurring is known as *Ready Time*. *Ready Time* in VMware is the amount of time a virtual machine that has requested service and is ready to run is delayed in the VMware Host scheduler queue waiting to be scheduled for execution. It is analogous to processor queue time. [19]

Unfortunately, CPU usage measurements gathered from inside the guest OS are not so precise. The fundamental measurement issue affecting internal measurements is that guest machine requests to read the hardware clock and set hardware-based timer interrupts are virtualized. Since running as a virtual machine is transparent to the guest OS – with some exceptions, as discussed later, measurements taken from inside the virtual machine guest are unaware of any delays introduced due to resource sharing. Consequently, measurements taken from inside the guest machine cannot reliably quantify queuing delays associated with queuing for virtual processors.

[18] Note that in the discussion that follows, the examples are drawn specifically from running Windows guest machines on VMware. Each of the other popular virtualization products does have related performance monitoring concerns, however.

[19] See the VMware white paper, "Ready Time Observations," available at http://www.vmware.com/pdf/esx3_ready_time.pdf, for details.

However, it is sometimes possible that secondary measures do provide evidence that contention-induced virtualization delays are occurring. For example, the *System**Processor Queue Length* performance counter in Windows, which is sampled as discussed earlier, sometimes provide evidence that contention for virtual processors is delaying execution of key tasks. From an internal perspective, elongation of the processor queue can result when executing threads become backed up. Timely processing of interrupts, especially timer interrupts are also prone to becoming backed up when the virtual machine is blocked from executing. Because of the way the *System**Processor Queue Length* performance counter in Windows is calculated using sampling, it is not, however, a very robust indicator of processor contention in a virtualization environment.

A service level-oriented measurement of application response time remains the best overall indication that there is a performance problem – with or without virtualization. The accuracy of internal measurements of application response time is affected by the virtualization environment, which virtualizes all the clock and timer services that the guest machine uses. Measurements of application response time that are gathered externally, such as the response time of web service calls made using instrumentation embedded in the web browser client-side Javascript code during execution, remain very accurate. Since it does not have visibility inside the guest machine, the VMware Host software does not have access to service level-oriented response time measurements for any applications running on the guest. One of the limitations of current VM Host scheduling schemes that attempt dynamic load-balancing in response to resource contention is that they are based solely on resource utilization metrics and do not take application response time into consideration. In the case study discussed below, we review measurements of both resource utilization and application response time.

Virtualized clocks. VMware virtualizes all calls made from the guest OS to hardware-based clock and timer services on the VMware Host. For the Windows OS, these include (a) the periodic clock interrupt that Windows relies upon to maintain the System Time of Day clock, (b) calls to the HPET, and (c) the **rdtsc** instruction itself.[20] You will recall from an earlier discussion that the QueryPerformanceCounter function that is used in performance monitoring to generate granular measurements of elapsed time uses the hardware **rdtsc** instruction in Windows 7, but reverts to the HPET external timer whenever rdtsc cannot be trusted. In VMware, virtualization influences all three time sources that Windows depends upon in performance measurement. From inside the Windows guest machine, there is no clock or timer service that is consistently reliable.

The fact that virtualization impacts external, hardware-based clock and timer services is unsurprising. Whenever the guest machine accesses an external hardware timer or clock, that access is virtualized like any other access to an external device. Any external device IO request is intercepted by the VM Host software, which then redirects the request to the actual hardware device. If the target device happens to be busy servicing another request originating from a different virtual machine, the VM Host software must queue the request. When the actual hardware device completes its processing of the request,

[20] See a VMware white paper entitled, "Timekeeping in VMware Virtual Machines," which has an extended discussion of the clock and timer distortions that occur in Windows guest machines when there are virtual machine scheduling delays.

there is an additional processing delay associated with the VM Host routing the result to the guest machine that originated the request. In the case of a synchronous HPET timer request, this interception and routing overhead leads to some amount of "jitter" in interpreting the clock values that are retrieved that is not otherwise present when Windows is running in a native mode.

An asynchronous timer request of the type Windows uses to maintain the System Time of Day clock is subject to an additional delay between the time that the VMware Host software services the original device interrupt and the time that the virtualized device interrupt is presented to the guest machine. An additional consideration is at that the specific point in time when the periodic clock interval the OS Scheduler relies upon is scheduled to expire, the guest machine might be delayed in the VMware Host scheduler (where it is accumulating Ready time). In VMware, even timings based on the lightweight **rdtsc** instruction issued from guest machines are subject to virtualization delays, despite that fact that an **rdtsc** instruction can normally be issued by a program executing at any protection level. The VMware Host OS traps all **rdtsc** instructions and returns virtualized timer values.

With this background and understanding of how virtualization perturbs all manner of Windows clocks and timer services, let's look specifically at how having undependable clocks affects the accuracy of Windows performance measurements. Virtualization affects the clock interrupts that the Windows Scheduler relies on to maintain its System Time of Day clock and perform CPU accounting the most. Servicing interrupts of any type are subject to scheduling delays when Windows is running as a virtual machine guest under VMware. As noted above, the potential for clock interrupts to be deferred impacts the CPU accounting function that is performed by the OS Scheduler. The interval between successive clock interrupt is no longer uniform. When the CPU accounting interface in Windows converts samples into % Processor Time, having non-uniform measurement intervals distorts this calculation.

Moreover, it is possible for VMware guest machine scheduling delays to grow large enough to cause some periodic timer interrupts to be dropped entirely. In VMware terminology, these are known as *lost ticks*, another tell-tale sign of contention for physical processors. In extreme cases where the backlog of timer interrupts ever exceeds 60 seconds, VMware attempts a radical re-synchronization of the time of day clock in the guest machine, zeros out its backlog of timer interrupts, and starts over.

In theory, at least, one should be able to adjust most performance measurements taken internally by the virtual machine guest OS based on how much Ready delay the guest machine encounters. In practice, this measurement reconciliation, however, is significantly complicated by the fact that clock and timer services of all types are virtualized in VMware. When a Windows guest machine is suffering from Ready Time delays, these scheduling delays tend to distort the clock and timer facilities that are used in Windows performance monitoring. Ironically, it is precisely when these internal OS measurements of processor utilization are most valuable that internal measurements are apt to be the least reliable.

Aware that the clocks and timers used by Windows for performance monitoring are distorted when there is contention for physical processors, VMware's timer virtualization facilities attempts to smooth out the distortion caused by VMware scheduling delays. VMware's virtualized timer services attempt to

deliver a consistent stream of *apparent time* to the guest machine, smoothing out the major distortions created by non-uniform timer intervals. The impact of VMware *apparent time* on performance measurements taken from inside a Windows guest machine is not well understood, as of this writing. It is safe to safe that this is area that badly needs investigation.

VMware's use of apparent time is designed to smooth out of the some of the obvious distortions that clock and timer-based measurements are subject to on Windows guest machines. Apparent time helps with all manner of Windows performance counters that are based on simple counting of events, including, for example, Pages/sec, Disk transfer/sec, TCP segments/sec, etc. These are all performance counters whose values are derived in the performance monitoring software by calculating an interval delta, and then dividing by the interval duration to create a rate, i.e., events/second. In making that events/second calculation, the numerator – the number of these events that were observed during the interval – remains a valid count field. What is *not* reliable under VMware, however, is the interval duration, something which is derived from two successive calls to virtualized timer services that may or may not be delayed significantly. There should be no doubt that the events that were counted during the measurement interval actually occurred. What is suspect is the calculation of the *rate* that those events occurred that is performed by Perfmon and other performance monitoring applications.[21]

Clock distortion that impacts the CPU usage metrics is arguably more serious. The CPU time accounting function of the OS Scheduler assumes that the *quantum* between any two successive periodic clock interrupts is uniform. Under VMware, this is not a valid assumption.[22] At the very least, managing a VMware server farm requires access to the measurements the VMHost scheduler gathers on Host and Guest CPU utilization and Guest Ready time.

Unfortunately, due to the clock distortions that can occur, none of the event-oriented measurement techniques discussed earlier yield reliable measurement results under VMware either. As a final note of caution on the subject, consider the use of the **rdtsc** instruction to calculate the elapsed time between two events, for example, the instrumented OS Scheduler that issues an **rdtsc** instruction on a context switch that transitions a thread into the running state and a later context switch when the running thread blocks. According to VMware,

[21] Performance counter rate/second values are automatically calculated by the PDH library functions described in an earlier footnote, based on the counter type definition.

[22] In theory, one ought to be able to correct for these distortions in the uniform time fabric from inside an "enlightened" guest machine. Whenever Microsoft adds a feature to the Windows OS to make it perform better when running as a Hyper-V guest machine, it uses the terminology "enlightened" to describe that OS feature.

For example, the TCP/IP networking driver stack in Windows 7 is enlightened when Windows is running as a Hyper-V guest machine. Running as a Hyper-V guest machine, the TCP/IP stack can avoid performing many of the time-consuming consistency checks that the enlightened driver knows were already performed on the packet by the Hyper-V Host machine for the physical NIC. To date, there is no "enlightened" performance monitoring facilities of the guest OS, but similar to VMware ESX, Hyper-V provides a rich set performance monitoring counters of its own. VMware, for obvious reasons, cannot rely on the Windows OS ever being enlightened about running as an ESX guest. But VMware provides a number of Windows facilities and tools that are VMware-aware, such as a driver for its virtual NICs that contains a similarly abbreviated TCP/IP networking stack.

> Current VMware products virtualize the TSC in apparent time. The virtual TSC stays in step with the other timer devices visible in the virtual machine. Like those devices, the virtual TSC falls behind real time when there is a backlog of timer interrupts, and catches up as the backlog is cleared. *The virtual TSC does not count cycles of code run on the virtual CPU; it advances even when the virtual CPU is not running*. [Emphasis added.][23]

Between the time that the 1st context switch occurred and the thread was dispatched on the guest machine and the 2nd context switch occurred when the thread blocked, the guest machine itself is subject to being blocked by the VMware Host scheduler. The OS Scheduler performing its QueryThreadCycleTime CPU time accounting function is not aware that the guest machine itself was blocked. The OS Scheduler naively assumes that the thread was running continuously all the time that elapsed between its two successive **rdtsc** instructions.

In summary, virtualization has the potential to distort significantly both the legacy sampling based measurements of processor utilization and the newer event-oriented measurement techniques. The underlying cause of these distortions to the measurements is that clock interrupts and other timer services are subject to VM Host scheduling delays whenever there is contention for physical processors. In theory, it should be possible to correct for these perturbations in the guest machine CPU time measurements using the VMware Host measurements of guest machine Ready Time. In practice, the ability to reconcile the CPU usage metrics across VMware Hosts and their Windows guest machines has not yet been thoroughly explored.

Case study. To see how guest machine performance monitoring is impacted when the VMware Host machine is overloaded, we set up a test environment with a 2-way multiprocessor running ESX Server version 5. We then defined 4 Windows Server guest machines, 2 with 2 virtual processors assigned, and 2 guest machines with single processors. With 4 machines and 6 virtual CPUs defined, running our CPU soaker program concurrently on each guest machine overloaded the VMware host machine. Figure 15 shows the utilization of the VMware guest machines defined, based on the external measurements gathered by the VMware Host software. An overlay, a dotted line plotted against the right hand chart y-axis, shows the VMware Host CPUs running at near 100% utilization for the duration of the test, which ran for three hours.

The chart in Figure 15 shows that one of the two-way Windows guest machines, R1-2008, accounts for about 70% utilization of one physical CPU. Another two-way Windows guest, SE-2008-32, accounts for about 40-50% utilization. Two additional Windows guest machines, defined to run only a single processor, are also shown, executing concurrently during the interval. The VM Host software evidently consumes the remaining CPU cycles that are accounted for, approximately 20-30% of one physical CPU.

[23] "Timekeeping in VMware Virtual Machines," a VMware white paper, cited in an earlier footnote.

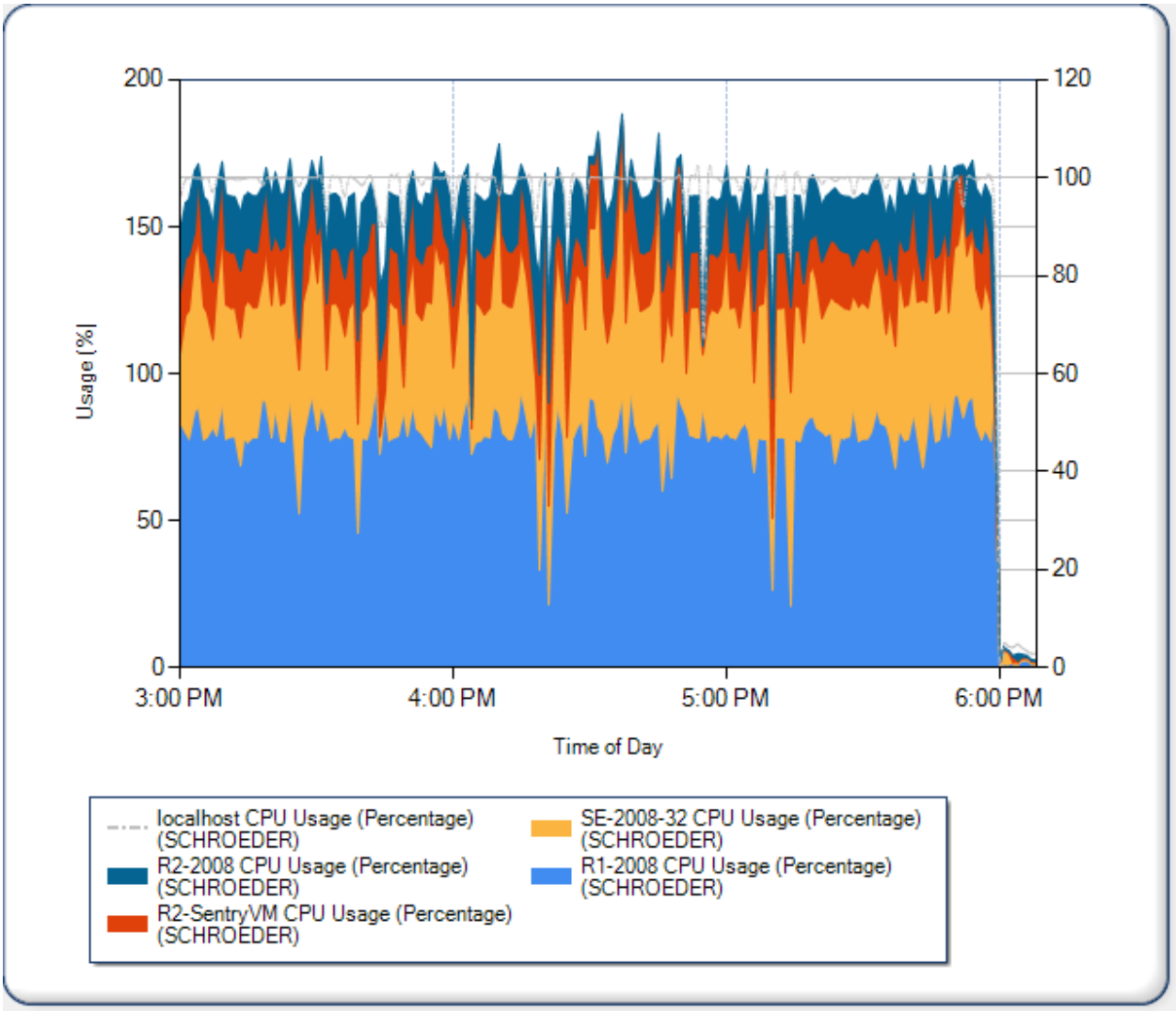


FIGURE 15. CPU USAGE OF 4 GUEST MACHINES RUNNING A VMWARE ESX HOST ON A 2-WAY MULTIPROCESSOR DURING A THREE-HOUR BENCHMARK RUN. THE UTILIZATION OF THE FOUR TEST WINDOWS GUEST MACHINES THAT ARE RUNNING IS SHOWN.

The aim of the test was to induce sufficient Ready Time in the guest Windows machines so that we could evaluate the internal performance measurement of CPU utilization under the worst possible conditions. Figure 16 illustrates the amount of wait time accumulated by each of the guest machines due to contention for the physical processors. With each of the guest machines running CPU soaker jobs, the physical CPUs that VMware is managing are overloaded. This leads to long delays as guest machines accumulate the Ready Time that is illustrated.

Each data collection interval graphed in Figure 16 corresponds to 20 seconds of elapsed time of the test run. Because the physical CPUs are overloaded, the guest machines can accumulate as much as 50 seconds of Ready Time during a 20-second execution interval. Note that the 2-way virtual machines accumulate far more Ready Time than the 0ne-CPU guests. Before dispatching an *n*-way virtual machine, the VMware Host scheduler waits until *n* processors are free. This is to ensure that workloads executing

inside the guest machine never experience an anomalous situation where the expected number of virtual processors are not available which might cause applications to fail or the OS to initiate some sort of hardware recovery operation.

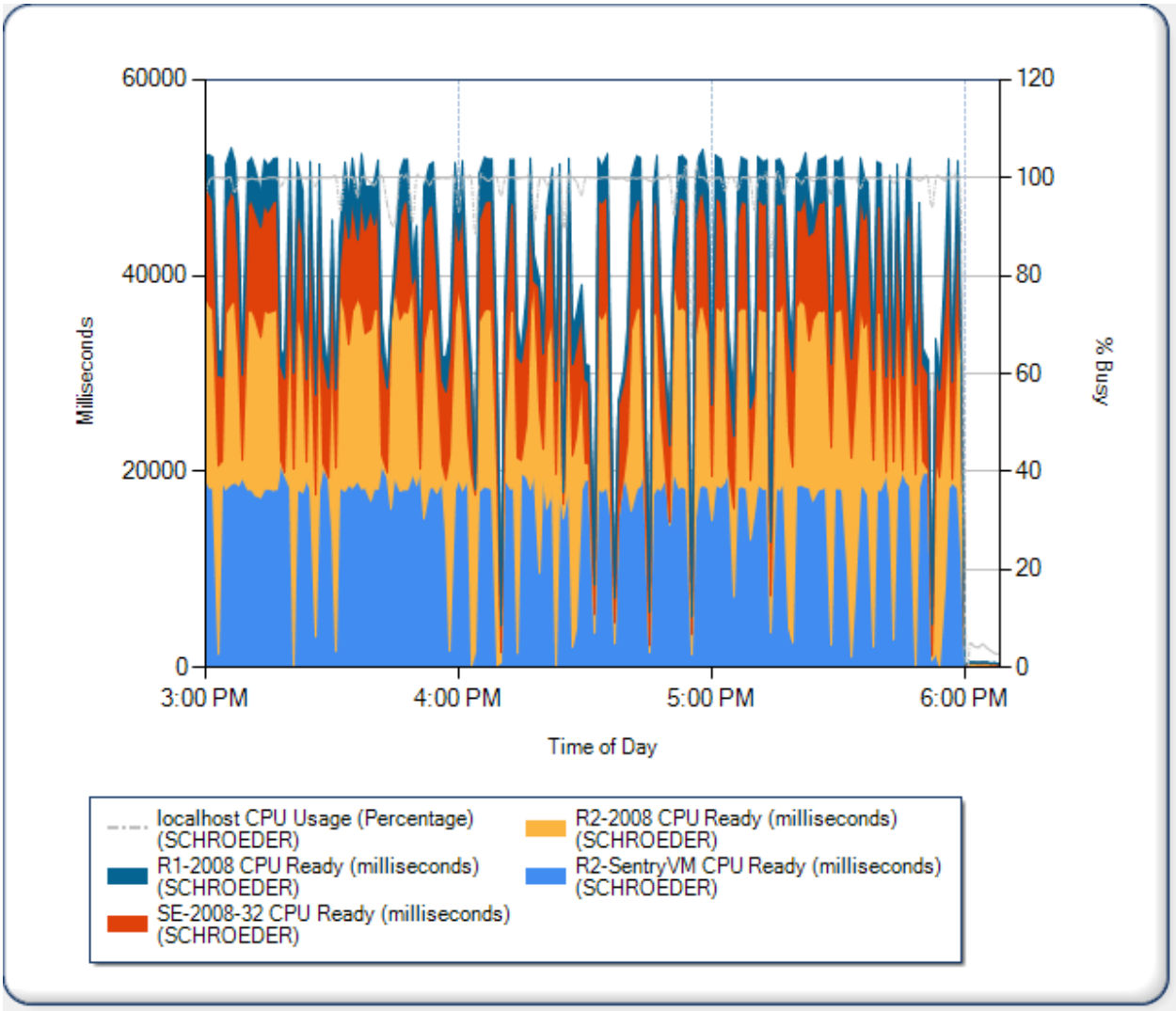


FIGURE 16. READY TIME (IN MILLISECONDS) ACCUMULATED BY THE FOUR WINDOWS GUEST MACHINES EACH INTERVAL DURING THE TEST RUN.

The test environment is running soaker workloads designed to saturate all six defined virtual processors that we forced to execute on a machine with only two physical CPUs. It is evident that contention for the processors in this instance produces a good deal of Ready Time delays. Next, let's turn to an inside view to see the impact of this processor contention on each of the Windows guest machines.

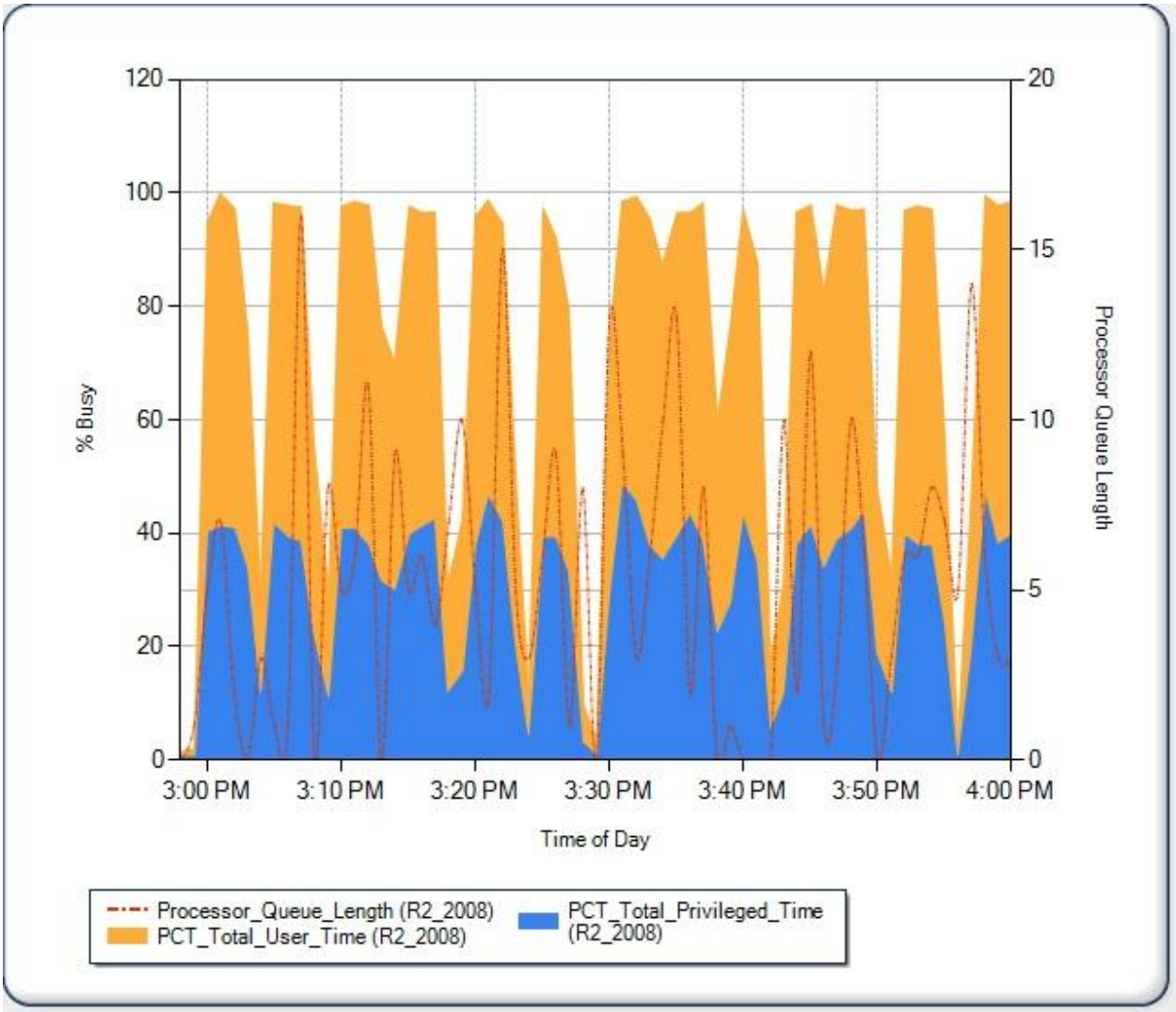


FIGURE 17. THE INTERNAL VIEW OF PROCESSOR UTILIZATION AND THE PROCESSOR QUEUE LENGTH FOR ONE OF THE 2-WAY VIRTUAL MACHINES. PROCESSOR UTILIZATION IS BROKEN OUT INTO KERNEL MODE TIME AND USER MODE TIME AND IS AVERAGED ACROSS BOTH VIRTUAL PROCESSORS.

Figure 17 provides an internal view of processor utilization and the processor queue length for one of the defined 2-way guest machines. For the sake of comparison Figure 18 zooms into the external measurements of processor utilization for the same one-hour interval. Peaks and valleys in the internal data do appear to match up well with the external data.

For example, a peak interval in the external utilization data such as the spike that occurs at 4:04 PM appears to correspond to a peak interval in the internal data. At 4:04 pm, internal measurements of CPU time reported overall processor utilization averaging almost 97% busy. The external VMware Host measurements indicate the guest machine consuming CPU averaging 44% of a single CPU. Another interval at 4:15 pm in which the guest machine suffers from starvation according to the external measurements shows little CPU time being consumed internally. The external measurement for this

starvation interval was just over 4% utilization, while internally the Windows measurements reported only about 5% utilization.

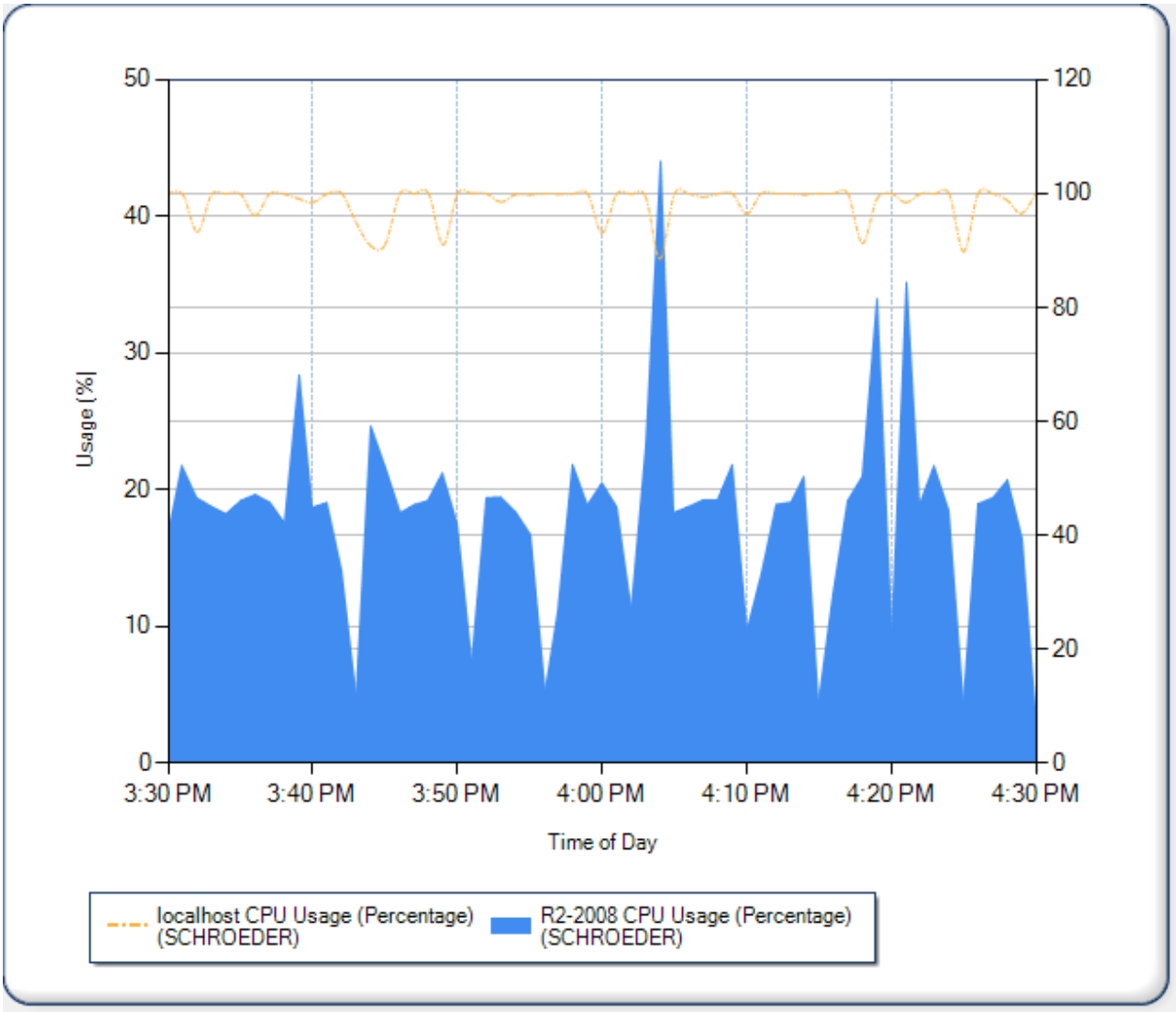


FIGURE 18. THE EXTERNAL MEASUREMENTS OF CPU UTILIZATION TAKEN BY THE VMWARE HOST SOFTWARE FOR THE R2-2008 WINDOWS 2-WAY GUEST MACHINE.

Figure 19 is a different view of the erratic CPU usage pattern that arises due to contention for physical CPUs in this example. It is a box plot showing the distribution of processor utilization measurements reported internally each hour. The bulk of the measurement intervals show the processor heavily utilized, running at or near 100% capacity. The box plot view also highlights those measurement intervals where, in contrast, little or no processor resources were used. These intervals of low utilization correspond to periods where the VMware Host scheduler provided few opportunities for the guest machine to execute, due to contention for physical CPUs.

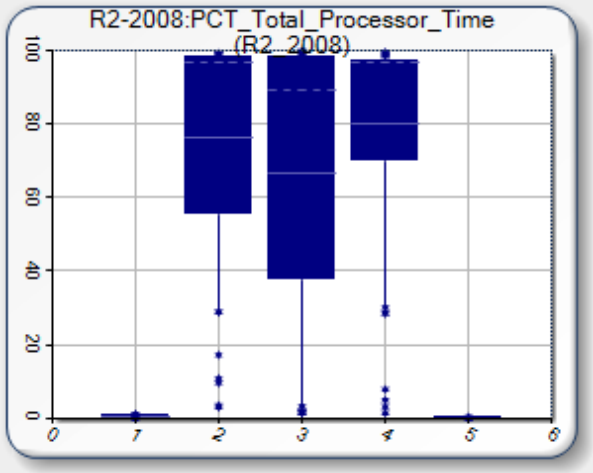


FIGURE 19. A BOX PLOT OF THE % PROCESSOR TIME PERFORMANCE COUNTER FROM ONE OF THE 2 CPU WINDOWS GUEST MACHINES FROM THE VMWARE BENCHMARK TEST. INTERVALS OF VERY LOW UTILIZATION CORRESPOND TO PERIODS WHERE THE VMWARE HOST SCHEDULER PROVIDED FEW OPPORTUNITIES FOR THE GUEST MACHINE TO EXECUTE, DUE TO CONTENTION FOR PHYSICAL CPUS.

Finally, let's look at measurements of application response time to see how this contention for physical CPUs is impacting the execution time of the application. The application response times were measured internally by the application using embedded calls to the Scenario instrumentation class library, which relies on the QueryPerformanceCounter (QPC) API to measure elapsed time. Since QPC() resorts to using either an **rdtsc** instruction or making an external call to the HPET in Windows 6.1 – as discussed above, the scenario timings in a virtual environment reflect VMware's use of apparent time.

To establish a baseline of performance expectations, we first ran each guest Windows machine in a standalone mode where only one virtual machine was active on the VMware Host. We were then able to compare the performance of the same application running with all four virtual machines active, the conditions which generated an extreme amount of contention for physical CPUs. The application response time measurements are illustrated in Table 1. In standalone mode, the 1-CPU guest was able to complete one full iteration of the benchmarking program's set of tasks, which execute concurrently until completion, in 1:27 (mm:ss). The 2-CPU flavor of the guest machine was able to complete the identical set of tasks in just 54 seconds, an improvement of approximately 40%.

	1 CPU Guest	2-CPU Guest
standalone	**1:27**	**0:54**
contention	**2:00**	**2:15**
Δ	**+38%**	**+150%**

TABLE 1. COMPARING APPLICATION RESPONSE TIMES WITH AND WITHOUT CONTENTION FOR PHYSICAL CPUS.

When all four Windows guest machines were executing concurrently, application response time increased by 38% on the 1-CPU guest machine. At the same time, application response time on the 2-CPU guest machine increased by 150%. Under conditions in the test where we generated an extreme amount of contention for physical CPUs, response times were reported that were worse when the application was run on the 2-CPU guest machines compared to the 1-CPU guests. This performance anomaly is due to the fact that VMware requires two available processors before it will schedule execution of a 2-CPU guest, so the guest machines that require fewer CPUs have more opportunities to run. The paradox of running in the virtualization environment is that a workload that normally requires concurrent access to more processor capacity can experience more processor contention-related delays.

In summary, virtualization perturbs the clocks and timers that the guest OS relies upon in performance measurement. It requires analysis of a significant amount of performance data, but it is possible to reconcile the performance measurements gathered by a Windows guest machine with the external measurements of processor utilization gathered by the VMware Host. The external measurements provide an accurate assessment of the guest machine's usage of the physical CPUs that are managed by the VMware Host. Contention for virtualized resources can create performance problems when the applications accessing those resources are delayed when they are in use by a different guest machine. The internal measurements taken by the guest OS are the only way to assess the impact of these resource constraints on the performance of the applications running on the guest machine.

Measuring processor utilization from inside the hardware

To conclude this article, I would like to reiterate that some of the concern expressed over the accuracy of the legacy processor utilization measurements in Windows is misplaced. I am sympathetic to complaints that the anomalies discussed here lead to measurements of processor utilization and processor queuing in Windows that contradict established analytic approaches to modeling computer performance. This is unfortunate, and is something that should be fixed. As discussed in some detail here, an event-driven approach to deriving these CPU usage measurements promises significantly greater fidelity.

However, I worry about performance analysts relying on external measurements of CPU time consumed by running processes and threads too much these days. That is not because the processor is any less critical a processing resource these days, but more because I see very serious issues that a naïve reliance on measurements of CPU elapsed time ignores. Measurements we are able take of processor activity at an OS level from the outside looking in obscure much of what is actually happening at the hardware level.

Let me explain. On current Intel and AMD microprocessors, what is going on *inside* the processor is often more important than any measurements you can take externally. On the Intel microarchitecture, for example,[24] every instruction processed is executed in a series of 20-30 steps in the instruction execution pipeline. If all goes well, each pipeline step in the execution of a complete instruction takes

[24] See, for example, this blog post that discusses the microarchitecture of the Core i7 (or Nehalem) machines that are Intel's current flagship CPU product: http://blogs.msdn.com/ddperf/archive/2008/04/01/thoughts-on-intel-s-recent-hardware-announcements.aspx.

one clock cycle to complete. In addition, these processors feature *superscalar* architectures. They are capable of exploiting instruction level parallelism to execute multiple instructions in parallel during each clock cycle.

In the previous paragraph I have been careful to cloak my account of the marvelous capabilities of current generation microprocessors in terms of their potential performance because, unfortunately, it is largely unrealized potential. Processors capable of executing multiple instructions per cycle seldom reach that potential on real world workloads. They contain elaborate parallel instruction execution pipelining hardware that is seldom fully utilized.

The most common reason that the hardware is under-utilized is the performance bottleneck in current microprocessors known as the "memory wall." The elaborate pipelined, superscalar microarchitecture of the processor *stalls* whenever the CPU is forced to fetch data or instructions from its attached RAM, instead of its on-board memory caches. Essentially, it takes so long to access DRAM that referencing data that is not already available in one of the caches is pretty much guaranteed to stall the pipeline. (Note: a pipeline *stall* is defined as a processor clock cycle in which no instructions are retired. The "instructions retired" terminology reflects the fact that the original instructions are decomposed into *μ-ops* inside the pipeline, which absent interlocks and other dependencies can be executed out of order and in parallel. Once all the μ-ops associated with the instruction are complete, the original instruction can be *retired*. Instructions must be retired in sequence, preserving the intended logical instruction execution stream of the original program.)

The term "memory wall" was first coined in 1994 when engineers first started to notice of the fact that microprocessor clock rate was increasing considerably faster than the speed of DRAM memory access. (See Wulf and McKee's article entitled "Hitting the Memory Wall: Implications of the Obvious" that is posted here.) For example, an Intel tutorial on hardware performance (here) gives the following (approximate) timings on memory latency:

Cache Level	Latency (ns)
1	1
2	5
3	120

By the way, these are representative, ballpark estimates – the actual memory latency on the machines you are running is apt to be different.

The tutorial then observes, "If a load misses in all caches it will eventually come to the head of the ROB and block instruction retirement." The ROB is the microarchitecture's Re-Order Buffer, a staging area in the pipeline where μ-ops from decoded instructions are queued prior to execution inside the pipeline. In plainer English, the delay associated with a memory reference that misses all the caches and must be resolved by latching DRAM is so long that the instruction pipeline is pretty much guaranteed to stall, no matter how much instruction pre-fetching and other tricks to keep the pipeline loaded are performed.

Consider a program that executes two Load instructions back-to-back that each references a memory location where a number is stored, multiplies them together, and stores them back in a third memory location. Something like (in pseudo-code):

```
L     Register1, faraway1
L     Register2, faraway2
M     Register1, Register2
ST    Register1, faraway3
```

Now, suppose all three references to the faraway memory locations are cache misses. The first Load will take about 120 ns. to execute, and so will the second. At that point, the Multiply Register instruction can execute very quickly because the operands are available in internal registers. The final Store instruction will also execute quickly, writing directly to cache and deferring the update physical memory as long as possible. Effectively, due to the memory wall, this is a program that will be able execute 4 instructions in roughly 240 nanoseconds. If the memory references the Load instructions need are L1 cache hits, the same instruction sequence can execute in about 1 nanosecond. (Note these are *super-scalar* processors capable of executing multiple instructions in parallel. With effective caching, the instruction sequence could be completed in just 2 or 3 clock cycles.)

The point of this example is the folly of naively equating CPU *time* with CPU instruction execution rates. Given the way these Intel processors perform, it is relatively easy to find examples of programs that execute for long periods of time, yet do not execute their instructions very efficiently. The Intel x64 architecture currently incorporates simultaneous multithreading (what Intel calls Hyper-threading), dynamic over-clocking and other high-end hardware features. Server machines with complex NUMA architectures are in widespread use. On these complex machines, relying only on execution time on the processor from the vantage point of the OS misses many important aspects of efficient use of processor resources.

Hardware performance counters are available that can report on actual CPU instruction execution rates, and many other useful internal metrics. Currently, these hardware performance counters are available only in specialized tools, such as Intel's vTune product, a similar product from AMD, and some limited support for Intel hardware performance counters in the Visual Studio Profiler. Intel hardware has a very rich performance monitoring interface. In the past, Intel was willing to make wholesale changes to the performance monitoring interface from release to release of the hardware that discouraged those software developers outside of Intel who were trying to keep up with all that model-dependent behavior.

Recently, however, Intel has designated a small set of performance counters of near-universal appeal and declared its intent to support this stable counter set on all x64 architecture machines going forward. This "architectural" counter set includes the Instructions retired counter that measures the instruction execution rate of the underlying processor hardware. This is an encouraging development. While many of the Intel hardware performance counters require esoteric knowledge of the hardware that few people outside specialists in the field in Intel possess, the basic hardware performance counters in the

architectural counter set are suitable for use by a much broader population of performance engineers and analysts. Further instrumenting the Windows OS Scheduler to grab a measure of instructions executed by the hardware at the time of a context switch would be a valuable addition to the QueryThreadCycleTime function in some future release of Windows.

There are currently two serious barriers remaining that limit wider adoption of the Intel hardware counters to augment the OS view of thread execution time. In the current hardware architecture, there is no reservation protocol that ensures that once a performance monitoring application such as vTune programs one of the special purpose Performance Monitoring Registers (PMRs) that those PMRs will stay in that mode. Today, it is possible for a 2nd performance monitoring application to change the way PMRs are programmed out from under the original application. Windows 7 actually provides a centralized facility to program the PMRs, but so long as programs like Intel's vTune continue to program the PMRs directly, this is not a wholly satisfactory solution. A reservation protocol at the hardware level would force every monitoring program that wanted to access the hardware performance counters to opt into a single, Windows-orchestrated sharing scheme.

A second obstacle is what to do about the limited number of Performance Monitoring Registers that exist – the number of possible performance counters far exceeds the number of PMRs available in the hardware to access them – and how to serve them in a virtualization environment. On the NUMA servers that most data centers run with virtualization, the hardware counters are potentially very useful to optimize the scheduling of guest machines. Similarly, massively parallel processing programs executing inside virtual machine guests might want to optimize their scheduling of parallel threads based on hardware counters. How to share the limited number of Performance Monitoring Registers that are available across the VM Host and its guest machine remains an open issue.