The mystery of the scalability of a CPU-intensive application on an Intel multi-core processor with HyperThreading (HT).

By Mark Friedman

http://performancebydesign.blogspot.co.uk/

This blog entry comes from answering the mail (note: names and other incriminating identification data were redacted to protect the guilty):

A Reader writes:

"My name is … and I am a developer for a software company. I encountered something I didn't understand profiling our software and stumbled across your blog. Listen, hot-shot, since you are such an expert on this subject, maybe you can figure this out.

Our software is very CPU-intensive, so we use QueryPerformanceCounter() (QPC) and QueryPerformanceFrequency() (QPF) to get the CPU cycles. Hence, if someone mistakenly commits some change to slow it down, we can catch it. It works fine until one day we decided to fire up more than one instance of the application. Since a large part of our code is sequential and could not be parallelized, by firing up another instance, we can use the other cores that are idle when one core is executing the sequential code. We found the timing profile all messed up. The time difference can be 5x difference for part of the code. Apparently QueryPerformanceCounter() is counting wall time, not CPU cycles. BTW, I have a quad-core hyper-threaded i7 PC.

Then I wrote this small bit of code (at the end of this email) to test QueryPerformanceCounter(), GetProcessTimes() (GPT) function AND QueryProcessCycleTime() function. If I run the code solo (just one instance), we get pretty consistent numbers.

However, if I run 10-instances on my Intel 4-way with HyperThreading enabled (8-logical processors) machine, all three methods (QueryPerformanceCounter(),GetProcessTimes() and QueryProcessCycleTime() report random numbers.

# of Concurrent Processes	1	10
QueryPerformanceCounter time	21.5567753	39.6047058
GetProcessTimes (User)	21.4501375	39.4214527
QueryProcessCycleTime	77376526479	141329606436

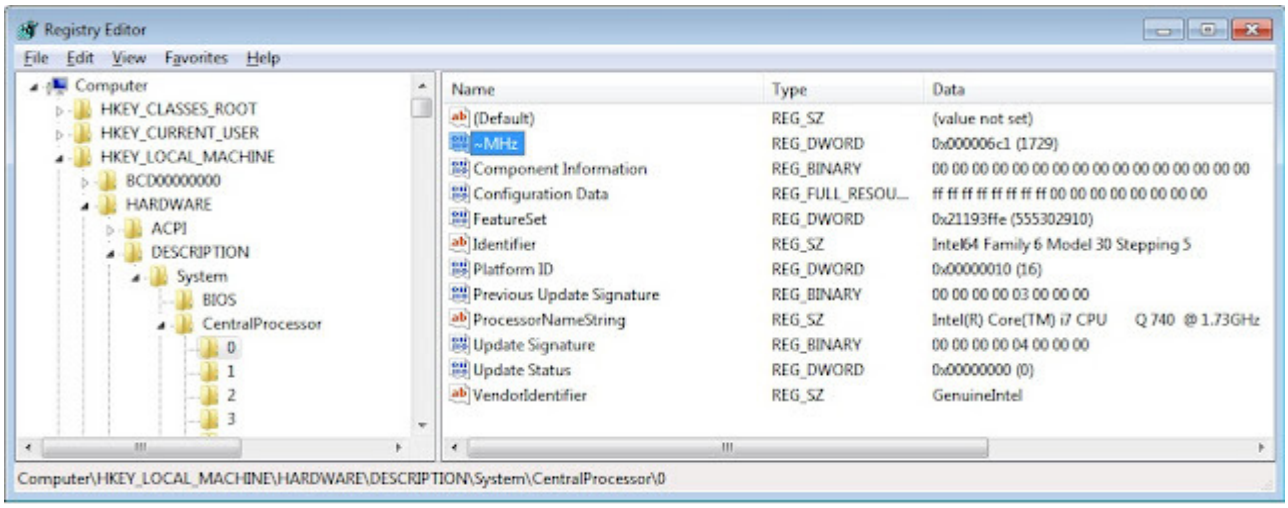
Can you please help me understand what is going on here?

Signed,

-- Dazed and confused.

Mark's Response (pre-amble):

Note that the QPC and GetProcessTimes are reported in seconds. The QPCT times are in cpu cycles, which are model-dependent. You can use the QueryProcessorFrequency() API call to get the processor speed in cycles/second. Or check the ~MHz field in the Registry as illustrated in Figure 1. In my experience, that will report a number similar to QFC().



Swallowing deeply, I humbly replied:

Dear Dazed,

I see that you are confused, but there is a very good explanation for what you are seeing.

If I understand this correctly, the app runs for 21 seconds in a standalone environment. When you run 10 processes concurrently, the app runs for 39 seconds.

You are correct, QueryPerformanceCounter() (QPC in the remainder of this article) is counting wall clock time, not cycles. Any time that the thread is switched out between successive calls to QPC() would be included in the timing, which is why you obtained some unexpected results. You can expect thread pre-emption to occur when you have 10 long-running threads Ready To Run on your 8-way machine.

But I think there is something else going on in this case, too. Basically, you are seeing two multiprocessor scalability effects – one due to HyperThreading and one due to more conventional processor queuing. The best way to untangle these is to perform the following set of test runs:

1. run standalone
2. run two concurrent processes
3. run four concurrent processes
4. run 8 concurrent processes
5. then, keep increasing the # of concurrent processes and see if a pattern emerges.

Using GetProcessTimes is not very precise -- for all the reasons discussed on my blog (see link at beginning of this article), it is based on sampling – but for this test of a long running processing (in this example: ~ 21 seconds), the precision is sufficient. Since the process is CPU-intensive, this measurement is consistent with elapsed time as measured using QPC, but that is just pure luck, because, as it happens, the program you are testing does not incur any significant IO wait time. My recommendation is to try calling QueryThreadCycleTime (QTCT) instead; it is a better bet, but it is not foolproof either (as I have discussed on my blog). Actually, I recommend you instrument the code using the Scenario class library that we put up on Code Gallery sometime back, see http://archive.msdn.microsoft.com/Scenario. It calls both QPC and QTCT in-line during thread execution.

I personally don't have a lot of direct experience with QueryProcessCycleTime (QPCT), but my understanding of how it works could make it problematic when you are calling it from inside a multi-

threaded app. If your app runs mainly single-threaded, it should report CPU timings similar to QTCT.

The Mystery Solved.

Within a few days, the intrepid Reader supplied some additional measurement data, per my instructions. After instrumenting the program using the Scenario class, he obtained measurements from executing 2, 4, 6, 8, etc. processes in parallel, up to running 32 processes in parallel on this machine. Figure 2 summarizes the results he obtained:

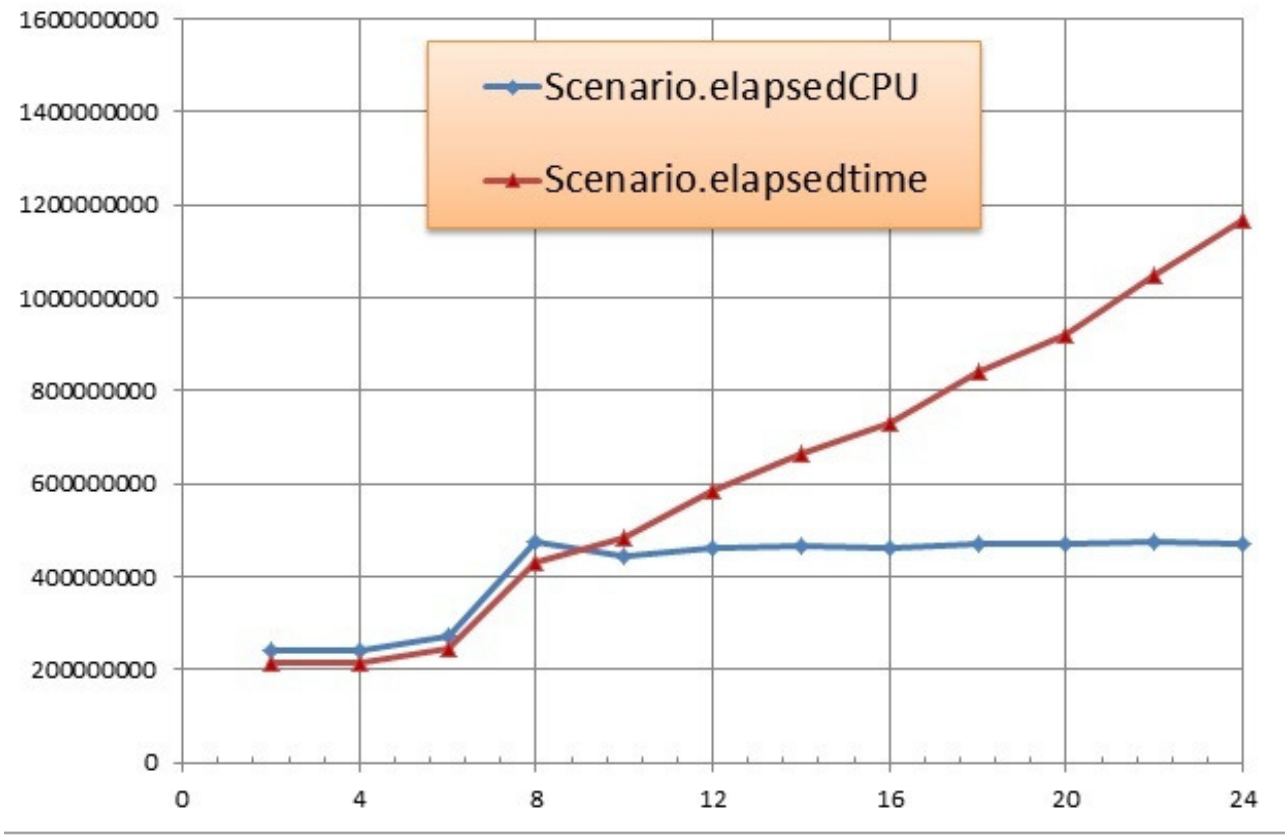


Figure 2. Elapsed times and CPU times as a function of the number of concurrent processes. The machine environment is a quad-core Intel i7 processor with HT enabled (the OS sees 8 logical processors).
(Note that I truncated the results at 24 concurrent processes to create this graphic to focus on left side of the chart. Beyond 24 processes, both line graphs continue consistently along the same pattern indicated. CPU time remains flat and elapsed time continues to scale linearly.)

To be clear about the measurements being reported, the **Elapsed** property of the Scenario object is the delta between a QPC() clock timer issued at Scenario.Begin() and one issued at Scenario.End(). It is reported in standard Windows 100 nanosecond timer ticks. The **ElapsedCpu** property is the delta between two calls to QTCT, made immediately *after* the QPC call in the Scenario.Begin method and just *before* Scenario.End calls QPC. It is also reported in standard Windows timer ticks. Respectively, they are the elapsed (wall clock time) time and the CPU

thread execution time for the thread calling into the embedded Scenario instrumentation library.

Looking at the runs for two and four concurrent processes, we see both elapsed time and CPU time holding constant. In terms of parallel scalability, this application running on this machine with 4 physical processor cores scales linearly for up to four concurrent processes.

We also see that for up to 8 concurrent processes, elapsed time and CPU time are essentially identical.

For more than 8 processes, the CPU time curve flattens out, while elapsed time of the application increases linearly with the number of concurrent processes. Running more processes than logical processors does nothing to increase throughput; it simply creates conventional processor queuing delays. The fact that the workload scales linearly up to 32 concurrent processes is actually a very positive result. The queuing delays aren't exponential, there is no evidence of locking or other blocking that would impede scalability. These results suggest that if you had a 32-way (physical processor) machine, you could run 32 of these processes in parallel very efficiently.

The HyperThread (HT) effects are confined to the area of the elapsed and CPU time curves between 4 and 8 concurrent processes. Notice with six processes running concurrently, CPU time elongates only slightly – the benefit of the HT hardware boost is evident here. However, running eight concurrent processes leads to a sharp spike in CPU time – this is a sign that 8 concurrent processes saturate the four processor cores. The HT feature no longer effectively increases instruction execution throughput.

The CPU time curve flattening out after 8 concurrent processes suggests that at least some of that CPU time spike at 8 concurrent processes is due to contention for shared resources internal to the physical processor core from threads running in parallel on logical (HT) processors.

HT is especially tricky because the interaction between threads contending for shared resources internal to the physical processor core is very workload dependent. Running separate processes in parallel eliminates most of the parallel scalability issues associated with "false" sharing of shared cache lines because each process is running in its own dedicated block of virtual memory. What is particularly nefarious about concurrently executing threads "false sharing" of cache lines is that it subjects executing to time-consuming delays due to writing back updated cache lines to RAM and re-fetching them. In theory, these delays can be minimized by changing the way you allocate your data structures. In practice, you do not have a lot of control over how memory is allocated if you are an application developer using the .NET Framework. On the other hand, if you are writing a device driver in Windows and working in C++, false sharing in the processor data cache is something you need to be careful to address.

I trust this further exploration and explanation will prove helpful, and I look forward to seeing a check in the mail from your company to me for helping to sweep aside this confusion.

Signed,

-- Your obedient servant, etc., etc.

markf@demandtech.com