Multiprocessor scalability in Microsoft Windows NT/2000

Abstract.

This paper provides an overview of the multiprocessing support in the Microsoft Windows NT/2000 operating system, with an emphasis on scalability and other capacity planning issues. It also discusses specific features of the Intel P6 architecture that provide the hardware basis for large scale multiprocessing systems. As a shared memory multiprocessing implementation, Windows NT/2000 is predictably vulnerable to saturation on the shared memory bus. Processor hardware measurements that can illuminate memory bus contention when it appears are also described and discussed.

Introduction

The specific type of multiprocessing Windows NT/2000 implements using Intel P6 processors (the Pentium Pro, Pentium II, and Pentium III models) is generally classified as *shared-memory* multiprocessing. In this type of configuration, the processors operate totally independent of each other. But they do share a single copy of the operating system and they share access to main memory (i.e., RAM). A typical dual processor shared-memory configuration is illustrated in Figure 5.1. Notice the illustration shows two P6 processors, which contain dedicated Level 2 caches. (They each have separate built-in Level 1 caches, too.) A two-way configuration, as illustrated, simply means having twice the hardware – two identical sets of processors, caches, and internal buses. Similarly, a four-way configuration means having four of everything. Having duplicate caches is designed to promote scalability since cache is so fundamental to the performance of pipelined processors.

The processors also share a common bus that is used to access main memory locations. This, obviously, is not so scalable. And, according to experienced hardware designers, this shared component is precisely where the bottleneck in shared-memory designs often is.

Operating system support for multiprocessing. Each processor in a multiprocessor is capable of executing work independently of the other. Separate, independent threads may be dispatched, one per processor, and run in parallel. Only one copy of the Windows 2000 operating system is running, controlling what runs on all the processors. From a performance monitoring perspective, you will see multiple instances of the processor Object reported in both Taskman (as illustrated below in Figure 2) and Perfmon (see Figure 3).

The specific type of multiprocessor support offered beginning in Windows NT 4.0 is known as *symmetric multiprocessing*, often abbreviated as SMP. *Symmetric* in this context means that every thread is eligible to execute on any processor. Prior to NT 4.0, Windows NT supported only *asymmetric multiprocessing*. Interrupts could only be processed in an NT 3.5x machine on CPU 0. When they are present, CPUs 1, 2, 3, can only run user and kernel code, never Interrupt Service Routines (ISRs) and Deferred Procedure Calls (DPCs). This asymmetry ultimately limits the scalability of NT 3.5x multiprocessor systems because the CPU 0 engine is readily overloaded under some workloads, while the remaining microprocessors are idling. In an SMP, in theory, all the microprocessors should run out of capacity at the same time. One of the key Microsoft development projects associated with the NT 4.0 release

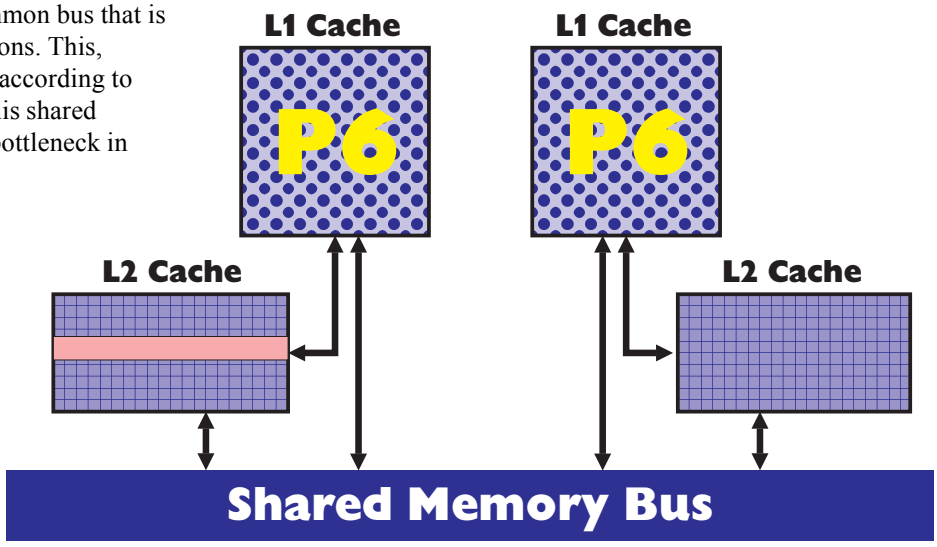

Figure 1. A shared-memory multiprocessor. Each processor in a multiprocessor has access to dedicated Level 1 cache and Level 2 caches. Access to system RAM, in contrast, is shared via a common system bus.

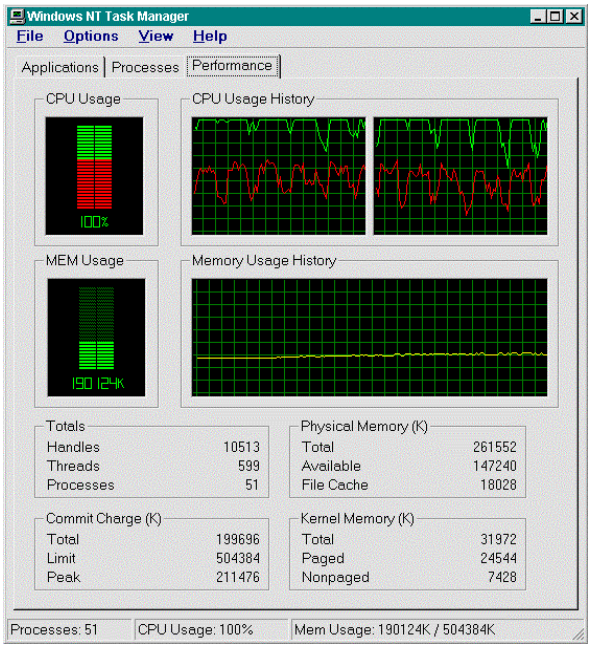

FIGURE 2. MEASUREMENT SUPPORT FOR MULTIPROCESSORS IN TASK MANAGER.

was changes to the kernel to support SMPs. In addition, the Windows NT development team fine-tuned the OS code to run much better in a multiprocessing environment. Windows 2000 also incorporates further improvements inside the operating system to boost performance on large n-way multiprocessor configurations.

The symmetric multiprocessing (SMP) support available in Windows NT 4.0 and above allows any processor normally to process any interrupt, as illustrated in Figure 3. This performance data illustrates a two-way symmetric multiprocessor system running NT 4.0 Server. (Be careful, the vertical axis scale was adjusted down to a maximum of thirty to make the chart data easier to decipher.) The two processor instances of % Privileged Time and % Interrupt Time Counters are shown. The processing workload is roughly balanced across both processors, although the load does bounces back and forth a bit, depending on what threads happen to be ready to run. As should be evident from this picture, threads are dispatched independently in Windows 2000 so that it is possible for a multithreaded application to run in parallel on separate processors.

Figure 3 shows the amount of CPU time consumed servicing interrupts on the two processors. The amount of time spent processing interrupts per processor is roughly equal, though there is some degree of variation that occurs naturally. This is not always the most efficient way to process interrupts. Having one type of ISR or DPC directed to a single processor can have a positive impact on performance if the processor that runs the DPC code, for instance, is likely to be able to cache it, rather than being forced to fetch it from memory. Similarly, The Win2K scheduler tries to dispatch a ready thread on the same processor where it recently ran for that very same reason. A thread is said to have an *affinity* for the processor where it was most recently dispatched. Processor affinity in Windows 2000 thread scheduling is discussed below.

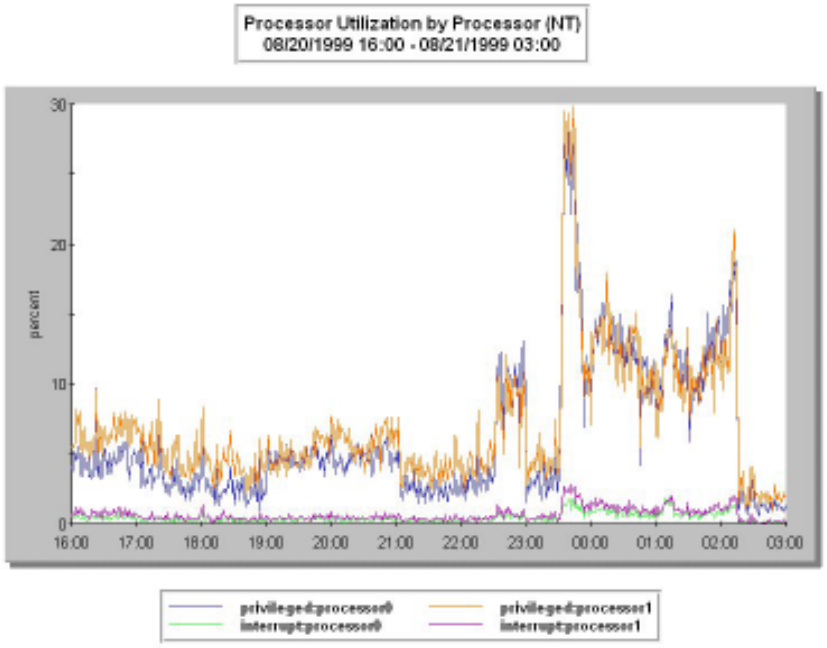

FIGURE 3. SYMMETRIC MULTIPROCESSING IN WINDOWS 2000 AND NT 4. OPERATING SYSTEM PRIVILEGED THREADS AND INTERRUPT SERVICE ROUTINES (ISRS) ARE ELIGIBLE TO BE DISPATCHED ON ANY AVAILABLE PROCESSOR.

Processor affinity. Logically, the structure of the Windows 2000 Thread Scheduler Ready Queue and the relative priority of threads waiting to execute is identical whether Win2K is executing on an single processor or on multiple processors. The main difference is that multiple threads can run *concurrently* on a multiprocessor, a little detail that leads to many complications. The Win2K Scheduler, in turn, selects the highest priority waiting thread to run on each available processor. In addition, certain execution threads may have an *affinity* to execute on specific processors. Win2K supports both *hard affinity*, where a given thread is eligible to run *only* on specific processors, and *soft affinity*, where Win2K favors scheduling specific threads on specific processors, usually for performance reasons.

Hard affinity is specified at the process and thread level using a processor affinity mask. The Win32 API calls to

accomplish this are straightforward. First, a Thread issues a *GetProcessAffinityMask* call referencing a process handle, which returns a 32-bit *SystemAffinityMask.* Each bit in the *SystemAffinityMask* represents a configured processor. Then, the Thread calls ***SetProcessAffinityMask*** with a corresponding 32-bit affinity mask that indicates which processors Threads from the process can be dispatched on. Figure 4, which illustrates the use of this function in Taskman, allows you to set a process's affinity mask dynamically, subject to the usual security restrictions that allow Taskman to operate only on foreground-eligible processes by default. There is a corresponding call to *SetThreadAffinityMask* override the process settings for specific threads. Once hard affinity is set, threads are only eligible to be dispatched on specific processors.

Suppose that following an interrupt an application program thread becomes Ready to run and there are multiple processors that are idle. First, Win2K must chose among available processors to select the processor for a Ready thread to run on. This decision is based on performance. If the thread was previously dispatched within the last Scheduler quantum (or timeslice), Win2K attempts to schedule the thread on that processor, so long as the current thread has a higher priority than the thread that is already running there. This is known as *soft affinity*. If the desired processor is currently busy with a higher priority task, Win2K is willing to schedule the waiting thread on a different processor. By scheduling the thread back on the same processor where it ran last, Win2K hopes that a good deal of the thread's code and data from the previous execution interval are still present in that processor's cache. The difference in instruction execution rate between a cache "cold start" when a thread is forced to fault its way through its frequently accessed code and data and a "warm start" when the cache is preloaded can be substantial.

Shared-memory multiprocessor scalability

Shared-memory multiprocessors running SMP operating systems are the most common breed of multiprocessor. A great deal is known about hardware performance in this context. Any multithreaded application will likely benefit from having a choice of processors to run on – where any thread can run on any available processor in an SMP. Even single threaded applications may benefit, since multiple applications can run in parallel. Because a dual processor system running Windows 2000 can dispatch two threads at a time, not just one, it seems reasonable to assume the dual processor configuration is twice as powerful as having a single engine to do the work. To see why this isn't exactly so, we will need to investigate a few aspects of shared-memory multiprocessor *scalability*.

Shared-memory multiprocessors have some well known scalability limitations. They seldom provide perfect linear

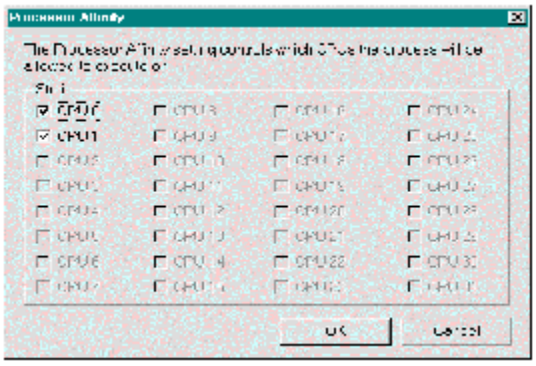

FIGURE 4. SETTING A PROCESS'S PROCESSOR AFFINITY MASK USING TASKMAN.

scalability. Each time you add a processor to the system, you do get a corresponding boost in overall performance, but with each additional processor the boost you get tends to diminish. It is also possible to reach a point of diminishing returns where adding another processor actually reduces overall capacity. In this section we discuss several fundamental issues that impact multiprocessing scalability, including:

- the overhead of multiprocessor synchronization,
- multiprocessor-related pipeline stalls that are caused by cache coherence conflicts, and
- cycles wasted by code executing spin locks.

Understanding these sources of performance degradation in a shared-memory multiprocessor will help us in examining and interpreting the extensive processor utilization measurements that are available in Windows NT/2000.

One thing to be careful about is that the processors in an SMP may look very busy, but if you are able to look inside the processor, you may find they are not performing as much productive work. The way to look inside is to use the Pentium Counters [1]. On a simple, single engine system, Instructions retired/sec, the internal P6 measure of Instruction Execution Rate (IER), generally tracks processor % Processor Time very well. Scalability issues mean that internal IER and external processor busy measures can no longer be expected to correspond in predictable ways for more complicated multiprocessors.

Figure 5, taken from a two-way multiprocessor, illustrates this essential point. Figure 5a shows the Task Manager histogram of processor utilization on this machine. Notice that both engines are running at near 100% utilization. This configuration contains two 200 MHz Pentium Pro machines. Remember, on a uniprocessor a good Rule of Thumb is to expect performance at or near 2 Cycles per instruction (CPI). This translates into capacity of about 100,000,000 instructions per second on a 200 MHz P6. Figure 5b shows actual measurements for one of these engines at only 12,345,000 instructions per second.

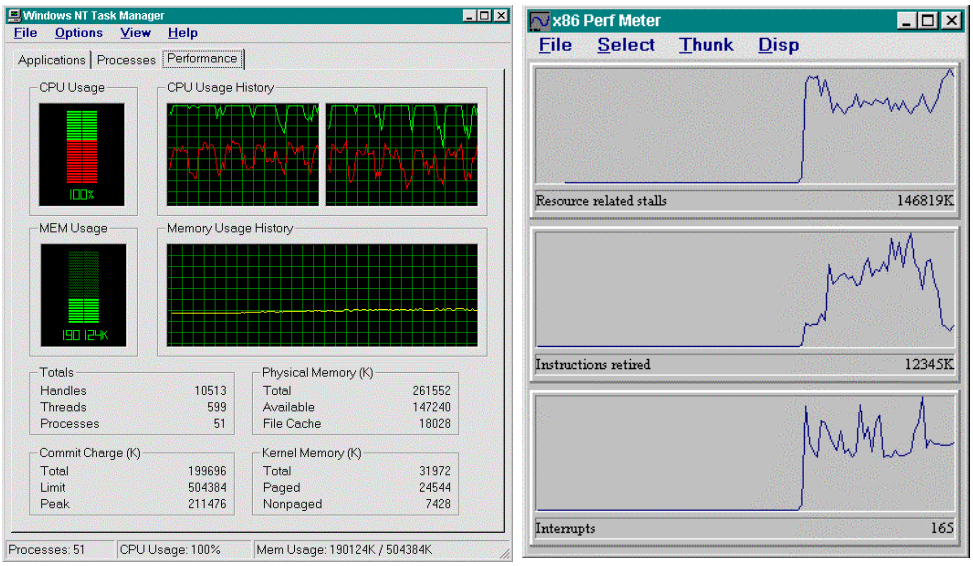

FIGURE 5. MULTIPROCESSOR SCALABILITY ISSUES FORCE YOU TO LOOK INSIDE THE PROCESSOR AT INTERNAL MEASURES OF EXECUTION RATE BECAUSE IT MAY NO LONGER CORRESPOND TO EXTERNAL PROCESSOR BUSY MEASUREMENTS. AT LEFT IS A *TASK MANAGER* HISTOGRAM OF PROCESSOR UTILIZATION SHOWING THAT BOTH ENGINES ARE RUNNING AT NEAR 100% UTILIZATION. THIS CONFIGURATION CONTAINS TWO 200 MHZ PENTIUM PRO MACHINES. FIGURE 5.B AT RIGHT SHOWS P6 MEASUREMENTS FOR ONE OF THESE ENGINES RUNNING AT ONLY 12,345,000 INSTRUCTIONS PER SECOND, OR A CPI OF ABOUT 16.7. (A GOOD TARGET CPI ON A UNIPROCESSOR WAS ABOUT 2.0.) OVER 146 MILLION RESOURCE-RELATED STALLS ARE BEING MEASURED CONCURRENTLY.

This machine is delivering only about 12% of its rated performance – notice over 146 million resource-related stalls, too. Making sure that an expensive 4 or 8-way multiprocessor Server configuration that you purchased is performing up to its capacity is not a trivial affair.

Speed-up factors. When we talk about *scalability* in the context of either multiprocessors or parallel processors, we are referring to our basic desire to harness the power of more than one processor to solve a common problem. The goal of a dual processor design is to apply double the CPU horsepower to a single problem and solve it in one half the time. The goal of a quad processor is to apply quadruple the processing power and solve problems in one quarter the time. You get the idea. The term *speed-up factor* refers to our expectation that a multiprocessor design will improve the amount of time it takes to process some workload. If a multiprocessing design supported a speed-up factor of 1, then two processors would provide fully twice the power of a single engine. This would be perfect linear scalability, something shared-memory multiprocessors are just not capable of. A reasonable expectation is for a speed up factor in the range of about 0.85. This means that two Intel processors tied together would only be able to function at about 85% efficiency. Together, the two would provide 1.7 times the power of standalone processor. An improvement, but certainly a more marginal and less cost-effective one. It turns out that Windows NT running on Intel hardware provides MP scalability in that range.

What happens when you add a third, fourth, or even more processors? The best models of shared memory multiprocessor performance suggest that the machines get progressively even less efficient as you add more processors to the shared bus. However, with proper care and feeding, multiprocessor configurations with quite good scalability can be configured, although this requires a fair amount of skill and effort.

Microsoft reported at a presentation at CMG 1996 that the symmetric multiprocessing support built for Windows NT version 4.0, in fact, sported a speed-up factor of 0.85. Figure 6 compares the theoretical prospects for linear speed-up in a multiprocessor design to the actual (projected) scalability of Windows NT version 4.0, based on the actual measurements reported by Microsoft. The projection used here is a guess, of course, and your mileage may vary, but it is based on the formula Gunther [2] recommends for predicting multiprocessor scalability. Actual performance is very workload-dependent, as we will discuss below. Figure 6 illustrates that actual performance of a multiprocessor running Windows NT falls far short of the ideal linear speed-up. In fact, beyond four multiprocessors, the projection is that adding more engines hardly boosts performance at all. Many published benchmark results of Windows NT multiprocessors evidence similar behavior, as in, for example, Figure 7, benchmark results that Intel published on its web site in 1998. (Look carefully — the results depict measurements of 1, 2and 4-way systems.) Results such as these suggest that the theoretical model has at least some underlying validity.

Notice that after a certain point (> 12 processors), adding additional engines actually degrades overall performance, according to the theoretical model. It is worth noting that Windows NT 4.0 multiprocessor scalability is rather typical of general-purpose operating systems – no worse, no better. MVS, IBM's flagship multiprocessing operating system, achieved a similar 0.85 scalability up until it was re-architected for massively-parallel processing. To achieve anywhere near linear scalability requires highly engineered, special purpose

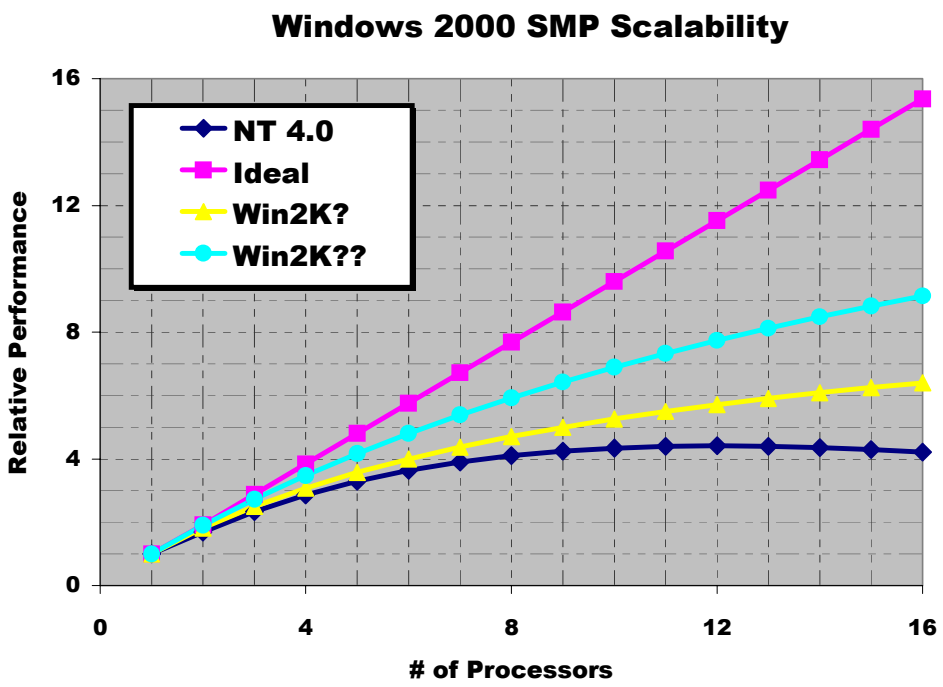

FIGURE 6. THEORETICAL LINEAR SCALABILITY OF A MULTIPROCESSOR COMPARED TO ACTUAL PROJECTED SCALABILITY OF WINDOWS NT 4.0, BASED ON MEASUREMENTS TAKEN AND REPORTED BY MICROSOFT IN 1996. THE PROJECTION USES A FORMULA RECOMMENDED BY GUNTHER [1], BUT IS CONSISTENT WITH A NUMBER OF PUBLISHED BENCHMARKING RESULTS. FOR INSTANCE, SEE THE ARTICLE ON 8-WAY SCALABILITY IN THE SEPTEMBER 1998 WINDOWS NT MAGAZINE AT HTTP://WINNTMAG.COM/MAGAZINE/ARTICLE.CFM?ISSUEID=58&ARTICLEID=3781. WINDOWS 2000 INCORPORATES FURTHER MULTIPROCESSOR SCALABILITY ENHANCEMENTS, BUT IT IS NOT YET CLEAR JUST HOW MUCH MORE SCALABLE WIN2K IS.

parallel processing hardware and complementary operating system services to match.

Windows 2000 incorporates some further enhancements designed to improve multiprocessor scalability. Microsoft implemented a new HAL function called queued spin locks that exploits a new Intel instruction on the Pentium III. It is not clear just how much this new function will help on large scale 8 and 16-way machines. Figure 6 suggests two possibilities, reflecting a relatively marginal increase in multiprocessor scalability to 0.90 or possibly even to 0.95.

To summarize this discussion so far, it is simply not possible to string processor after processor together and double, triple, quadruple, etc., the amount of total processing power available. The principal obstacle of shared-memory multiprocessor designs, which are quite simple from the standpoint of the programmer, is that they typically encounter a bottleneck in accessing shared-memory locations using the shared system memory bus. To understand the nature of this bottleneck, let's proceed to a discussion of the sources of performance degradation in a multiprocessor.

Serializing instructions. The first noticeable multiprocessor effect is the performance impact of *serializing* LOCKed instructions. Instructions coded with the LOCK prefix are guaranteed to run uninterrupted and gain exclusive access to the designated memory locations. Locking the shared-memory bus delays any threads executing on other processors that need access to memory. There are, in addition, a number of hardware-oriented operations that are performed by the operating system that implicitly serialize by locking the shared-memory bus on an Intel shared-memory multiprocessor. These include setting the active Task State Segment (TSS), which is performed during a context switch of any type. Intel hardware also automatically serializes updates of the Page Directory Entries and Page Table Entries that are used in translating virtual memory addresses to real memory locations. (This impacts the page replacement algorithm that Win2K uses on Intel multiprocessors, as discused in [3].)

Intel documentation [4] describes some specific serializing instructions that force the processor executing

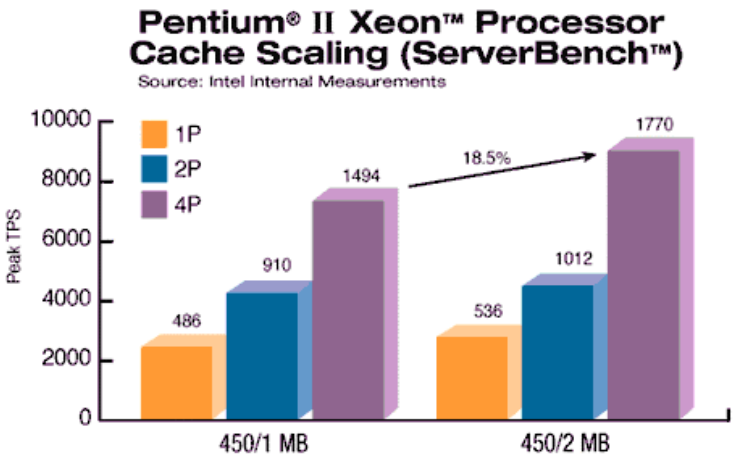

FIGURE 7. MULTIPROCESSING BENCHMARK RESULTS PUBLISHED BY INTEL IN 1998. LOOK CAREFULLY — THE RESULTS DEPICT MEASUREMENTS OF 1, 2AND 4-WAY SYSTEMS, COMPARING SIMILARLY CONFIGURED SYSTEMS WITH 1 AND 2 MB OF LEVEL 2 CACHE.

these instructions to drain the pipeline before executing the instruction. Following execution of the serializing instruction, the pipeline is started up again. These serializing instructions include privileged operations that move values into internal Control and Debug Register, for example. Serializing instructions have the effect on the P6 of forcing the processor to re-execute out of order instructions, for example.

The performance impact of draining the instruction execution pipeline ought to be obvious. Current generation P5 and P6 Intel processors are pipelined, superscalar architectures. The performance impact of executing an instruction serialized with the LOCK prefix includes potentially stalling the pipelines of other processors executing instructions until the instruction that requires serialization frees up the shared-memory bus. This can be a fairly substantial performance hit, too, which is solely a consequence of running in a multiprocessor environment. The cost of both sorts of instruction serialization contribute to at least some of the less than linear scalability that we can expect in a multiprocessor. How much is very difficult to quantify, and certainly workload dependent. There is also very little one can do about this source of degradation. Without serializing instructions, multiple processors would simply not work reliably.

A second source of multiprocessor interference is *interprocessor signaling* instructions. These are instructions issued on one processor to signal another processor, for example, to wake it up to process a pending interrupt. By its very nature, interprocessor signaling is quite expensive, in performance terms.

Cache effects. Effective on-board CPU caching is critical to the performance of pipelined processors[5]. Intel waited to introduce pipelining with its 486 chips until there was enough real estate available to include an on-board cache. It should not be a big surprise to learn that one secondary effect of multiprocessor coordination and serialization is that it makes caching less effective. This, in turn, serves to slow down the processor's instruction execution rate. In order to understand why SMPs impact cache effectiveness, we will take a detour into the realm of cache coherence in the next section. From a configuration and tuning perspective, one intended effect of setting up an application to run with processor affinity is to improve cache effectiveness and increase the instruction execution rate. Direct measurements of both instruction execution rate and caching efficiency, fortunately, are available via the Pentium Counters. Unfortunately, the Pentium Counter support Microsoft provides in the NT 4.0 Resource Kit falls short of the precision tool that MP configurations require. Moreover, Microsoft no longer provides a means to gather Pentium statistics in Windows 2000.

Spin locks. If two threads are attempting to access the same serializable resource, one thread will acquire the lock, which then blocks the other one until the lock is released. A block of code guarded by some synchronization or locking structure is called a *critical section*. (The generic name should not be confused with the Win32 API function which provides platform independent locking services for critical sections.) Problem: what should the thread that is blocked waiting on a critical section do while it is waiting? An application program in Windows 2000 is expected to use Win32 serialization runtime services that puts the application to sleep until notified that the lock is available. Win32 serialization services arrange multiple threads waiting on a shared resource in a FIFO queue so that the queueing discipline is fair. This suggests that a key element of designing an application to run well on a shared-memory multiprocessor is to *minimize* the amount of processing time spent inside critical sections. The shorter the time spent executing inside a locked critical section of code, the less time other threads are blocked waiting to enter it. Much of the re-engineering work Microsoft did on NT 4.0 and again in Windows 2000 was to redesign the critical sections internal to the OS to minimize the amount of time kernel threads would have to wait for shared resources.

When critical sections are designed appropriately, then threads waiting on a locked critical section should not have long to wait. Furthermore, while a thread is waiting on a lock, there may be nothing else for it do. For example, a thread waiting on the Win2K Scheduler lock can perform no useful work until it has successfully acquired that lock. For example, consider a kernel or device driver with OSD privileges that is blocked waiting on a lock. The resource the thread is waiting for is required before any other useful work on the processor can be performed. The wait can be expected to be of very short duration. Under these circumstances, the best thing to do may be to loop back and test for the availability of the lock again. Code that tests for availability of a lock that finally enters a critical section and sets the lock using a serializing instruction. If the same code finds the lock is already set (presumably by a thread running on a *different* processor), there is nothing to do on a shared memory multiprocessor other than retry the lock again. The entry code simply branches back to retest the lock. This coding technique is known as a *spin lock*. If you are able to watch this code's execution, it appears to be stuck in a very tight loop of just a few instructions — until the lock requested is finally available.

Spin locks are used in many, many different places throughout the operating system in Windows 2000 because operating system code waiting for a critical section to be unlocked often has nothing better to do during what is, hopefully, a very short waiting period than retest the lock. For example, device drivers are required to use spin locks to protect data structures if there is any possibility that

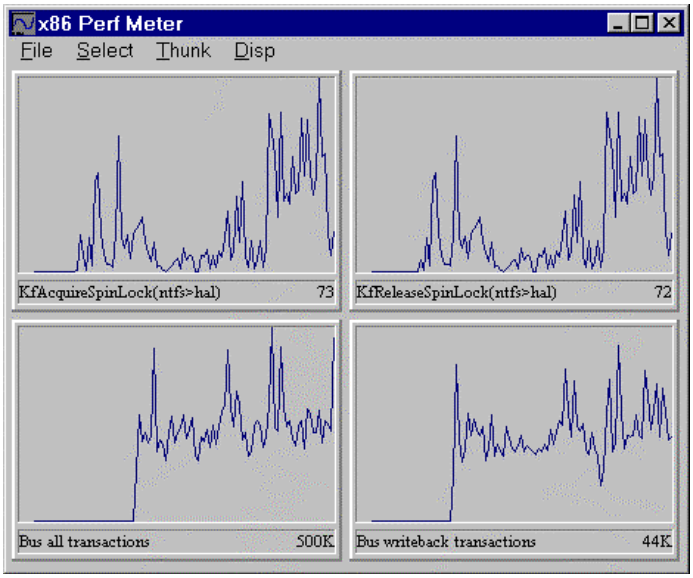

FIGURE 8. THE THUNK FUNCTION IN THE X86 PERF METER APPLICATION IN THE RESOURCE KIT CAN BE USED TO MONITOR SPIN LOCK ACTIVITY. IN THIS EXAMPLE, THE NTFS.SYS FILE SYSTEM DRIVER MODULE IS CALLING INTO HAL.DLL (THE TARGET MODULE). THE KFACQUIRESPINLOCK AND KFRELEASESPINLOCK HAL FUNCTIONS ARE BEING MONITORED. NTFS FILE SYSTEM REQUESTS THAT MODIFY FILE SYSTEM USE HAL SPIN LOCKS FUNCTIONS TO PROTECT CRITICAL SECTIONS OF CODE.

multiple threads could be active consurrently on a symmetric multiprocessor where device interrupts are eligible to be processed on any processor. Windows 2000 provides a standard set of spin lock services for Device Drivers to use in Interrupt Service Routines and kernels threads outside of ISRs. (See the DDK documentation on *KeInitializeSpinLock*, *IoAcaquireCancelSpinLock*, and related services for more detail on these services.) These standard services allow Device Drivers written in C language to be portable across versions of Windows NT running on different hardware.

In Windows NT version 4.0, you can use the Thunk function in the x86 Perf Meter application in the Resource Kit (pperf.exe – the same application used to access the Pentium Counters) to witness spin lock activity. For example, from the Thunk menu, select the ntfs.sys file system driver module, using hal.dll as the target module. Then select the *KfAcquireSpinLock* and *KfReleaseSpinLock* for monitoring, as illustrated in Figure 8. If you then generate some ntfs file system requests, like emptying the Recycle bin, you will observe ntfs driver code using the HAL spin lock function to protect critical sections of code. Consider a 2, 4 or 8-way multiprocessor with an ntfs file system. ntfs.sys functions can be executed on any processor where there is an executing thread that needs access to disk files. In fact, it is likely that ntfs functions will execute concurrently (on more than one processor) from time to time. ntfs.sys uses HAL spin lock functions to protect critical sections of code, preserving the integrity of the file system in a multiprocessor environment.

Spin lock code is effectively dormant when it is run on a single processor system, but consumes a significant number of processor cycles on a multiprocessor. Again, the use of spin locks is, to a large degree, unavoidable. The performance implication of spin locks is that processor utilization increases, but no useful work is actually being performed. Here's where simple measures of processor utilization are misleading. Win2K client applications care about throughput and response time, which may be degraded on a multiprocessor even as measurements of CPU utilization look rosy.

The combined impact of serializing instructions, interprocessor signaling, diminished cache effectiveness, and the consumption of processor cycles by spin lock code serve to limit the scalability of shared-memory multiprocessors in Windows 2000 and other operating systems. Furthermore, each additional processor that is added to the configuration amplifies these multiprocessor scalability factors. These scalability factors make sizing, configuring, and tuning large scale n-way Win2K multiprocessors a very tricky business. Just how tricky should become more apparent after a consideration of the cache coherence problem in the next section.

Cache coherence

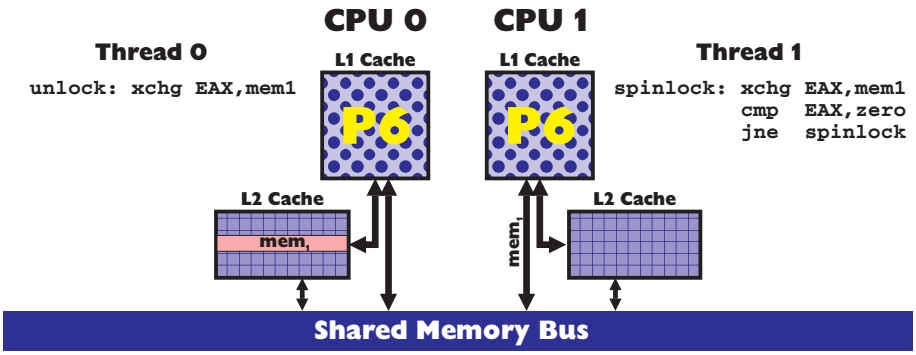

FIGURE 9. TWO THREADS OPERATING ON THE SAME MEMORY LOCATION CONCURRENTLY LEAD TO PROBLEMS MAINTAINING THE COHERENCE OF INFORMATION STORED IN LOCAL PROCESSOR CACHES. IN THIS EXAMPLE, THREAD 0 EXECUTING ON CPU 0 IS ABOUT TO RESET A LOCK WORD AT LOCATION MEM_1, RESIDENT IN ITS LEVEL 2 CACHE. MEANWHILE, THREAD 1 EXECUTING ON CPU 1 IS ATTEMPTING TO SET THE SAME LOCK WORD AT LOCATION MEM_1 TO ENTER THE CRITICAL SECTION THREAD 0 IS ABOUT TO EXIT. HOW THE UPDATE TO AT LOCATION MEM_1 PERFORMED IN LOCAL CACHE ON CPU 0 IS PROPAGATED TO CPU 1 IS AN EXAMPLE OF THE CACHE COHERENCE PROBLEM.

The cache effects of running on a shared-memory multiprocessor are probably the most salient of the factors limiting the scalability of this type of computer architecture. The various forms of processor cache, including Translation Lookaside Buffers (TLBs), code and data caches, and branch prediction tables, all play a critical role in the performance of pipelined machines like the Pentium, Pentium Pro, Pentium II, and Pentium III. For the sake of performance, in a multiprocessor configuration each CPU retains its own private cache memory, as depicted in Figure 9. We have seen that multiple threads executing inside the Win2K kernel or running device driver code concurrently can attempt to access the same memory locations. Propagating changes to the contents of memory locations cached locally to other engines that may have their own copies of the same memory is a major issue in designing multiprocessors to operate correctly. This is also known as the *cache coherence* problem in shared-memory multiprocessors. Cache coherence issues also have significant performance ramifications.

Maintaining cache coherence in a shared-memory multiprocessor is absolutely necessary in order for programs to execute correctly. While, for the most part, independent program execution threads operate independently of each other, sometimes they must interact. Whenever they Read and Write common or shared-memory data structures, threads must communicate and coordinate accesses to these memory locations. This necessary coordination inevitably has performance consequences. We will illustrate this side effect by drawing on an example where two kernel threads are attempting to gain access to the Win2K Scheduler Ready Queue simultaneously. As indicated earlier, a global data structure like the Ready Queue that is subject to access from multiple threads executing concurrently on different processors *must* be protected by a lock. Let's look at how a lock word value set by one thread on one processor is propagated to cache memory in another processor where another thread is attempting to gain access to the same critical section.

In Figure 9, Thread 0 running on CPU 0 that has just finished updating the Win2K Scheduler Ready Queue, for example, is about to exit a critical section. Upon exiting the critical section of code, Thread 0 resets the lock word at location **mem$_1$** using a serializing instruction like XCHG. Instead of locking the bus during the execution of the XCHG instruction, the Intel P6 operates instead only on the cache line that contains **mem$_1$**. This is to boost performance. The locked memory fetch and store that the instruction otherwise requires would stall the CPU 0 pipeline. In the Intel Architecture, if the operand of a serializing instruction like XCHG is resident in processor cache in a multiprocessor configuration, then the P6 does not lock the shared-memory bus. This is a form of *deferred write-back caching*, which is very efficient. Not only does the processor cache hardware use this approach to caching frequently accessed instructions and data, but we will see that so do Win2K systems software and hardware cached disk controllers, for example.

In the interest of program correctness, updates made to private cache, which are deferred, ultimately must be applied to the appropriate shared-memory locations *before* any threads running on other processors attempt to access the same information. Moreover, as Figure 9 illustrates, there is an additional data integrity exposure because another CPU can (and frequently does) have the same mem1 location resident in cache. The diagram illustrates a second thread that is in a spin loop trying to enter the same critical section. This code continuously tests the contents of the lock word at **mem$_1$** until it is successful. For the sake of performance, the XCHG instruction running on CPU 1 also operates only on the cache line that contains **mem$_1$** and does not attempt to lock the bus, each time, because that would stall each processor's instruction execution pipeline. We can see that unless there is some way to let CPU 1 know that code running on CPU 0 has *changed* the contents of **mem$_1$**, the code on CPU 1 will spin in this loop forever. The Intel P6 processors solve this problem in maintaining cache coherence using a method conventionally called *snooping*.

Intel MESI snooping protocol. Snooping protocols to maintain cache coherence have each processor listening to the shared-memory bus for changes in the status of cache resident addresses that other processors happen to be operating on concurrently. Snooping requires that processors place the memory addresses of any shared cache lines being updated on the memory bus. All processors listen on the memory bus for memory references made by other processors that affect memory locations that are resident in their private cache. Thus, the term snooping. The term snooping also has the connotation that this method for keeping every processor's private cache memory synchronized can be performed in the background (which it is) without a major performance hit (which is true, but only up to a point). In practice, maintaining cache coherence is a complex process that can interfere substantially with normal pipelined instruction execution and generates some serious scalability issues.

Let's illustrate how the Intel snooping protocol works, continuing with our Ready Queue lock word example. CPU 1, snooping on the bus, recognizes that the update to the **mem$_1$** address performed by CPU 0 *invalidates* its cache line containing **mem$_1$**. Then, because the cache line containing **mem$_1$** is marked invalid, CPU 1 is forced to refetch **mem$_1$** from memory the very next time it attempts to execute the XCHG instruction inside the spin lock code. Of course, at this point CPU 0 has still not yet updated **mem$_1$** in memory. But CPU 0, also snooping on the shared-memory bus, discovers that CPU 1 is attempting to read the current value of **mem$_1$** from memory, CPU 0 intercepts

and delays the request. Then CPU 0 writes the cache line containing **mem**$_1$ back to memory. Then, and only then, is CPU 1 allowed to continue refreshing the corresponding line in its private cache and updating it.

The cache coherence protocol used in the Intel Architecture is denoted MESI, which corresponds to the four states of each line in processor cache: *modified*, *exclusive*, *shared*, or *invalid*. The MESI protocol very rigidly defines what actions each processor in a multiprocessor configuration must take based on the state of a line of cache and the attempt by another processor to act on the same data. The scenario described above illustrates just one set of circumstances that the MESI protocol is designed to handle. Let's review this example using the Intel MESI terminology.

Invalid	**An invalid line that must be refreshed from memory**
Exclusive	**Valid line, unmodified, guaranteed that this line only exists in this cache**
Shared	**Valid line, unmodified, line also exists in at least one other cache**
Modified	**Valid line, modified, guaranteed that this line only exists in this cache, the corresponding memory line is stale**

TABLE 1. THE MESI CACHE COHERENCE PROTOCOL USED IN THE INTEL ARCHITECTURE. MESI REFERS TO THE FOUR STATES THAT A LINE OF CACHE CAN BE IN: MODIFIED, EXCLUSIVE, SHARED, OR INVALID. AT ANY ONE TIME, A LINE IN CACHE IS IN ONE AND ONLY ONE OF THESE FOUR STATES.

Suppose that Thread 1 running in a spin lock on CPU 1 starts by testing the lock word at location mem$_1$. The 32 bytes containing this memory location are brought into the cache. This line of cache is flagged *exclusive* because it is currently contained only in CPU 1 cache. Meanwhile, when CPU 0 executes the first part of the XCHG instruction on **mem**$_1$ designed to reset the lock, the 32 bytes containing this memory location are brought into the CPU 0 cache. CPU 1, snooping on the bus, detects CPU 0's interest in a line of cache that is currently marked *exclusive* and transitions this line from *exclusive* to *shared*. CPU 1 signals CPU 0 that it too has this line of memory in cache so that CPU 0 marks the line *shared*, too. The second part of the XCHG instruction updates **mem**$_1$ in CPU 0 cache. The cache line resident in CPU 0 transitions from *shared* to *modified* as a result. Meanwhile CPU 1, snooping on the bus, flags its corresponding cache line as *invalid*, as described above. Subsequent execution of the XCHG instruction within the original spin lock code executing on CPU 1 to acquire the lock finds the cache line *invalid*. CPU 1 then attempts to refresh the cache line from memory, locking the bus in the process to ensure coherent execution of all programs. CPU 0, snooping on the bus, blocks the memory fetch by CPU 1 because the state of that memory in CPU 0 cache is *modified*. CPU 0 then writes the contents of this line of cache back to memory, reflecting the current data in CPU 0's cache. At this point, CPU 1's request to refresh cache memory is honored, and the now current 32 bytes containing **mem**$_1$ are brought into CPU 1 cache. At the end of this sequence, both CPU 0 and CPU 1 have valid data in cache, with both lines in the *shared* state.

The MESI protocol ensures that cache memory in the various independently executing processors is consistent no matter what the other processors are doing. Clearly, what is happening in one processor can interfere with the instruction execution stream running on the other. With multiple threads accessing shared-memory locations, there is no avoiding this. These operations on shared-memory stall the pipelines of the processors affected. For example, when CPU 0 snoops on the bus and finds another processor is attempting to fetch a line of cache from memory that is resident in its private cache in a modified state, then whatever instructions CPU 0 is attempting to execute in its pipeline are suspended. Writing back modified data from cache to memory takes precedence because another processor is waiting. Similarly, CPU 1 running its spin lock code must update the state of that shared line of cache when CPU 0 resets the lock word. Once the line of cache containing the lock word is marked invalid on CPU 1, the serializing instruction issued on CPU 1 stalls the pipeline because cache must be refreshed from memory. The pipeline is stalled until CPU 0 can update memory and allow the memory fetch operation to proceed.

Memory bus contention. One not so obvious performance implication of snooping protocols is that they utilize the shared-memory bus heavily. Every time an instruction executing on one processor needs to fetch a new value from memory or update an existing one, it must place the designated memory address on the shared bus. The bus itself is a resource which must be shared. With more and more processors executing, the bus tends to get quite busy. When the bus is in use, other processors must wait. Utilization of the shared-memory bus is likely to be the most serious bottleneck impacting scalability in multiprocessor configurations of three, four, or more processing engines.

The measurement facility in the Intel P6 or Pentium Pro processors (including Pentium II and Pentium III processors) was strengthened to help hardware designers cope with the demands of more complicated multiprocessor designs. By installing the Pentium Counter support provided in the Windows NT 4.0 Resource Kit, system administrators and performance analysts can access these hardware measurements, as discussed last chapter. (This

facility does not work under Windows 2000 and was removed from the Windows 2000 Resource Kit.) While these Counters are given the cautionary rating of Wizard within Perfmon, we hope that the discussion above on multiprocessor design and performance will give you the confidence to start using them to help diagnose specific performance problems associated with large scale Win2K multiprocessors. The P6 Counters provide valuable insight into multiprocessor performance, including direct measurement of the processor instruction rate, level 2 cache, TLB, branch prediction, and the all important shared-memory bus.

The P6 measurements that can often shed the most light on multiprocessor performance are the shared-memory bus measurements. Appendix A lists the various P6 bus measurement counters, using the Microsoft counter names from *Counters.hlp* [5]. Many of the counter names and their unilluminating Explain text are very arcane and esoteric. For example, to understand what Bus DRDY asserted clocks/second means might send us scurrying in vain to the Intel Architecture manuals for help, where, unfortunately, not much help can be had. A second observation, which is triggered by the experience viewing the counters under controlled conditions, is that some of them probably do not mean what they appear to. For example, the Bus LOCK asserted clocks/sec counter consistently appears to be zero on both uniprocessor and multiprocessor configurations. Not much help there. The shared-memory bus is driven at the processor clock rate, and some counter names use the term *cycles* and others use the term *clocks*. The two terms appear to be interchangeable. Although not explicitly indicated, some counters that mention neither clocks nor cycles are also measured in clocks. For example, an especially useful measure is Bus requests outstanding, which measures the total number of clocks the bus is busy.

Bus memory transactions and Bus all transactions measure the number of bus requests. One thing about the bus measurements is that they are not processor-specific since the memory bus is a shared component. The memory bus that the processors share is a single resource, subject to the usual queuing delays. We will derive a measure of bus queuing delay in a moment.

Now, let's look at some more P6 measurement data from a multiprocessor system. A good place to start is with Bus all transactions/sec, which, as noted above, is the total number of bus requests. Figure 10 shows that when the bus is busy, it usually is busy due to memory accesses. Bus memory transactions/sec represent over 99% of all bus transactions. The measurement data is consistent with the discussion above suggesting that bus utilization is often the bottleneck in shared-memory multiprocessors that utilize snooping protocols to maintain cache coherence. Every time any processor attempts to access main memory, it must first gain access to the shared bus.

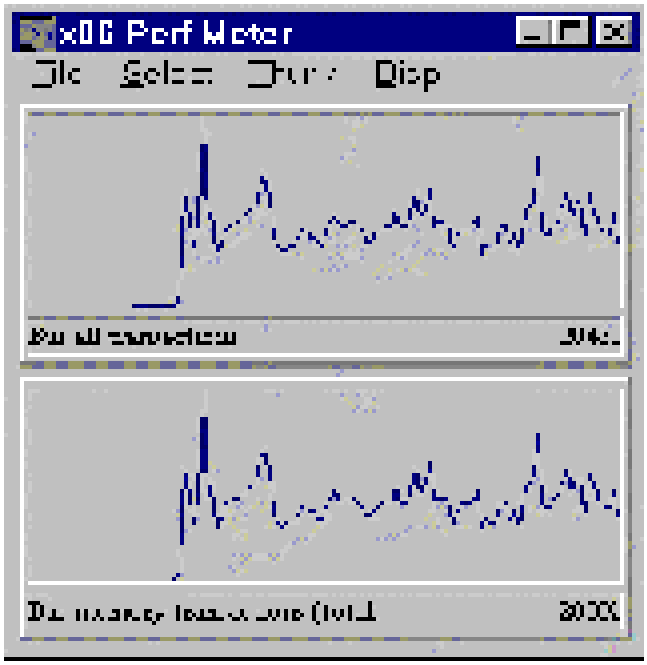

Figure 10. Memory accesses drive bus utilization. Memory transactions represent over 99% of all bus transactions in this example, which is typical of both uniprocessors and multiprocessors. The shared bus can easily become a bottleneck on a multiprocessor.

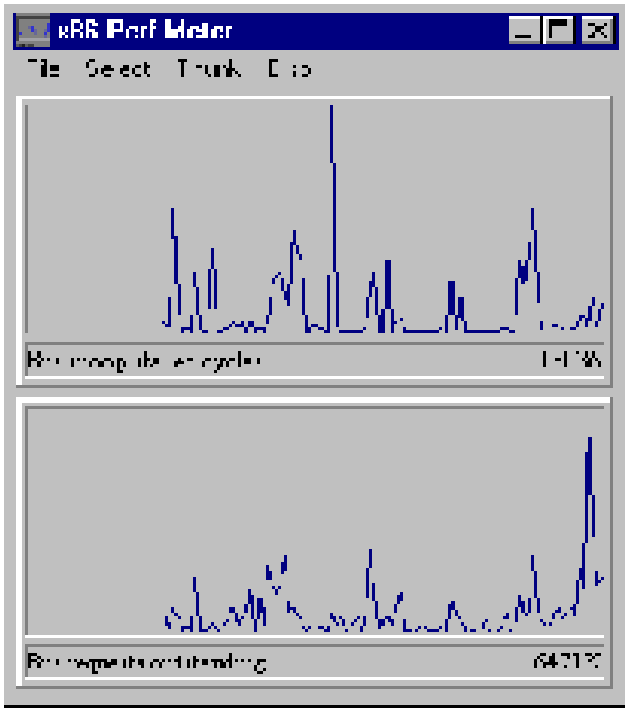

Figure 11. Bus snoop stalled cycles/sec provides a direct measure of multiprocessor shared-memory contention. In this example from a two-way multiprocessor, the number of stalls due to snooping is relatively small compared to all resource stalls.

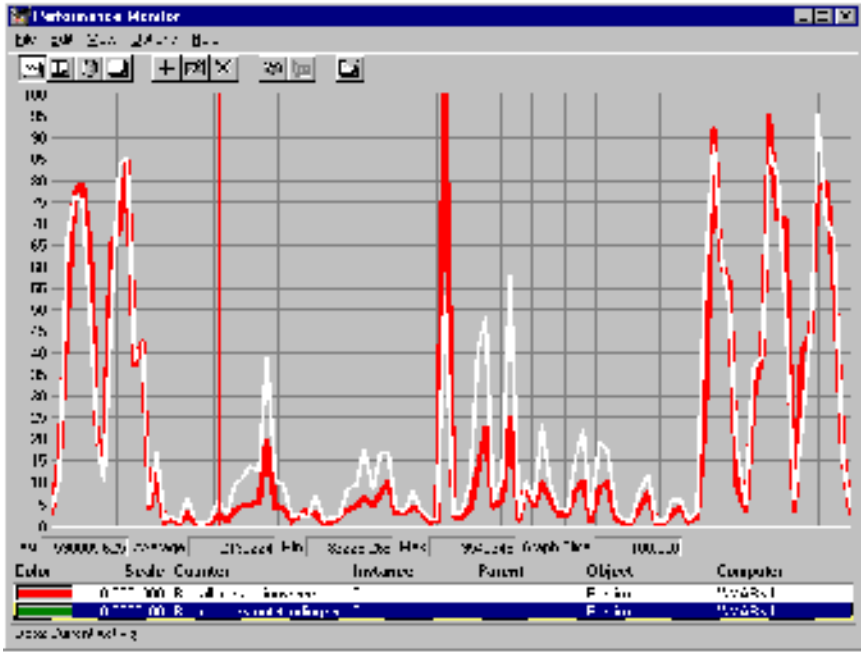

FIGURE 12. TRACKING P6 BUS MEASUREMENTS ON A UNIPROCESSOR USING PERFORMANCE MONITOR.

Larger Level 2 caches help reduce bus traffic, but there are diminishing returns from caches that are, in effect, too big. Each time a memory location is fetched directly from a Level 1 or Level 2 cache, it is not necessary to broadcast the address on the bus. However, at some point, larger caches do not result in significant improvements in the rate of cache hits, yet they increase the management overhead necessary to maintain cache coherence. In this regard, both the rate of Level 2 cache misses and the number of write-back memory transactions is relevant because both actions drive bus utilization. The P6 Level 2 cache performance measurements are especially useful in this context for evaluating different processor configurations from Intel and other vendors that have different amounts of Level 2 cache. By accessing this measurement data, you can assess the benefits of different configuration options directly. This is always a better method than relying on some Rule of Thumb value proposed by this or that performance expert, perhaps based on a measurement taken running a benchmark workload that does not reflect *your* workload.

Another Counter called Bus snoop stalled cycles/sec has intrinsic interest on a multiprocessor. A high rate of stalls due to snooping is a direct indicator of multiprocessor contention. See Figure 11, which again was measured on a two-way multiprocessor. Notice the number of snooping-induced stalls is low in this example. Even though as a percentage of the total resource stalls they are practically insignificant in this example, this is still a measurement that bears watching.

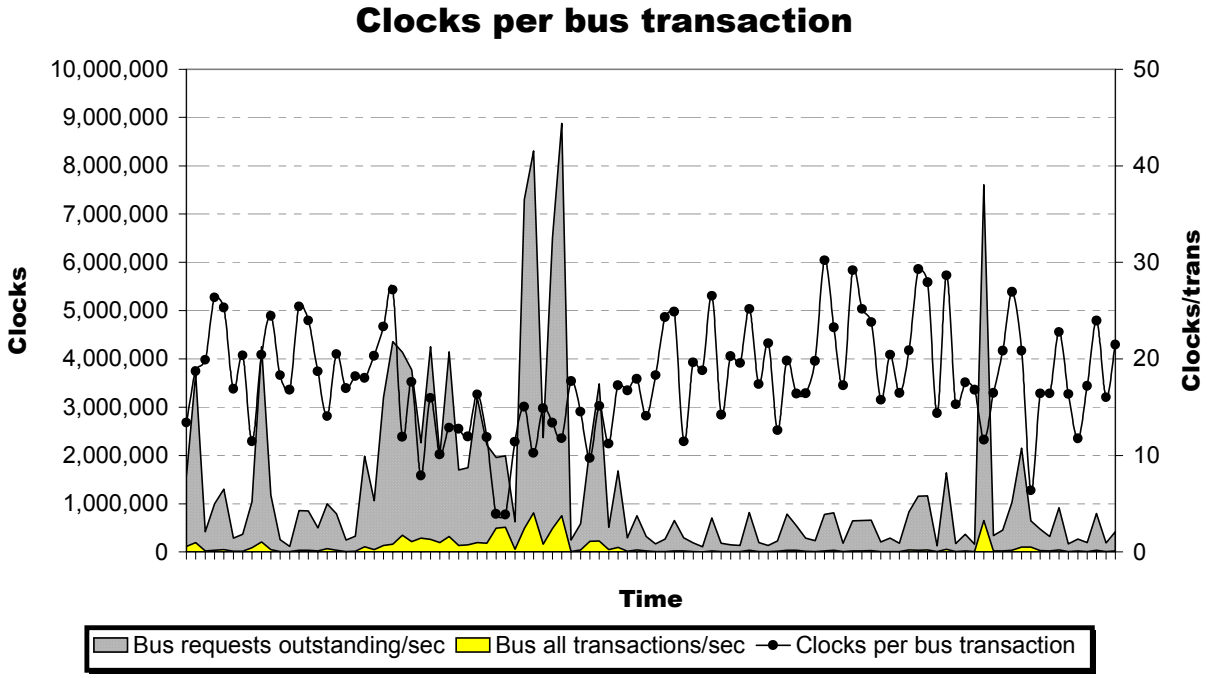

FIGURE 13. CALCULATING THE AVERAGE CLOCKS PER BUS TRANSACTION FROM THE UNIPROCESSOR MEASUREMENTS SHOWN IN FIGURE 11 USING EXCEL. THE NUMBER OF CLOCKS PER BUS TRANSACTION RANGES BETWEEN 10 AND 30, WITH AN AVERAGE OF ABOUT 18.

Next, consider the P6 Bus requests outstanding Counter, which is a direct measurement of bus utilization in clocks. By also monitoring Bus all transactions, you can derive a simple response time measure of bus transactions measured as the average clocks per transactions:

AVERAGE CLOCKS PER TRANSACTIONS =

BUS REQUESTS OUTSTANDING ÷ BUS ALL TRANSACTIONS

Assuming contention for the shared-memory bus is a factor, saturation of the bus on an n-way multiprocessor will likely drive up bus transaction response time, measured in clocks per transaction on average. Figure 12, a Perfmon screen shot, provides a uniprocessor baseline for this calculation. Since Perfmon cannot perform any arithmetic calculations, we exported the chart data to a file so that it could be processed in an Excel spreadsheet. Using Excel, we are able to divide Bus requests outstanding by Bus all transactions to derive the average number of clocks per transaction. (Bear in mind that we can access only two P5 or P6 Counters at a time. Since memory transactions typically represent more than 99% of all bus transactions, it is safe to assume that clock cycles calculated using this formula genuinely do reflect the time it takes the processor to access memory.) The average clocks per bus transaction in this example generally falls in the range of 10-30 clocks, with an average of about 18 clocks per transaction. These calculations are summarized in the chart shown in Figure 13. In the case of a uniprocessor, the memory bus is a dedicated resource and there is no contention. Now compare the uniprocessor baseline in Figure 12 to a two-way multiprocessor in Figure 14. Here the average clocks per transaction is about 30, coming in at the high end of the uniprocessor range. The average number of clocks per bus transaction increases because of queuing delays in accessing the shared-memory bus in the multiprocessor. In a shared memory multiprocessor, there is going to be memory bus contention. By tracking these P6 Counters, you can detect environments where adding more processors to the system will not speed up processing any because the shared-memory bus is already saturated. Bus contention tends to set an upper limit on the performance of a multiprocessor configuration, and the P6 Counters let you measure this.

Queued spin locks. With the Pentium III, Intel introduced a new instruction called PAUSE that reduces the bus contention that results from repeatedly executing spin lock code. The Windows 2000 HAL adds a new queued spin lock function that exploits the new hardware instruction, where available. Device driver code trying to enter a critical section protected by a queued spin lock issues the PAUSE instruction, referencing the lock word protecting that piece of code. PAUSE halts the CPU until the lock word is changed by a different executing thread running on another processor. At that point, the PAUSEd processor wakes up and resumes execution.

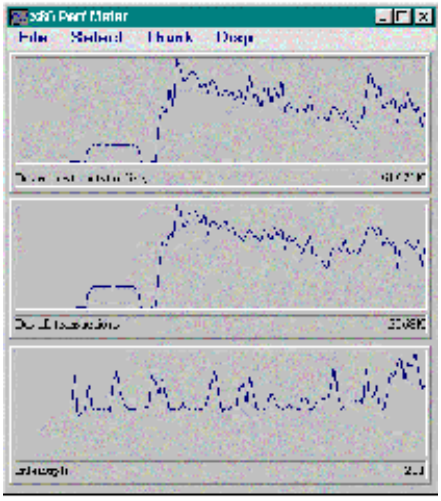

FIGURE 14. AVERAGE CLOCKS PER BUS TRANSACTION ON A TWO-WAY MULTIPROCESSOR. THE AVERAGE BUS TRANSACTION HERE TAKES ABOUT THIRTY CLOCKS.

The PAUSE instruction was designed to eliminate the bus transactions that occur when spin lock code repeatedly tries to test and set a memory location. Instead of repeatedly executing code that tests the value of a lock word to see if it is safe to enter a critical section, queued spin locks wait quietly without generating any bus transactions. Since saturation of the shared memory bus is an inherent problem in shared memory multiprocessors, this innovation should improve the scalability of Windows 2000. At the same time, Intel designers also boosted the performance of the Pentium III system bus significantly, something which should also improve multiprocessor scalability under Windows 2000.

References.

[1] Neil J. Gunther, *The Practical Performance Analyst*. New York: McGraw-Hill, 1998.

[2] Mark B. Friedman, Optimizing the Performance of Wintel Applications. CMG '98 *Proceedings*, (December 1998) 245-259.

[3] Mark B. Friedman, Windows NT Page replacement Policies. CMG '99 *Proceedings*, (December 1999) 234-244.

[4] *Intel Architecture Software Developer's Guide: Volume 3, System Programming Guide*, 1998.

[5] *Microsoft Windows NT 4.0 Workstation Resource Kit*. Redmond, WA: Microsoft Press, 1996.

APPENDIX A. P6 SHARED-MEMORY BUS MEASUREMENTS.

Bus all transactions/sec
Bus BNR pin drive cycles/sec
Bus burst read transactions/sec
Bus burst transactions (total)/sec
Bus clocks receiving data/sec
Bus CPU drives HIT cycles/sec
Bus CPU drives HITM cycles/sec
Bus deferred transactions/sec
Bus DRDY asserted clocks/sec
Bus instruction fetches/sec
Bus invalidate transactions/sec
Bus IO transactions/sec
Bus LOCK asserted clocks/sec
Bus memory transactions (total)/sec
Bus partial transactions/sec
Bus partial write transactions/sec
Bus read for ownership trans/sec
Bus requests outstanding/sec
Bus snoop stalled cycles/sec
Bus writeback transactions/sec