Under the Hood: GigaHertz and CPWs

January 30, 2010

Disclaimer – GigaHertz and CPWs

GigaHertz and CPWs

"It depends", often preceded by a sigh. Ask any processor performance guru a general question about how fast something can be done on a computer and that, as often as not, is the answer you get. So what's going through their mind as they say this? Or imagine you are at your local personal computer retail location considering the laptop to buy for your college-bound student. What's the first statistic that you see? Very likely, the processor frequency. Higher is better right? And they charge for it too. But what is processor frequency really? And what does processor frequency really mean to the real overall performance of your system? That, as it relates to the main processor complex[1], is the purpose of this paper. We're going to talk here about what really constitutes the performance of a computer system.

A few definitions first ...

- **Frequency** ... MegaHertz (MHz), a.k.a., Millions of Cycles Per Second. GigaHertz (GHz), 1000s of Megahertz. For example, ranges of frequencies for a few PowerPC systems include:
  - POWER5 ... 1.9 to 2.3 GHz
  - POWER6 ... 4.2 to 5 GHz
  - POWER7 ... 3 to 4.1 GHz
- **Cycle Time** ... This is mathematical inverse of frequency. For many recent processors, this can often be expressed in some fraction of nanoseconds/cycle. A 4 GHz processor has a ¼ nanosecond cycle time.

Did you know, though, as the frequency was doubled from POWER5+ (2.3 GHz) to POWER6 (4.7 GHz) the CPW-based capacity of the system increased 30-40%? It is also true that even though POWER7's frequencies are less than those of POWER6, corresponding core-to-core CPW capacity of POWER7 is again considerably higher than that of POWER6. Have no doubt that the frequency of a processor is important, but processor frequency is only one of the many tools that processor designers have available to them to provide the required performance.

So let's now take some time to look under the hood and see how the pieces, when put together, really affect the performance of your workloads.

Processor Instruction Sets and Pipe Lines

You probably have some recollection that the PowerPC architecture on which IBM i runs is a Load/Store-based RISC-style architecture. Your line of code in a high-level language is being converted to one or more of these PowerPC instructions by the compiler. The processor frequency is often thought of as the rate at which the processor executes these instructions. That's a handy shorthand, but this mind set is largely incorrect. The processor frequency does dictate the rate at which a lot of work is done on the processor core. Other parts of the system run completely asynchronously to that of a core. So let's start by looking at the relationship between frequency and instruction execution on most processor cores.

[1]This is the processor chips, caches, memory controllers, memory DIMMs, and the busses interconnecting it all into a single cache-coherent SMP.

Although you might expect otherwise from a RISC instruction, PowerPC instructions do not execute from beginning to end within the span of one cycle. Cycle times are so fast that only a portion of the instruction - say the actual addition of data - takes one cycle. Each instruction actually executes in a number of stages, each stage taking one cycle. Every processor implementation is different but if you are thinking roughly 5-10 stages to fully execute an instruction that's about right. I can say roughly because, except for some cases with branching instructions, the number of stages does not significantly contribute to performance. And the reason for this? In many cases, within any given cycle, it is possible that each stage is executing some portion of a different instruction; it is entirely possible for instructions to enter the first of these stages - also often called a "pipe" - and another to leave the last of a pipe's stages every cycle. This is where the idea that an instruction is executing every cycle comes from; even though instructions take multiple cycles to execute, one can exit a "pipe" every cycle.

Clearly, if data and instruction stream are available to the processor's pipes, the faster the cycle time, the faster that instructions appear to execute. And to be available to the pipes, the data and instruction stream must reside in the processor core's L1 cache(s); this is something that we'll be looking at shortly.

As a bit of a side note, partly because there are various classes of instructions, there are multiple processor pipes, allowing instructions to also execute in parallel. On recent PowerPC processor cores, there are two pipes for arithmetic instructions, another two pipes for instructions which access storage, one for branch instructions, and other pipes for longer running instructions such as those associated with floating-point instructions. Due to these multiple pipes, with the right mix of instructions, it is entirely possible to be executing (and completing) instructions at a rate of five instructions per processor cycle; having a different instruction executing in every stage of every pipe is the ideal. A PowerPC POWER6 processor running at 5 GHz and executing at this ideal rate would allow for

$$5 \text{ X } 10^3 \text{ Million Cycles per Second} * 5 \text{ Instructions/Cycle} = 25{,}000 \text{ MIPS}[2]$$

for each processor core. This ideal happens occasionally for short bursts, but hardly ever for any extended periods of time. Typical programs are such that it is more likely that the instructions are only capable of being executed at a much lower rate. Much more typical for commercial applications is a rate in the neighborhood of **5 cycles/instruction** (or worse), rather than this 5 instructions/cycle. Why this is the case and why performance does not always scale up with cycle time - and from there the meaning of CPWs - is the purpose of the remainder of this paper. And, interestingly, you have some control over this difference. So, yes, cycle time is important, but so is the way that the program dictates the instructions and data that the processor perceives. So what's causing this less-than-efficient use of the processor cores(s). First, there are periods of time within which a processor really is executing at this ideal rate. But there are also periods of time within which conditions are such that instruction stream execution in one or more pipes

[2]MIPS ... Millions of Instructions Per Second. Also humorously defined by the performance community as Misleading Indicator of Performance.

(and perhaps all pipes) is delayed, often for short periods, occasionally for very long periods of time. We'll be taking a look at a few of these scenarios.

Simple Pipe Line Delays

The short delays in instruction stream execution come largely from the program logic specifying the instruction stream itself. The compiler understands the processor pipeline characteristics and so attempts to lay out instructions to run most efficiently. But the compiler must also produce correct code, following the intent of the program and - just as important - following the requirements of the programming language used. The logic of the program creates dependencies between instructions; when these occur these dependent instructions can not be executed in parallel. For some types of dependencies, there are also delay bubbles of one or more cycles between instructions. Although the compiler is always looking for the opportunity to execute multiple instructions per cycle, often the program logic and the language requirements don't allow the compiler to produce this parallelism. For example, consider the following very simple snippet of code:

If (A == (B+1)) D = C + 1;

Although it would be nice to execute the comparison of A and B+1 and the increment of C all at the same time, the compiler only allows access of C after any code which finds that A is equal to B+1. The processor needs to first be told to produce B+1, to later execute the comparison, and only then can it conditionally access the variable C in storage. And then only after C is accessed can C+1 be calculated, which is subsequently stored into D. Even though multiple pipes exist in the hardware, the hardware simply cannot allow these dependent steps to execute in parallel. Further, the hardware is often such that dependencies like these create very short delays between them. There are many types of such delays. For all of the compiler's efforts, with the compiler often being successful, these delays nonetheless happen very frequently throughout the code.

Another example of a pipeline delay comes from "mispredicted branches". The hardware guesses the direction of most conditional branches with a high degree of success. This is done well ahead of actually knowing the direction. When it does know, the direction of the instructions being fetched may need to change away from the direction predicted; the instructions along the wrong path are already partially in process working through the earlier stages of the pipe (which is what you would want with correct prediction). As a result, these wrong instructions are all thrown away and the pipe starts all over again. If you are thinking that this delay is roughly ten cycles, you are about right.

There are many other types of such pipeline delays. Certainly, a faster cycle time allows for the next instruction(s) to be executed sooner <u>after</u> handling such delays, but neither is the processor making much progress through the instruction stream <u>during</u> these delays.

Storage Access Delays

The next types of delays occur when accessing storage. For many years now, processor cycle times have improved at rates well ahead of main storage access latencies. What had[3] taken just a few processor cycles to access the contents of main storage has become many hundreds of cycles. This is not because main storage access latencies have become longer - indeed they have actually become shorter - it is because the processors proper (also called the "core") have become faster at a faster rate. It may take <u>less time</u> to access main storage today but, as a result faster cycle times, it also takes <u>more cycles</u>. For example, in the time that the processor must wait for a single memory access, a processor is capable of executing many hundreds of instructions. In order to minimize this effect, processor designers long ago had introduced the notion of cache, a relatively small memory close to the processor cores with access speeds at or approaching the processor's cycle time. Without cache, the processors would only execute a small handful of instructions at processor frequency and then wait for the hundreds of cycles to fetch the next instructions and data out of main storage. Clearly, without cache the very rapid cycle time of the processor becomes nearly irrelevant. Cache, though, only acts as a filter; some memory accesses do occur, at the very least to prime the cache with data or instructions. Programs, BTW, are normally completely unaware of the hardware's cache.

Because of the relatively fast frequency of recent processors, these recent processors are designed with a rather - and ever more - complex cache topologies. They have a small on-core cache for instructions and another for data, both of which can be accessed at very nearly the cycle time of the core; these are called the L1 Data and L1 Instruction caches. From here each new design introduces various other forms and sizes of Level 2 and Level 3 caches, each level being larger and slower than the previous.

Although it makes one's head hurt to think like this, it is nonetheless true that the processor cores do not ever really access main storage; the core accesses the data and instructions in its cache. If what the core wants is not there in the cache, the cache control initiates something called a "cache fill" that can take a while. So let's look at this next.

All levels of the cache work with a common sized block of storage 128 bytes in size. In the event that data or instruction stream is not found in the many 128-byte blocks of the L1 cache(s), the core then checks the L2 cache; succeeding there, the core pulls 128 bytes (a cache line) from the L2 into the L1 cache. Such block-size cache fills succeed in speeding performance because there is a spatial locality to most storage accesses; an access of one byte in a storage block is very likely to be followed soon by a subsequent access of other bytes in the same storage block.

Again, it takes only a handful of cycles to fill a 128-byte block into an L1 cache, often with the instruction stream execution delayed until complete. If an L1 cache access also misses on the L2 cache, there is a still larger L3 cache from which the needed data might be able to be filled. The L3 cache increases the probability that data is found in some cache, but it takes considerably longer to find it there. Although this L3 cache access introduces a larger delay, it is still considerably shorter than the amount of time needed to access main storage.

[3]Read this as "quite a few processor designs ago".

You can largely think of any L1 cache miss as ceasing the execution of this particular instruction stream until the associated cache fill is complete; instruction execution is delayed due to cache misses. Focusing only on a single chip, this delay, which is short for misses satisfied by a local L2 cache, gets increasingly longer as the data is satisfied from the L2 cache of the neighboring core, the L3 cache associated with the chip, and then from the main storage hung on this chip.

Although the L1 cache access latencies - the actual time, not cycles - are certainly a function of the cycle time of the processor (in fact the cycle time itself might be defined by this latency), accesses of main storage proper naturally take a certain amount of time, a time period independent of the current cycle time of the processor core. The time period might be defined in terms of the nominal cycle time, but for much of the access the core's cycle time does not matter.

Let's look at this a tad differently. Suppose that after executing 100 instructions at a rate of - say - one cycle per instruction, the 100th instruction incurs an L1 cache miss. To keep it simple, let's say that the associated cache fill takes exactly 100 cycles to complete. The delay of that **<u>one</u>** instruction halved the average rate that instructions are executing to. What would have taken 100 cycles without the cache miss takes 200 cycles with it. If instead that cache miss had taken - say - 400 cycles, the same 100 instructions now take 500 cycles; the rate of instruction execution is down to one fifth or 5 cycles per instruction. If you wanted to speed this up, where would you spend your time? The problem starts with the number of misses. If your application were to access one byte in each of 128 128-byte blocks of storage (each incurring a cache miss), wouldn't the performance of your application have been far better off if all 128 bytes being accessed had been in contiguous storage and incurred a single cache miss? Again, whether you access a single byte or 128, the hardware is still going to fill the cache with a 128-byte block of storage; the application may as well go along for the ride.

Again the fewer the 128-byte blocks of storage that need to be pulled into the L1 cache(s), the faster the instruction stream is executed. If there are relatively long periods of time where instruction stream and data are found in the L1 cache (as compared to the time spent waiting to fill the L1 cache), the average number of cycles per instruction executed during that period can become quite small. But even with L2 cache hits, if the L1 miss rate is rather high, the frequent need to access the L2 also slows down instruction processing noticeably, but certainly less so than if these same accesses were from memory.

And now to the point of all of the above discussion....

Don't think of cache misses as being somehow a tragic event. Cache misses happen and often frequently. Cache hits happen as well, hopefully far more frequently; the locality of data references assumed by the hardware's cache design often works very well. When data is found in the L1 cache(s), the instruction stream executes at the rapid rates mentioned earlier. The more instructions that can be executed without incurring a cache miss, that faster that work will complete. And some workloads really do have far lower rates of cache misses than others. The performance of these workloads will improve at rates more closely proportional to that of the processor frequency improvements. Here the faster the core's cycle time, the faster the workload completes. Those workloads that are more prone to cache misses - especially those requiring access outside of the local L1(s), L2, and L3 - improve with cycle time, but not nearly as much.

So, it depends. Depending on the workload, cycle time matters and sometimes it does not matter as much. In the section on SMT, we'll see a way that makes cycle time matter more.

Perhaps obvious by now, an application design which is observant of the cache design is also one which will execute faster, consuming less of the processing capacity of the system. And when it does, its performance becomes increasingly a function of the cycle time. It's worth noting that, whether the application is accessing one byte or more bytes, whether it is accessing data or instruction stream, if that byte is not in the L1 cache, the processor cores will be filling the cache with **128 byte blocks** found on a **128-byte boundary**.

A Side Glance at SMT

We'll be looking at SMP (Symmetric Multiprocessing) and multi-core/multi-chip computing shortly, but we are going to pause and consider SMT (Simultaneous Multithreading) here. OK, so now you have the picture of a valuable processor core, happily burning through a set of instructions at some outrageously rapid rate. And then it stops, and it waits; the next instruction to execute is sitting somewhere in far away main storage and the core can't continue execution until the needed instruction stream is available in the L1 cache. And your task waits, all the while you are being charged (at least conceptually) with the use of that processor because your task is dispatched to this processor core. Scads of compute capacity is effectively going to waste.
In the meantime why can't this processor core be executing the instruction stream of one or more other dispatchable tasks? It can and it's done via SMT[4]. During the times that one task is unable to be executing instructions, another task can be, and it can do so on the same core using the same pipes. Interestingly, SMT's capabilities do not stop there. Each core has multiple pipes and it takes multiple stages for any given instruction to execute. And even without cache misses, hardly any instruction stream of a single task consumes all of these stages of all of these pipes. If one task doesn't use it all, this means that there is more compute capacity in a core available for consumption by the execution of instructions of other tasks. On the POWER5/POWER6 processors, the cores allowed two such hardware threads of execution; POWER7 - enabled partly by its enhanced cache design - drives this number up to four.

So essentially, SMT allows more of the capacity of each core to be used. There is fewer cycles going unused. It seems to follow that the more SMT drives up capacity utilization, the more that the cycle time of the core matters to the execution speed. As we've seen, the cycle time of a core executing a single task's instruction stream may or may not strongly dictate the throughput being produced by that task. As more task's instruction streams are added - and there is less wasted capacity - the total throughput of all the tasks increasingly becomes a function of the cycle time. But for the performance of individual tasks having to share a core with others, there is a bit of a dark side. Let's suppose there is an environment in which a core is executing the instruction stream of multiple tasks, and each task can successfully execute its instructions while the other is waiting on a cache miss. Then each task is essentially executing alone, seeing the full benefit of

[4]Some of you may recall that this same notion was supported on a PowerPC processor design a while ago as something called Hardware Multi-Threading (HMT). SMT proper was first implemented on POWER5 processors.

the core.   But more often the instruction streams also get in each other’s way to varying extents, decreasing the benefit. For examples, within the core’s pipes, each pipe’s stage(s) can only be used by one instruction at a time; another task wanting to use the same pipe stage at the same time much wait for a moment. All the instruction streams are sharing the same single L1 caches; they are bound to be overlaying 128-byte blocks of another task’s data with their own. The point is that - unlike executing tasks on different cores - since the instruction streams are sharing the many resources of a core, the are bound to get in each other’s way. As a result, although core capacity usage can increase significantly with SMT, the execution speed of the individual instruction streams slow some as well. There is more capacity available at higher system utilization because of SMT, but during this same period each task’s individual throughput slows as well.

The Translation Lookaside Buffer

While we are still discussing individual cores, let’s take a look at something that typically gets ignored, the performance of virtual address translation. It is a major factor in limiting the benefit of improved cycle time on processor scalability. What follows is a bit esoteric, so reader beware. Are you sure now that you don’t want to skip to the next section? OK, I’m duly impressed. Here goes!

You probably recall from your days in college that your application’s nice contiguous address space is not so nice and contiguous when it comes to how it resides in main storage. You knew that some main storage manager in the OS’ kernel handled it for you and that was the end of it. Well, it also matters to the processor and therefore to performance. The processor manages a structure within each core - typically called a TLB (Translation Lookaside Buffer) - which maps your view of an address space onto the physical address necessary to actually access main storage. When your application executes an instruction to access a variable, that instruction is using your view of the address space. The hardware, when executing that instruction, checks the TLB to see whether it knows of your address and, when it does, returns the corresponding main storage address - also called a “real address” - of your data.

The key concept to know here is that only a very few “pages” of all of main storage are mapped within this TLB at any given moment. The TLB is effectively a short term cache of recently accessed address mappings. So, as with cache misses, it is possible to have a TLB miss when an instruction uses an effective address which the TLB is incapable of translating to a real address. On processors used by IBM i, a TLB miss then ceases instruction execution while the hardware accesses a very large table in main storage where many times more pages are mapped. Upon successful translation of an address through this table, the TLB is updated, allowing the instruction incurring the TLB miss to continue; hopefully, many subsequent instructions can use this newly updated information in the TLB.

Interesting right? But what’s the point? An instruction which should not have been delayed at all if the address translation had succeeded instead took a lot more time; perhaps as much as one or more main storage accesses, many hundreds of cycles away. And, as with cache misses, instruction stream execution of that thread effectively ceases during this entire time.

Again, as with cache misses, this sort of thing is happening all the time. It is therefore worth noting that if some application - or large number of instructions within this application - is accessing only a relatively few number of pages, the probability of a TLB miss is very small. Here, TLB miss delays can be largely irrelevant. On the other hand, if a large number of pages are being accessed in a short period of time, it is possible that the single biggest contribution to the time required for an application might be these TLB miss delays. And, per this paper, these might not scale with cycle time.

The SMP Fabric

On POWER5 and POWER6 processors, the basic building block for constructing larger systems consists of a processor chip, each with two processor cores, the core's L1 and L2 cache, a separate L3 cache; each chip supports a memory controller for some number of memory DIMMs. POWER7 has some significant differences in core and cache topology, but the one glaring difference is that each chip is designed to support eight such cores. These are all effectively what is called "multi-core" computer chips; POWER7 is just more so.

All of these grow systems to still more cores by using these chips - and their attached memory - as the basic building blocks and replicating them to whatever size system is needed. Where POWER5 and POWER6 can grow to 64-way using 32 such chips, POWER7 systems can grow to 256-way using the same number. We call the hardware interface between these chips creating the single SMP the "fabric".

These are all cache-coherent SMP systems; the contents of all main storage and all cache in the entire complex is accessible from any processor core. All memory is accessible, all cache is coherent.

Allow me a moment to quickly define the term "coherent cache" since it is key to understanding the performance effects of an SMP fabric. From an application's point of view, the application is simply accessing the contents of main storage. But the processor cores know that they are really accessing the contents of the cache. And changes that the application makes to what it considers to be main storage can actually continue to reside in its processor's cache indefinitely; only much later does the change make its way out to main storage. Further, the contents of the same block of main storage can reside within the cache(s) of multiple processor cores, both on the same and different chips. A coherent cache is one where hardware largely hides the fact that cache exists at all from the software. This coherency management requires control traffic both within and between the chips. It also often means that data is copied (or moved) by hardware from a cache line on one core to a cache line of another core. For example, if a core of Chip S incurs a cache miss on some data access and the data happens to still reside in the cache of a core of Chip T, the hardware will find the needed data, read it out of Chip T's cache, and transfer it across the inter-chip SMP fabric to the core on Chip S; this is all without first going to the data's real location in memory. Doing this efficiently is, of course, very important to performance.

The distributed approach used for both main storage and cache - that of having main storage and cache associated with each chip - allows for massive bandwidth as well as allowing many accesses to have no performance impact on similar accesses done by other chips. But it also comes at a performance cost. A core's access latency of a local access - a cache fill satisfied from the cache or main storage nearby (local) to the core - is done faster than a similar access of cache or memory associated with another chip. Indeed, typically, the further away - the more chips the access must traverse - the longer the access. The contents of all cache and memory might be accessible from all cores, it's just that the time it takes to do so is dependent on location. This is a limited instance of a CC-NUMA architecture, Cache-Coherent NonUniform Memory Access.

Percentage difference of a local vs. remote memory access is a few tens of percent longer for the remote memory access. Since what could have been a considerably faster local cache access can also become a remote cache access, the amount of additional time required for the inter-chip communication over the SMP fabric considerably increases the relative percentage difference between local and remote cache accesses. The percentage differences differ between each of the POWER5 - POWER7 systems as well.

This all may appear to be generally bad news for performance. It is not. It is only bad if the yardstick for comparing performance is of an application running on a single chip and getting much if not all of its data and instruction stream from the cache on or associated with that chip. Certainly that happens and it happens even for short bursts when data and instruction stream aren't obviously easily cached. But even remote accesses as mentioned above are “fast” - occurring in small fractions of a microsecond - just not as fast as strictly local sub-nanosecond accesses. Most applications are a mix of the various forms of - and various latencies of - memory accesses. It's all fast. But appropriate management - management largely provided automatically by IBM i - of an application can increase its probability of local accesses, decreasing its overall execution time, and increasing the remaining capacity of the system. If you or your application can further change the balance toward more local accesses, performance - and the capacity of the entire system - will improve. Indeed, IBM i provides a number of controls that allow some control of the NUMA characteristics of such systems and includes tools to change this balance. It is worth noting also that this balance can change through cache optimization techniques as well, a subject a bit too long for sufficient discussion here.

Of course, it is not just IBM i that is aware of the way that the hardware really works. The hypervisor is aware as well. Although the user interface provided to define partitions treats any processor or allocation of main storage as being as good as any other, as you can see it is not and the hypervisor knows it. The user interface allows partitions to simply define the amount of capacity that a partition ought to have in terms of the capacity available from a single core. The user interface allows partitions to define their memory requirements in terms of contiguous memory blocks ranging in size from 16 to 256 Mbytes. But the hypervisor then takes these firm requirements and attempts to allocate the partition its processor cores from as few chips as possible and to allocate its required main storage nearby these cores. Often it succeeds completely. Occasionally, it is considerably less than successful. Indeed, as changes to the partition's capacity and storage requirements change, this locality degrades. It follows that - being aware that the memory model is not flat - all cores and memory are not equal; taking this into account when defining partitions is likely to also increase available system capacity.

Getting back to the point of this article - that of to what extent cycle time matters - as with the memory accesses, much of the path of both of the local and remote accesses is outside of the realm of an individual core and its cycle time. Even the remote cache accesses might be only minimally dependent on the cycle time of a core. Much of the time spent on such inter-chip accesses is a function of the speed and remaining bandwidth of the links between the chips as well as the number of links required to support such an access. The speed of the links, of course, matter and to a greater extent the cycle time of each core, but these are quite different concepts.

Back to GigaHertz and CPWs

It may appear that we have gone far afield from the issue of scaling per cycle time. But everything that is being discussed here is reality. This is how the system works. Applications don't know anything about, but are definitely affected by these effects. The only real issue is how often some off-core operation may occur. An application that is able to stay on a processor core and avoid the usual delays between instructions will indeed see the full benefit of a rapid cycle time. At the other extreme are those sets of applications which are spending most of their time merely waiting, executing a relatively smaller set of instructions between such delays. All possible uses of a computing system occur out there somewhere. You've been believing that the use of your system is similar to CPW, and it may well be; that's partly the CPW workload's intent. So what of CPW? And how does it compare to your uses in general?

CPW is a synthetic workload based at least conceptually upon database transaction processing. It consists of a very large number of virtual users - each represented by a job/task - each randomly requesting that transactions be executed against the contents of a large handful of large database tables. From a processor utilization point of view, you can think of each of many individual job/tasks as waking after a short wait, and executing on a processor once or some number of times for short bursts of time. There is no strong relationship between a job/task and any one processor or chip in the system. The database tables and the associated indexes are considerably larger than the size of memory; the intent is that some transactions will require one or more reads of the database from the disk, this implying a fair amount of I/O DMA read and write operations. It's also worth noting that each short burst of time that such a task is using a processor, it is working with data and instruction streams which must ultimately reside in a cache in order actually be processed.

For any given system, the number of users and random think time for each user, the number of disk drives, the amount of memory, and a number of other parameters are chosen to allow the entire workload to consume an <u>average</u> of roughly 70% of the total compute capacity of the system. But this 70% is not always exactly 70%; this is an average of 70% of processing capacity over the entire workload. Within periods of seconds, it does tend to also stay around 70%, but it represents about all possible uses (or non-usage) of a processor. There are periods of time where dispatchable tasks are queued up waiting for any available processor; this is an instantaneous utilization of 100%. There are periods of time where there are no dispatchable tasks (0% utilization) or no dispatchable tasks assigned to any particular SMT-capable core. There are also periods of time when cores are executing on only one of its possible two SMT hardware threads. Every possible state that a core or its SMT hardware threads could be at will likely occur in a short period of time within CPW.

On average, the CPU utilization within CPW is such that over 80% of the time is spent within code associated with i5/OS' kernel; much of the remainder is within higher levels of i5/OS (XPF). The application proper is written in C and COBOL, but that is not really pertinent since only a small percent of the whole system capacity is actually spent within the application code. I won't take you through the details, but this kernel componentry uses the processors in a lot of ways as well. Some components tend to be more compute intensive, touching very little distinct data, but execute a lot of instructions in doing so, and so tend to heavily use the core's pipes. Other components tend to touch data which is shared heavily between tasks, resulting in data flowing between processor caches frequently. Still other i5/OS components simply touch a lot of data while executing fewer instructions; these components tend to fill the cache from main storage. Although there is some tendency to access local memory, these components are also likely to be accessing the contents of memory remote from the core doing the access. I should add that, with paging of data out of and into main storage (that is to/from disk), a fair amount of I/O code is executed with an associated amount of DMAing of data into and out of main storage. The bottom line is, within CPW, about every possible use of the processor will be occurring within a very short period of time. It is intentionally using all processor cores, all SMT states, all forms of address translation, cache accesses of all kinds, and processor and DMA accesses of all of memory. CPW intentionally stresses much of the entire system. It is by no means a set of tasks which are bound to a core (and its SMT threads) which access only the data and instruction stream held within that core's cache (and TLB).

A Near Conclusion

So what of GigaHertz and CPWs? You've read through an awful lot to now roughly answer that question yourself. As you've seen, performance of some part of the computing system is very sensitive to cycle time. This is primarily in the scope of the cores and its pipes. Usage of other components, like main storage access and SMP fabric, take whatever time it takes them to complete; we just happen to measure this time in "cycles". You've also seen that the topology of the system matters to performance. A faster cycle time matters where cycle time matters, but not throughout the entire system.

IBM, of course, publishes CPW ratings for each system supporting IBM i. You've seen that these systems scale up roughly per the number of processors in that system and you've seen that there is a nonlinear relationship between CPW ratings and cycle times. A doubling of the frequency does not necessarily translate to a doubling in the system's CPW rating. Nor would a decrease in the frequency always result in a proportional decrease in the CPW rating. There are a lot of other characteristics which also impact CPW ratings such as cache size, cache topology, TLB size, main storage speed, and instruction execution characteristics. Their interrelationships and relative usage also matter. Each system configuration is different. As with cycle time, any given application will use these characteristics to greater or lesser extents than does CPW. Every application is different than CPW. Fortunately, enough applications use the system and its processors, cache, and main storage in a manner which is sufficiently similar to CPW that the scalability characteristics of CPW have some general applicability elsewhere as well. Indeed, we've considered it serendipitous that CPW's design allows it to be so.

But what about your use? Will it scale per CPW? Some do not; some are close enough. It all depends. Hopefully this overview of a cache-coherent SMP system supported by IBM i has provided you with enough information to know.

… As An Assignment Left for the Interested Student

POWER6's cycle time is considerably faster than that of POWER5. Although processor cycle time is certainly important, you've now seen that a system's performance is much more than just its cycle time. Most applications use so much more - and so are influenced by - the full characteristics of a cache-coherent SMP system. For this same reason, you know why CPWs - for example - do not scale up linearly with cycle time. So, dear student, here's your question….. Suppose that the cycle time did not improve, indeed it became something less. And in doing so energy consumption was decreased. And suppose that system capacity - represented by CPW - was improved through other means, more cores, more SMT, or lower cache coherency overhead being examples. What would that mean to you?

Energy Management

A lot can be written on this subject alone, so here let's stick to the basics. It is a true statement that, for a given processor design and technology, if you were to linearly increase the processor frequency, the amount of energy expended to execute instructions using the frequency increases at a much higher rate. For this reason, a lower frequency is used as a tool to save energy in a computer system. Frequency can be dropped by some fixed amount below the processor's nominal, saving some energy certainly, but lowering the system's compute capacity by some non-proportional amount as well. For this reason, systems also offer dynamic frequency adjustment; when the compute capacity is needed, the system adjusts the processor cycle time upward to it maximum; the nominal capacity is there if you need it. When the capacity is not needed, the frequency can be adjusted downward to some minimum. And, if our claim about frequency is correct, the amount of capacity available at the minimum frequency can be higher than the relative frequency drop implies. (Of course, that is hard to measure because the frequency dynamically increases as capacity demands increase.)

The POWER7 Processor Design

It is also true that POWER7's nominal frequencies are lower than corresponding POWER6 systems. But for a large class of workloads, the average capacity of POWER7 systems is considerably higher than POWER6. It almost goes without saying that, when comparing the capacity of the same number of POWER7 chips to POWER6 chips, the capacity available on POWER7 far exceeds that of POWER6 (i.e., an 8-core POWER7 provides more capacity than a 2-core POWER6, over and above the POWER7's core-to-core performance advantage).

So if the frequency of POWER7 core is less than that of a POWER6 core, where is the additional capacity coming from. As a very quick overview of the differences we find:

1. POWER7 supports 4-way SMT and POWER6 supports 2-way SMT, allowing the execution of twice as many thread's instruction streams per core.
2. POWER7's cache topology has a number of advantages including a much faster L2 cache and more on-chip cache available for cache line cast-outs.
3. POWER7's core support out of order instruction execution, POWER6 executes instructions largely in order.
4. POWER7's eight cores per chip ensures that as many as eight SMP cores can support a coherent cache across cores without needing to cross chip boundaries (i.e., data sharing across the cache of multiple cores is done more efficiently).

Essentially, POWER7 provides proof of what has been discussed through this long paper; **frequency, although important, is only a component of what enables performance in a computer system.**

As frequency improves in the future, and it will but at a lower than historical rate, that higher frequency will provide its part in improving the performance of future systems.

© IBM Corporation 2016
IBM Corporation
Systems and Technology Group
Route 100
Somers, New York 10589

Produced in the United States of America
February 2013
All Rights Reserved