Of GigaHertz and CPWs

Mark Funk
Robert Gagliardi
Allan Johnson
Rick Peterson

January 8, 2008

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, your PowerPC P5 processor ranges from 1.9 to 2.3 GHz. The PowerPC P6 variously comes as systems with processor frequencies of 3.5, 4.0, 4.2 and 4.7 GHz (and higher).
- **Cycle Time** ... This is mathematical inverse of frequency. For many recent processors, this can often be expressed in some small number of nanoseconds/cycle.

Next, you probably have some recollection that the PowerPC architecture on which i5/OS resides is a Load/Store-based RISC-style architecture. Your line of code in a HLL is being converted to one or more of these 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. We'll be examining this shortly. The processor frequency - or a unit fraction thereof - does dictate the rate at which a lot of work is done on the processor chip. Other parts of the system run completely asynchronously. But let's goes back and look at the relationship between frequency and instruction execution on most processors.

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 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.

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.

---

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

As a bit of a side note, partly because there are various classes of instructions, there are multiple processor pipes. In the later pipe stages, on PowerPC P5 and P6 processors, there are two pipes for arithmetic instructions, another two pipes for instructions which access storage, one for branch instructions, and other pipes of 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 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 P6 processor running at 4.7 GHz and executing at this ideal rate would allow for

4.7×10^3 Million cycles per second * 5 instructions/cycle = 23,500 MIPS[2]

for each processor core. But this ideal hardly ever happens for any extended period of time. 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 of - say - **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(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 (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 logic implied by the instruction stream itself. The compiler understands the processor pipeline characteristics and so attempts to lay out instruction stream 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 it is clear that this dependent instructions can not be executed in parallel. For some types of dependencies, there are also 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:

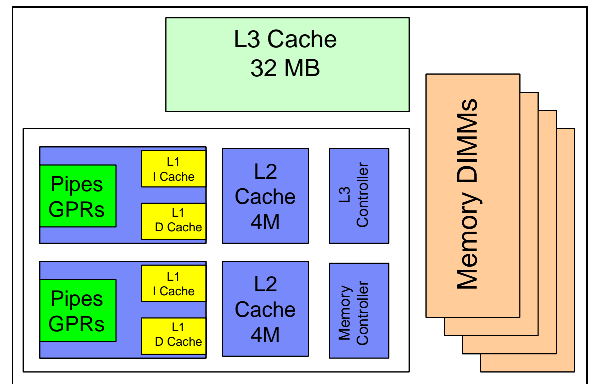
```
If (A == (B+1))  D = C + 1;
```

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

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 may only access C after any code which finds that A is equal to B+1. The processor needs to first know B+1, then the comparison, and only then can it access the variable C in storage. And then only after C is accessed can the C+1 be calculated, which is subsequently stored into D. Each of these dependencies create very short delays; there are many types of such delays. But for all of the compiler's efforts, these delays happen very frequently throughout the code.

Storage Access Delays

The next types of delays are essentially storage access delays. 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 less time to access main storage today, but as a result it also takes more cycles. For example, in the time that the processor must wait for a single memory access, a processor is capable of executing hundreds of instructions. In order to minimize this effect, processor designers long ago had introduced the notion of cache, a relatively small memory with access speeds at or approaching the processor's cycle time. Without cache, the processors may 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.



Because of the relatively fast frequency of P5 and its very considerable improvement on P6, the processors are designed with a rather complex cache topology. On P6 there is a small cache for instruction stream and another for data both of which can be accessed at very nearly the cycle time of the processor; these are called the L1 Data and L1 Instruction caches and you can think of one each of these as residing within the core of the processor. Each P6 core is also associated with a larger cache, these being 4 Megabytes in size and taking tens of cycles to access. In the event that data or instruction stream is not found in 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. This process of pulling the contents of a block of storage into a cache is called a "cache fill". Such block 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 pull 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 an L3 cache (32 Megabytes in size) associated with each pair of P6 cores from

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

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 in excess of one hundred cycles 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.

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. Focussing only on a single chip, this delay, 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. As it relates to comparing P5 to P6, each of these misses take more cycles - but not necessarily time - on P6 than were perceived on P5.

Let's look at this a tad differently. Suppose that after executing 100 instructions at a rate of - say - one instruction per cycle, 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 **one** instruction halved the average rate that instructions are executing. 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 rate of execution is down to one fifth, 500 cycles with the miss vs. 100 cycles without it. If you wanted to speed this up, where would you spend your time? It's 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 those bytes 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 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 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.

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 cache design often works well. When data is found in the L1 cache(s), the instruction stream executes at 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 lower rates of cache misses than others. And these workloads will improve at rates more closely proportional to that of the processor frequency improvements. Those that are more prone to cache misses - especially those requiring access outside of the local L1(s), L2, and L3 - will improve with cycle time, just not nearly as much. 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. 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 P5 and P6 processors will be filling 128 bytes on a 128-byte boundary.

A Side Glance at SMT

We'll be looking at SMP (Symmetric Multiprocessing), 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 mean time, why can't the processor be executing the instruction stream of one or more other dispatchable tasks. It can and its done via SMT[4]. During the times that one task is unable to be executing instructions, another can and do so on the same processor core using exactly the same processor pipes. Interestingly, SMT's capabilities do not stop there. As was mentioned earlier, 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, this means that there is more compute capacity in a core available for consumption by the execution of instructions of other tasks.

Wonderful. If the instructions of one task can't consume every stage of every pipe, can an additional task's instruction stream do it, allowing the ideal MIP rating of this one core to be achieved with more tasks? In an ideal world, yes. It's important to realize that there are other processor resources aside from those of the pipes that are also being consumed. For example, the L1 cache(s) of a single core are being shared by these tasks. If the data and instruction stream of a single task is too large for this smallish cache, it is also very likely to be too small for multiple tasks. The effect is a larger L1 cache miss rate than would have occurred if there were only one task executing. However, for most uses, the capacity gain occurring from the sharing of the processor pipes exceeds the capacity loss of the increased L1 cache miss rate. And keep in mind that the L1 caches are backed up by a considerably larger L2 cache. But, in fairness, SMT puts increased pressure on the L2 cache as well. There are other shared resources within each core which, when busy, also slow instruction processing.

Clearly enough, if there is only ever one task dispatched to a core, SMT buys nothing in terms of performance capacity. But 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 it's instructions while the other is waiting on a cache miss, then the capacity of the system would tend to increase at a rate which really is closer to the rate at which the cycle time increased. In effect, SMT does allow cycle time improvements to be perceived as proportionally improving capacity. But again, the instruction streams also get in each other's way to varying extents, decreasing the benefit. Further, cache miss rates of each task, even without the extra competition, may be such that both/all of the tasks are concurrently waiting on cache misses at the same time. During such periods of time, capacity improvements due to cycle time improvement can disappear.

---

[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 P5 and P6 processors.

As a caveat, the multiple tasks really do compete for the resources of any given core. At the simplest, if an instruction of one task happens to be using a stage of a pipe during the same cycle that another task could have profited from executing an instruction there (and then) as well, then one of the tasks must wait. This, along with increased pressure on the caches and similar resource conflicts, are happening all the time. For this reason, SMT buys more capacity. But when multiple tasks do execute on the same core, SMT is very likely to also slow the average execution of each of these tasks. The capacity benefits, though, typically exceed the degrading effects on each such task; it is not uncommon to see capacity improvements in the range of 30-50% due to P5-P6's SMT alone.

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. It 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 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. 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 i5/OS[5], a TLB miss then ceases instruction execution while the hardware accesses a very large table in main storage (portions of which are often also in the cache) 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. And the amount of delay is normally about the same amount of time it takes to handle an L1 cache miss. As with an L1 cache miss, the access of the translation table might require an access all the way out to main storage, many hundreds of cycles away. And, as with L1 cache misses, instruction stream execution of that thread effectively ceases during this entire time.

---

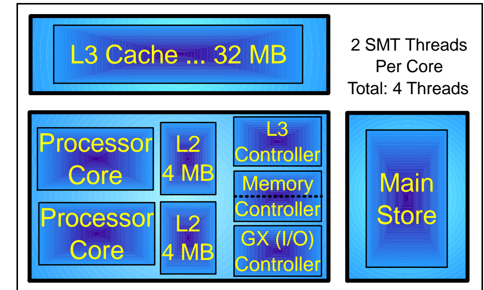
[5] Some PowerPC processors not used by i5/OS result in an interrupt to software upon TLB miss, requiring software - and many more cycles - to update the TLB.

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. Indeed, it is worth noting that the size of the core's structure doing this address translation within the core is smaller on P6 than it is on P5.

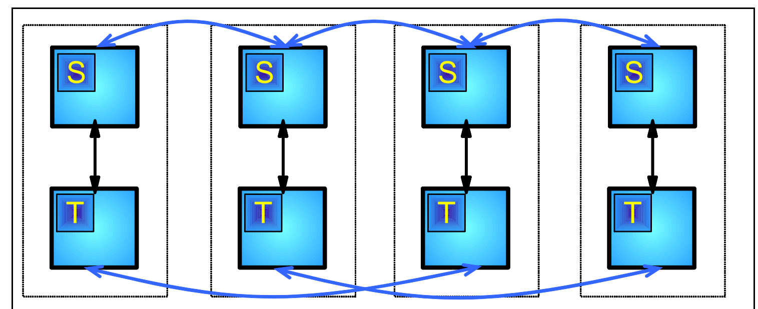
Since you are still with me, I'll drop one more bit of esoterica on you. It is because of this effect alone that processor architecture supports a notion of a "large page". The intent of a large page is to represent a larger portion of the real address space (of physical memory) using fewer TLB entries. For instance, a single 64K large page uses a single TLB entry to represent the same real address space as sixteen contiguous 4K pages. When large pages can be used, the TLB allows fast access to more physical memory. Large pages do get used internally - but only where appropriate - by i5/OS.

The SMP Fabric

On P5 and P6 PowerPC processors, the basic building block for constructing larger systems consists of a processor chip, each with two processor cores, their L1 and L2 cache, a separate L3 cache, and some number of memory DIMMs connected to each such processor chip. P5 and P6-based SMP systems all the way from 4-way up to 64-way are built using these basic building blocks. In such a system, even though cache and memory has some locality to each processor chip, 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.

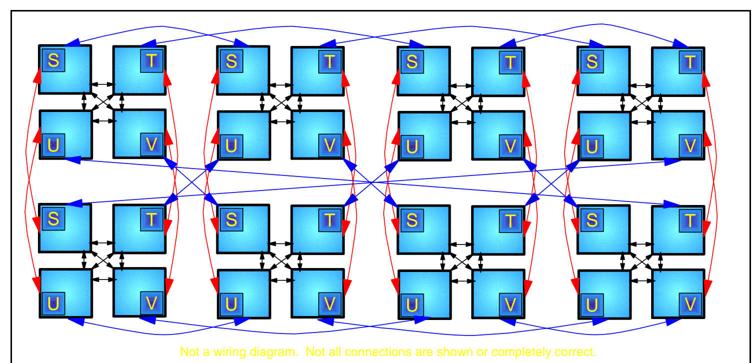


Allow me a moment to quickly define a "coherent cache" since it is key to understanding the performance effects of an SMP fabric. From an application's point of view, it is simply accessing the contents of main storage. But the processor core's know that they are really accessing the contents of the 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 making 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 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) from the contents of cache of one core to the cache 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 system will find the needed data, read it out of Chip T's cache, and transfer it across the inter-chip fabric to the core on Chip S; this is all without first going to memory and the data's real location there.



The distributed approach used for both main storage and cache allows for massive bandwidth as well as allowing many accesses to have no performance impact on similar accesses done on other chips. But it also comes at a performance cost; an access - a cache fill - satisfied from cache or main storage which is local or nearby the core doing the access 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.

Accesses across some types of links are also faster than others. A variation on this is found on the P6 high end systems which link more chips together with the lower latency links between them (as in the 64-way at right), but the performance effects of relative access latencies applies here as well.



A key notion here is that memory access latencies are variable; they are variable based on the location of the data or instruction stream that is being accessed. This is a limited instance of a CC-NUMA architecture, Cache-Coherent NonUniform Memory Access. Relative to a cache fill from memory local to the core doing the access, a cache fill from memory in the same physical drawer takes roughly 20% longer; it takes very roughly twice as long if found in another drawer's memory. The i5/OS knows this and attempts to maximize local accesses, but in some applications there will be some considerable percentage of accesses to "remote" memory. Because of the i5/OS' management, many applications might succeed in having their accesses be heavily weighted toward "local" access.

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. So if you or your application can change the balance toward more local accesses, performance - and the capacity of the entire system - will improve. Indeed, i5/OS 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 i5/OS 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, and by taking this into account when defining partitions is likely to also increase available system capacity.

Of 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 see these effects. The only real issue is the frequency in which they 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 there 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. So what of CPW? And how does it compare 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 transactions be executed against the contents of a large handful of database tables. From a processor utilization point of view, you can think of each individual job/task 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.

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 a particular chip. 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 in 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 proper. 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).

In Conclusion

So what of MIPS 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, 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.

IBM, of course, publishes CPW ratings for each system supporting i5/OS. 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. 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 i5/OS has provided you with enough information to know.

... As An Assignment Left for the Interested Student

You've seen that P6's cycle time is considerably better than that of P5. You've seen that the performance of a system is much more than just cycle time based. But you've also seen that it is possible that the broader use of a larger SMP system can limit itself in ways that do not frequently use the full characteristics of the system. You know that CPWs - for example - do not scale up linearly with cycle time. So 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?