Under the Hood: Simultaneous Multi-Threading on POWER7 Processors

January 28, 2010

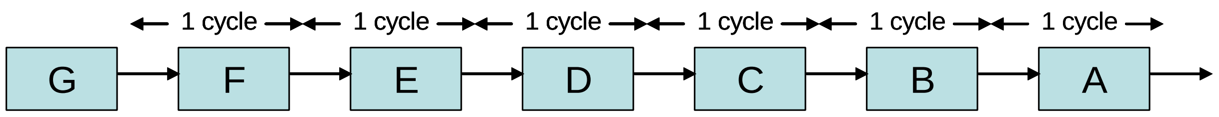
Disclaimer – Simultaneous Multi-Threading on P7 Processors

Simultaneous Multi-Threading on POWER7 Processors

PowerPC Server processors have supported Simultaneous Multi-Threading (SMT) for a number of processor designs now, starting in SMT's current form with the Power5 processors but initially in an earlier form called Hardware Multi-Threading (HMT) in the Northstar processors (processors even preceding POWER4 designs). With POWER7 its time for us to write an update; POWER7 processor cores support SMT4, a concept we will be describing here.

Let's start with an overview of what SMT really is and then we will get into SMT4.

What you know for sure is that processor cores execute instructions, instructions defined by the PowerPC Instruction Set Architecture (ISA). You've likely got some inkling that the processor executes these instructions at a rate approaching one instruction per processor cycle and you probably picture that as meaning that an instruction executes from beginning to end in this cycle. (A cycle, BTW, is the inverse of the processor frequency; a 4 GHz processor - 4 billion cycles per second - represents a ¼ nanosecond processor cycle.) Although this rate of execution - 4 billion instructions per second - is a rough first guesstimate, the truth is that instructions do not execute from beginning to end within the time of one cycle. Instead, instructions execute through multiple stages before complete, each typically one cycle long, from the point from where they begin their execution to the point where the processor core treats them as complete. Think of this sort of like an assembly line; each one-cycle long stage does something with an instruction before passing it off to the next stage. What is cool is that - as with an assembly line - once an instruction has gone through one stage, a subsequent instruction can execute using that now available stage. You can see this in the following simple figure showing the execution of an ordered instruction stream of instructions A through G. This is called - rather descriptively - a "pipeline".



This single pipe allows for a maximum instruction execution rate of one instruction per cycle. Recent server-based PowerPC processors support multiples of such pipes, allowing independent instructions to execute in parallel. POWER7 supports

- two pipes for executing the instructions which access storage (i.e., instructions loading/storing registers from/to cache),
- two pipes for executing arithmetic instructions on the contents of registers (e.g., Add, Subtract, Compare, AND),
- a pipe for branch instructions (i.e., instruction stream control flow),
- as well as parallel support for floating-point and vector operations.

This enables a remarkable amount of capacity for concurrently executing instructions. Just looking at the first five pipes - those used most frequently - it is quite possible to execute instructions at a rate of **five instructions per cycle.** In most programs, executing at rates of multiple instructions per cycle is quite typical for shorts bursts of time. But on average over an entire program, a single thread executing its instruction stream, it is more typical that the rate of instruction execution is very roughly **five cycles per instruction**. This 25X difference means that, rather than the pipes stages being typically very busy, the pipe stages are on average relatively sparsely used; a lot of each processor core’s capacity is simply going unused.

So why the big difference and what does this mean to SMT? There are quite a few reasons for the difference but two are typical, cache misses and inter-instruction dependencies:

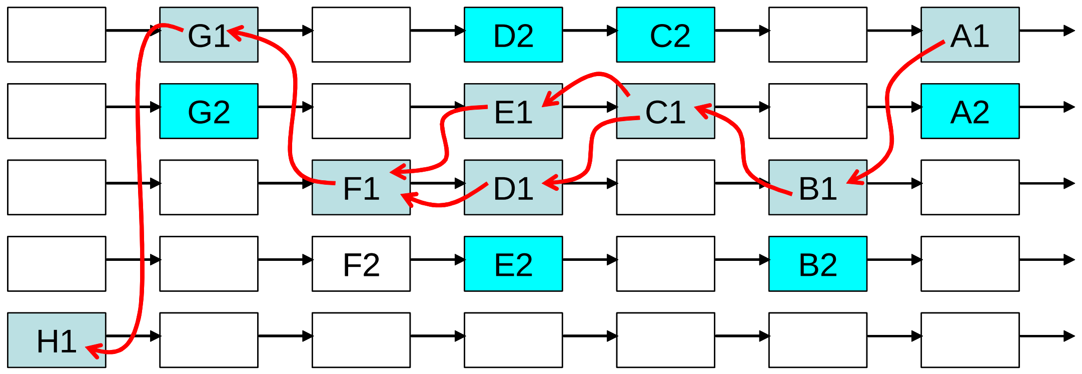
- **Cache misses** ...The reality with processors is that the cores do not execute instructions directly out of memory; the core executes instructions out of the instruction cache. In the event of an instruction cache miss, no more instructions are fed into any of the multiple pipes until more instructions have been filled into the cache; this can result in a delay of tens to hundreds of cycles, all the while the pipe stages are sitting idle waiting for the next instruction. The same is largely true for data cache misses as well. Simply stated, when a thread incurs a cache miss, the thread still remains associated with the core, but its instruction stream execution temporarily ceases instruction execution. Part of what SMT buys is the capability for one or more other threads to execute their instruction streams during such delays.
- **Instruction dependency delays....** The compiler knows all about the multiple pipes of the cores and attempts to lay out instructions to execute independent instructions in parallel. It is often successful. However, any program is also likely to have dependencies between the instructions; a LOAD of data into a register is followed by a dependent ADD, which feeds a dependent COMPARE, which feeds a dependent conditional branch. Such dependent instructions cannot be executed in parallel in the same cycle; they are delayed by the hardware until the results of a preceding instruction are available to be fed into a following instruction. Such delays are individually minimal but also very frequent. The result is that pipe line stages which might have been used by independent instructions here instead leave pipe stages unused because of the dependencies. This is not bad, it happens quite frequently. But what it also means is that there are frequently pipe line stages - read that core capacity - left unused which could be used by the instruction stream of another thread.

So what SMT enables is the concurrent execution of the instruction stream of multiple threads on the same core. Their instruction streams, which are typically independent, really are being executed concurrently; if a pipe line stage is not being used by one thread’s instructions, another thread’s instruction stream gets to use it. Conversely, if multiple task’s instruction’s want to use the same pipe line stage, only one gets to use it in that cycle, the other task’s instruction(s) are momentarily delayed. This distinction is important.

For example, consider the following figure. Where a Thread 1’s instructions A-H could execute in parallel in just a couple cycles if independent and of the right types, we see here instead that the inter-instruction dependencies require more cycles to execute the same number of instructions. But more importantly for SMT, it also means that there are a lot of pipe line stages simply not being used by this one thread’s instruction stream.

So, as in the following figure, let’s add one more thread’s instruction stream, making it identical to the first to show an effect.

Notice that the number of cycles spent by Thread 1 remained unchanged even with the addition of Thread 2's instruction stream because the core's pipes were able to provide pipe stages to both instruction streams as needed. Of course, that is not always the case; where a pipe line stage is not available when needed a thread must wait. In general, this typically means that both thread's execution speed becomes slightly slower.



For POWER7 processors, instead of just two instructions streams, POWER7 has support for 4-way SMT; up to four thread's instruction streams can concurrently use a core. As can be seen in the above figure, there still remains some - but now less - additional pipe line capacity to take on the additional independent instruction streams of a couple more threads. But it is just as obvious that there is diminishing opportunity for additional threads. As additional threads are added, each individual thread is slowed by the competition for processor core resources. But slower or not, the fact that they can execute concurrently - as opposed to waiting their turn for an available core - typically provides more performance capacity.

It should be obvious now that 4-way SMT (within a single core) does not provide the same capacity as 4-way SMP (i.e., four independent cores). What SMT does is to allow for the use of the compute capacity within each core that normally can't be consumed by a single thread's instruction stream. Where in SMT each of the multiple threads is competing for the use of a single core's resources, when executing alone on separate SMP cores these same thread's instruction streams each get the full use of the core.

The IBM i OS is well aware of this difference and of the performance characteristics of SMT. It also knows that there are often periods when fewer than the maximum number of SMT4's hardware threads are actually executing instructions; an 8-core partition also means 32 hardware threads and the potential for 32 concurrently executing software threads. When there are fewer dispatchable tasks, tasks are often first dispatched to cores which either have no or the fewest number executing; tasks are spread over the cores available on a chip to allow them to individually execute more rapidly. Only as utilization increases - as the number of tasks needing to execute concurrently increases - does POWER7's core switch to support the increasing number of instruction streams.

Observations on SMT Performance

As you think about SMT you quickly realize that both the performance of individual threads and even the overall capacity available from moment to moment is dependent on what it is that the hardware threads are doing. A thread, which when executing alone might have used every stage of every pipe for a while, does not leave much capacity on that core for even one more thread, much less two or three more. On the other hand, one thread with a fairly high rate of cache misses leaves a lot of pipe line capacity available for a threads with a higher rate of executing instructions. It happens that the way each thread uses a core's resources typically varies rapidly from moment to moment. And the core gets used a lot of different ways by even one thread over a longer period of time. As a result we can speak of the average benefit to capacity that SMT provides, but please realize that the use and so benefit of it can be quite variable.

With that in mind, we typically speak of the capacity opportunity of SMT4 (4-way SMT) over ST (Single Thread on a core) as being in the range of 1.5 - 2X. For example, let's define a thread which, when executing alone, produces some throughput of - say - 100 transactions/second. If we replicate that thread four times and place them all onto the same core, together they are said to generate 1.5 - 2X of the throughput - so 150 - 200 transactions/second. Notice, again, that these four tasks placed individually on separate cores would approach 4X more throughput. With SMT4 it is possible, but not typical, that for some classes of work that the throughput improvement of SMT4 over ST can approach 4X. Conversely, it is also possible for there to be minimal benefit when core resource competition is extreme. The point is that we typically see some considerable capacity benefit from the use of POWER7's SMT4 and highly recommend its use. But it is also true that the average benefit of SMT4 can vary considerably.

As the instruction streams of multiple threads compete for the resources of a core, the threads individually slow down. So, how much? We offer here a rule of thumb. Let's suppose a workload with a 2X throughput improvement of four tasks on the same core over the throughput of just one. Let's also suppose that these same four threads, when placed individually on separate cores, produced a throughput improvement of 4X. The single-core SMT4-based tasks are producing only half the throughput of the four core-based tasks. This implies that the four SMT tasks are executing individually at half the speed on average; on average each task is taking twice as long to execute the same thing. Of course, when there are three, then two, then just one task remaining on a core, the individual task's speed improves. Clearly individual task performance also can vary quite considerably, depending on what the other hardware threads are doing but more so on whether other tasks are even present. But let's now reconsider the notion of improved capacity. Yes, the individual threads might be slower, but they are slower **while executing on a processor**. So suppose that SMT4 did not exist. The same tasks which are executing slower with SMT4 are nonetheless executing; without SMT4, they would be taking their turn waiting and competing for the use of the same core that they are executing on with SMT4. So slower while executing, yes, but faster because of decreased queueing up behind the cores.

Observations on Virtualization

Virtualization offers a means of using the capacity improvements of SMT in shared-processor partitions.

Virtualization includes a concept called a "Virtual Processor". The virtual processor is what gets assigned to a core; it is permanently assigned to a core for dedicated-processor partitions and temporarily for shared processor partitions. For POWER7's SMT4, assigning a virtual processor to a core also brings with it all four hardware threads. If one or more virtual processor's hardware threads are not being used at some moment, if its capacity is not being used, the target core's capacity is also not going to be fully used.

It is also true that a shared-processor pool can have assigned to it many more virtual processors than there are cores in that pool[1]. This occurs naturally from the existence of fractional-core capacity partitions; for example, even with one virtual processor in - say - ten partitions with 0.1 core's capacity results in ten virtual processors using one core's capacity.

The value of maximum capacity defined for each shared-processor partition is used as a limit of what each partition is allowed. To manage this the hypervisor measures the amount of capacity used; once used in a time slice, that partition might not be allowed any more processing time until the next time slice. It does this measurement by determining how much time each partition's virtual processor(s) were assigned to a core. In the context of SMT, whether there are one, two, three, or four active hardware threads, whether all of the SMT-perceived capacity is being used or not, the hypervisor only tracks how long a virtual processor is assigned to a core.

Now recall that for reasons of the performance of individual threads, the OS intentionally spreads the dispatchable tasks over the available cores, here meaning over the virtual processors. As you have seen, this intentionally leaves some of each core's capacity unutilized for the benefit of individual task's performance. For dedicated-processor partitions, this can[2] remain exactly the right thing to do. For shared-processor partitions, it is not as clear whether spreading tasks over cores is always appropriate. Since there can be a lot more virtual processors active than cores in the shared-processor pool, it also means that these virtual processors will compete for the use of these cores; some will execute while some will simply wait. Now picture all of these virtual processors intentionally executing just one thread; they are executing just one thread each because each partition wants these threads to execute faster, but since many virtual processors are simply waiting to use a core, some of these threads are temporarily not executing at all. And for those that are executing, the single threads are often not really consuming all of the capacity available in the cores.

---

[1]Please observe, though, that it is never a good idea to define any single shared-processor partition with more virtual processors than there are cores in the shared-processor pool.

[2]Energy management considerations might suggest otherwise.

One trick to buy back some of this capacity is to decrease the number of active - as opposed to actual - number virtual processors. This decreases the contention of an excessive number of active virtual processors being time sliced over the fewer available cores; fewer active virtual processors decreases wait time. To acheive this for shared-processor partitions, rather than each partition spreading its tasks over the available number of virtual processors (i.e., rather than optimizing for the performance of individual tasks), the OS can spread less and start using SMT sooner. This is called "Processor Folding". The same number of dispatchable tasks can still make progress, albeit slightly slower when executing, but on fewer virtual processors, but wait time will decrease as well.

What this also does is good for the shared-processor pool as a whole; more of the pool's capacity is able to be used. Not only are the cores being used as desired, but the capacity within the cores as well. There also happens to be an upside for the shared-processor partition's themselves. Because of processor folding, each partition activating fewer virtual processors, the hypervisor <u>perceives</u> that each partition is consuming less of its allocated capacity and as a result its virtual processor(s) can remain attached to cores for a longer period of time. Further, yes, the tasks execute individually slower due to increased SMT usage, but as a result they also get to be assigned to a core for a longer period of time, allowing them to potentially complete their execution sooner. Because of the decreased switching of virtual processors onto and off of cores, this also has a side effect of decreasing the thread's cache miss rate, indirectly further speeding the thread's execution.

Observations on the Measurement of CPU Utilization

It used to be that measuring CPU utilization was easy. In the days of single-processor systems, CPU utilization merely meant finding out what percentage of the time a processor was or was not being used. In the slightly more complex days of SMPs, the system would add up the amount of time that each core was being used in a measurement period, divide that by the number of cores, and divide that by the measurement period. Converting this to percentage and subtracting this from 100 provide the remaining capacity percentage.

SMT throws a monkey wrench into the workings. SMT provides a way to consume more of the capacity of each core. So that means that there is capacity to consume and that needs to be measured and factored into the measurement of CPU utilization. So how does one determine how much capacity remains in each core even when one or more hardware threads are executing instructions. And, conversely, how much of the capacity was used and by who? It happens that this is a tall order.

Prior to SMT, one could measure the CPU cycles consumed - the value used to calculate CPU utilization - and from there know how much time a particular task had spent on a processor. So when an SMT4 processor can have four different tasks executing at least one instruction from each task during the same cycle, which of these tasks is perceived to have used a processor cycle?

The SMT processors - POWER5 through POWER7 - offer an internal mechanism for tracking relative usage of each core. POWER7 took it a step further and provided a means of tuning this measurement. The general intent is to provide a measure of CPU utilization wherein there is a linear relationship between the current throughput (e.g., transactions per second) and the CPU utilization being measured for that level of throughput. For example, a throughput of 100,000 transactions/second at 50% utilization should imply that at 100% utilization the throughput should be able to reach 200,000 transactions/second.

This internal mechanism attempts to proportionally assign fractions of cycles to each of the four hardware threads. This is true whether the hardware thread is executing a Run State task or is idle (i.e., idle meaning executing a Wait State task or simply not being used). Since each Run State task executes slower as additional Run State tasks are added to the core, the mechanism adjusts cycle assignment accordingly. As a result, although there can be no perfect means of measuring CPU utilization with SMT, POWER7 is producing the desired near linear relationship between throughput and CPU utilization for a number of typical workloads. However, for atypical workloads seeing either little or a very large benefit from SMT4, the desired linear relationship can become nonlinear.