IBM eserver

# Simultaneous Multi-Threading (SMT)

# on eServer iSeries POWER5™ Processors

Mark Funk
iSeries Systems Performance
May 2004

**For the latest updates and for the latest on iSeries performance information, please refer to the Performance Management Website: http://www.ibm.com/eserver/iseries/perfmgmt.**

## Table of Contents

## Introduction

The IBM® POWER4™ processor introduced two processors embedded in a single chip, sharing a common L2 level cache to increase processing efficiency. The POWER5™ processor builds on this topology with a significant enhancement called **Simultaneous Multithreading** (SMT), a concept where multiple threads of execution can execute on the same processor at the same time. IBM introduced a similar concept called **Hardware Multithreading** (HMT) in its Star line of processors, but did not include HMT in POWER4; it instead relied on dual processor chips to meet its performance objectives. Now with POWER5 processors, IBM brings the best of both worlds together, by keeping multiple processor cores per chip, with all the density-compute advantages, and going back and significantly building upon the efficiency and hardware cost "lever" provided by HMT. And, IBM has enhanced the processor to deliver full SMT.

With the Star line's HMT, a processor could support two streams of instructions (two threads, a.k.a. tasks) by alternating between the two whenever one experienced a long delay. HMT enabled additional capacity by simply executing one task's instruction stream while the other task was experiencing a long delay such as a cache fill. This plays on an ever increasing effect in processor designs; processor cycle times are speeding up faster than storage access latencies. Although the HMT processor did manage two tasks, at any given time only one task executed instructions. In that sense only, HMT can be thought to have two processors per chip. Chip multiprocessing within POWER4 took a different approach by providing double the amount of processors on a chip (one thread per processor), double the number of resources completely available to attack the user's problem set, each having its own L1 cache, but sharing a common L2 cache. The POWER5 series improves on the multithreading capabilities of the Star line within a processor topology similar to POWER4's, so that each processor can execute two instruction streams at the same time, without switching between them as the Star processors did. When operating system's task dispatcher looks at this chip, it "sees" four available processors, four places onto which to dispatch a task: two physical processors, then because of multithreading, two logical processors per physical processor.

In a manner similar to HMT, SMT reintroduces the notion of a single processor that is capable of executing multiple threads. And as with HMT, simultaneous multithreading alters the notion of system performance, capacity, and processor utilization. But as you will see, SMT is also fundamentally different than HMT. It is implemented on the POWER5 processor with quite different processor pipes, cache sizes, and topologies than the processors upon which HMT is built. Therefore, SMT needs to be treated as more than just another version of HMT. Indeed, performance modeling predicts that SMT should enable a roughly **35-40%**[1] performance capacity increase over identical processors not using SMT[2], a performance improvement value considerably larger than was measured for HMT.

---

[1] For a database transaction processing benchmark with one-to-16 processors. The capacity increase available to other environments varies.

[2] For instance, POWER4 processors do not support SMT but have otherwise relatively similar designs. Further, as you will see, it is possible to disable SMT on the POWER5 processors.

It is the intent of this document to present SMT and its performance capabilities, as well as to introduce a set of topics of which you should be aware.

## SMT Overview

As is true with POWER4 processors, the POWER5 processor supports two load/store units, two arithmetic units, and a branch unit, as well as two floating-point units. Any instruction flowing through these units takes multiple processor cycles (a.k.a., stages) to complete. In a given cycle, the processor is used most productively if every stage of each unit executes a different instruction. This, however, happens very rarely for the typical workload, so a large majority of the idle stages go unused. This leaves a great deal of the processor's capacity still available to improve performance. Many of the stages not used by one thread can be utilized by concurrently executing the instruction stream of another thread (i.e., another entire control flow). This is the basic notion behind SMT; the underutilized processing capacity resulting from the unused processor stages can be used by another thread. Rather than consecutively executing portions of each thread's instruction stream, SMT enables concurrent execution of multiple instruction streams.

In POWER5's SMT, each physical processor can simultaneously support two threads. Each of these two threads is said to execute on a "logical processor", two logical processors per physical processor. The instruction stream for both threads are accessed from the same L1 instruction cache and issued, jointly, to the processor's units (i.e., the processor's pipes). Of course, there is conceptually, and in some cases, physically a separate register space for each thread.

Ideally, the instruction stream of both threads is being read from the instruction cache, flowing through the pipe line stages, and using the physical processor's resources without contention between the threads. When this happens, SMT can theoretically double processing throughput. Clearly though, a given stage in a particular unit can be used by only one thread per cycle; other processing resources also exist that get momentarily consumed by each thread. So if both threads have instructions contending for the same pipe stage or resource, an instruction for one of the threads must wait. Given that both threads never vie for common processor resources (such as pipe stages), you can see that it is possible, though not likely, for both threads to execute as quickly as one.

To understand this better, let's take a look at how instructions are issued and dispatched. Multiple instructions of either thread can be issued into the set of pipes in any one cycle. Any five instructions representing up to two load/store, two binary arithmetic, two floating-point arithmetic, and a branch can be issued per cycle, often from both threads in the same cycle. Only one instruction from either thread can be issued into a given pipe per cycle. Once issued, instructions proceed through the pipe stages in this order. The physical processor determines the issue order of instructions based on a number of rules which take into account (1) operand dependencies, (2) how long instructions have been waiting to be issued, and (3) to allow instructions to execute out of order where appropriate. Think of it as a funnel; the funnel is filled by dispatching instructions from the instruction cache at a rate of multiple instructions per cycle and emptied by issuing instructions into the set of pipes at a rate which is optimal for these pipes

while meeting the functional requirements of each thread's instruction stream. Instructions are dispatched into this funnel at a rate of up to five instructions per cycle; a given dispatch cycle only contains the instructions of one thread. All of this takes place within one physical processor.

Each physical processor contains an L1 data cache and an L1 instruction cache, so they are shared by both logical processors. A much larger L2 cache is common to POWER4's two physical processors and POWER5's four logical processors. So with SMT, four threads share the same L2 cache. Multiple cache fills that are incurred by multiple threads - on both logical and physical processors - can be concurrently processed.

With HMT, instructions could still execute during a a long processor delay (e.g., cache fills from main storage), but there is an explicit processor-initiated thread switch on such long processor delays. Of course, cache misses and such (relatively longer) long delays in instruction execution also occur on POWER5 processors, and again one thread's instruction stream can continue executing while one or more cache fills for the other proceeds. The thread incurring the delay requires less use of the processor pipes during that delay; the result being that the remaining thread can then use nearly every stage of the processor pipes until it, too, incurs a long delay or the long delay incurred by the first thread is complete.

## SMT Observations and Associated Design

As you can see, SMT has some tradeoffs from a performance point of view. Overall two threads can concurrently execute on the same processor, so one need not wait as long for an available processor. However, the contention for common resources, such as pipeline stages, would tend to slow an individual thread's execution speed. If pipeline stages and other processor resources are available when needed by each thread, the two threads can execute as efficiently as one; the two threads could take no more time than one. When there is contention, a thread's execution is slowed. But, although SMT may slow an individual thread, the capability of executing multiple threads at the same time can significantly increase the system's throughput, i.e., its capacity to take on more work. The usage of the processor cache's is also changed by SMT and this too affects performance gains.

- With SMT, two threads now share the 64-kilobyte L1 Instruction-cache and 32-kilobyte L1 Data-cache found in each physical processor. L1 cache sizes are the same as on POWER4 processors. Given that there is only one thread running on this physical processor, all this cache is available to this one thread. When running two threads, each thread's state must reside in this shared cache, effectively decreasing the size of the cache seen by a given thread and possibly slowing processing. Inter-thread cache thrashing is also possible. But there is also the potential for inter-thread sharing of the same data or instruction stream held in L1 cache lines, giving a relative performance boost with SMT. The amount of this sharing —and conversely, the contention for cache resources— directly affects the additional capacity implied by SMT's logical processors. But the capability to execute one thread's instructions while the other thread's cache fills are "in process" acts to offset some of the effects of cache contention.

- With SMT, four threads now share the 1.9-megabyte L2 cache—an increase from POWER4 processor's 1.4 megabytes. As with the L1 caches, there is sharing between these threads and some contention for cache lines. Relating to the sharing, think of it like this; with POWER5's SMT, four threads can be concurrently changing data held in their common L2 cache. Without SMT (say on POWER4 processors) these four threads are executing on, at best, two different chips with different L2 caches. Changes made to this shared data results in a delay to move the data between each chip's L2 cache. But with SMT, no data movement between L2s is required since the state of all four threads' data is held within this one L2 cache. Conversely, where there is no sharing between the threads, the cache space being used by four threads with SMT is effectively smaller than the cache used by two threads not running SMT.

Clearly, SMT provides a potentially significant opportunity for more performance capacity. Where there is no contention for shared processor resources, each logical processor can execute as well as if it were a separate physical processor. But there is also an opportunity for unfortunate performance effects. This is dependent upon: (1) the type of workload, (2) the level of cache sharing between the threads that are simultaneously multithreaded, and (3) the coding style producing the instructions to be executed concurrently. Even within a database transaction-based workload, where the average benefit is modeled to be in the 30-40% range, both the ideal and

"not-so-ideal" effects of SMT will be experienced. Largely due to cost and chip size issues, sharing of a physical processor's pipes and pipe stages is an efficient way to increase the performance capacity of a system. But as you can and will see, the measured gain that results from its use is also dependent on the environment. Due to SMT alone, some environments will experience more capacity, others less.

As you can easily see, SMT does not speed the execution of any given task. You will realize no benefit due to SMT alone, and in reality, you may see less performance for a single-threaded application. SMT improves throughput in much the same way as do additional processors, by allowing more tasks to execute at the same time. When every logical processor is used, the amount of performance improvement provided by SMT is seen either because a greater percentage of the work is done in the same amount of time, or all the work is completed in less time. Response time, though, can also be thought to improve since a "logical" processor is more likely to be available when needed - lessened processor queuing effects - either because of the additional "logical" processors or because work other than your own task got done more quickly.

Please realize that SMT is not SMP (Symmetric Multiprocessing). Although both SMP and SMT scale through concurrent execution of multiple threads, they scale system capacity for quite different reasons. In SMP environments, not every processor is used all the time. At some level, SMT may appear to simply double the number of processors. But as you've seen, a thread which runs alone on a physical processor typically runs faster than one sharing a physical processor with another thread. For that reason, at lower utilization, the operating system, specifically the Task Dispatcher, tends to allocate only one thread per physical processor. As system utilization increases - as the number of dispatchable threads exceeds the number of physical processors - both logical processors get used.

When any processor, including an SMT's logical processor, is not doing real work, that processor is nonetheless executing instructions on behalf of a "Wait State task[3]." Often, system utilization is such that only one of the two SMT logical processors will happen to execute a work task[4]; the other logical processor executes the instruction stream of the Wait State task, instructions that are repeatedly executed while that logical processor polls for useful work. The instruction stream of the Wait State task contends for physical processor resources just as would two work tasks, effectively slightly slowing the processing of the work task on the same physical processor. For this reason, the SMT hardware also supports the notion of "priority." Instructions from the lower priority Wait State task are dispatched at a slower rate. This means that the work task gets a large majority of the available dispatch cycles and therefore use of most of the processor resources. This clearly helps the performance of the work task. But it also has an effect on measured CPU utilization, a subject which will be discussed shortly.

---

[3] The Wait State task typically executes nothing more than a small amount of code which continually asks the question "Do you have anything for this logical processor to do?"

[4] A "work task" is essentially just the active part of an application or program.

## SMT Operational Modes

Three SMT modes of operation are offered via the system value **QPRCMLTTSK**[5]**,** the same one used to control HMT:

- ***OFF** ... As the name implies, no processor controlled by this operational mode will use SMT. The hardware will be set to Single-Thread (ST) Mode with the result being that only one thread is ever dispatched to a physical processor and the full resources of that processor can be used by that thread. It is as though the capabilities of SMT discussed earlier do not exist. When *OFF is set, it affects every processor used by that partition.

- ***ON** ... The task dispatcher can dispatch up to two threads onto a single physical processor. However, dispatching a task to a completely unused physical processor is typically better for performance than dispatching a task to a physical processor that is already executing one thread. A wait state task will execute on any unused logical processor, and will run at a lower priority than a work task. When *ON is set, it affects every processor used by that partition.

- "**System Controlled"**... This mode offers an additional capability over that of *ON. It offers the opportunity for the operating system to temporarily mark a physical processor as being in single-thread mode when there is only one work task on a physical processor. The intent is to allow single tasks to execute slightly faster when the operating system suspects that both logical processors will not be needed. If a physical processor is in this single-threaded mode when a second task needs to be dispatched here, the physical processor is returned to SMT mode. The measurement of CPU utilization might produce less predictable results in this mode than with the others. This value is the default.

Unlike HMT, POWER5's SMT mode is associated with an operating system, not with the Central Electronic Complex (CEC) proper. Where logical partitioning (LPAR) enables multiple operating systems, each partition can set its mode differently. Indeed, where shared processors are used, the processor's mode changes to that of the current partition. It is a physical processor - a pair of logical processors together - which are switched between operating system partitions.

The POWER5 hardware proper supports a mode - single threaded (ST) - where all of the resources of the processor (all cycles, all registers) can be thought of as being assigned to only one of the logical processors and, therefore, to one thread. In this mode, only one thread executes on the physical processor. Because of the lack of contention for hardware resources, a work task executing in ST mode has the opportunity to execute slightly faster than in SMT mode. Response time for a particular work task in ST mode is typically better even if, when in SMT mode, it is only ever sharing the processor with a Wait State task. SMT adds to system capacity, not single task performance. So, as noted earlier, a physical processor's ST mode can be used, either when it is explicitly required by the **QPRCMLTTSK** system value (*OFF), or when this system variable indicates that the system can decide. The hardware's ST mode is never used when **QPRCMLTTSK** = *ON. When the system decides to switch the physical processor to ST mode,

[5] At this writing, changes to the QPRCMLLTTSK system variable require a partition IPL to take affect.

it is because the utilization of the processor(s) is low enough SMT mode for that processor will not soon be required. The decision to automatically switch to ST mode is complicated by processor utilization measurement issues. The usage of ST mode also impacts the measurements of processor utilization and capacity. This concept will also be discussed as part of a later section, after we cover issues relating to measuring CPU utilization.

## Processor Utilization Concepts

The advantages of SMT, as compared to ST on the same hardware, are environment-dependent. Determining the benefit is fairly straightforward when every processor (and logical processor) is utilized 100% of the time. As mentioned earlier, the benefit can be estimated by noting that x% more work is being done in a fixed period of time, or that a set amount of work represented by multiple tasks is being completed in y% less time. There is no way to determine the increase in capacity due to SMT alone except through experimentation or direct comparisons to previously published results[6].

Often, though, average CPU utilization is commonly used to determine the amount of performance capacity[7] still remaining in a server, as in:

**Capacity Remaining % = 100 % - Current Percent CPU Utilization**

Prior to SMT, the notion of CPU utilization could essentially be represented by way of the percentage of a period of time that a processor is not in its wait state. Or said differently:

**Capacity Remaining % = Percent CPU Utilization of Wait State Task(s)**

From an SMT point of view, CPU utilization of a physical processor could be interpreted as the time that both logical processors are not in their wait state. If either of the two logical processors is executing a work task, the entire physical processor would be viewed as being utilized. By this definition, if exactly one logical processor always executes one work task, then CPU utilization would be presented as 100%. Clearly though, when there is only one work task, there is also one Wait State task. So the utilization should be something less than 100%; the executing Wait State task implies that available capacity exists. There should be a way of noting the remaining capacity perceived by the Wait State task. Ideally, given the Wait State task was replaced with a work task, there is a need to know the available percentage of processor cycles that it could consume. You can see that this technique would overstate CPU utilization.

Alternatively, each logical processor could be viewed as just another SMP processor. The amount of time that any work task is executing on a logical processor divided by the total measurement period is just the definition of normal SMP processor utilization, quite independent of SMT. If the system only had one physical processor (two logical processors) this would read:

$$\textbf{CPU utilization} = ( (\textbf{RunCycles}_0 + \textbf{RunCycles}_1)/2 ) / \textbf{Cycles\_In\_Measurement\_Period}$$

This equation implies that if only one or the other logical processor was ever used (but not both), utilization would be 50%. This would further imply that the remaining capacity is also 50% or

---

[6] Indeed, even at 100% utilization - meaning that every logical processor is being used all of the time - there might actually be slightly more capacity available. As with practically any thing else in a computer system, this additional capacity becomes available through the efficient parallel usage of the many available processing resources.

[7] In this document, we use the term "capacity" to refer to the server's capability to accept additional work.; Whereas, "capacity" more usually means the total amount of work the server is able to accomplish..

that the system was still capable of doing twice as much work, an unlikely situation for simultaneously multithreaded logical processors. So this alternative for measuring CPU utilization overstates available capacity (and understating CPU utilization).

Instead, in an attempt to avoid both under and overstating the available capacity, the POWER5 processor adds a register called PURR (Processor Utilization of Resource Register) to each logical processor. The processor hardware assigns each and every dispatch cycle within a physical processor to one, or the other, logical processor, doing this by incrementing one or the other logical processor's PURR per cycle. A logical processor's PURR is incremented by one for each cycle in which instructions are dispatched on behalf of its thread. When there are unused dispatch cycles, hardware decides to which thread the cycle should be assigned, incrementing that PURR. This works relatively well when both threads are work tasks; their sum still represents total actual work. When one of the logical processors is executing a Wait State task, cycles are assigned to its PURR like any other task[8]. And it is the counts assigned to either of the two Wait State task's PURR that ultimately represent the remaining processing capacity of a physical processor.

For each logical processor, the CPU utilization of that logical processor is the time not spent in its Wait State task. The CPU utilization of a single processor over a time period M then is

**((M - (Wait_State_PURR1 + Wait_State_PURR2)) / M) * 100 %**

where Wait_State_PURRx is the time the logical processor spent in its Wait State task during the time period M. Since only one, or the other, PURR is incremented in any given cycle, if both logical processors are only executing in their wait state, then

**M = Wait_State_PURR1 + Wait_State_PURR2**

So, as you would expect, remaining capacity is 100% and CPU utilization is 0%.

Total system utilization in a multiprocessor system then, when using dedicated processors[9], is essentially the average utilization of all the processors using the above definition of CPU utilization. CPU utilization represents any usage of a processor other than the Wait State task, even if the processor is only waiting (filling the cache from main store for instance); even a utilization of 100% does not necessarily mean that the processor is executing instructions. Interestingly, with SMT, the Wait State task's PURR absorbs cycles spent waiting on a work task's cache miss - representing these delay cycles now as capacity cycles - which would have been otherwise assigned to a work task as CPU utilization cycles. Said differently, through the use of the Wait State PURR, the long delay periods experienced by a work task are now counted

[8] To keep track of CPU utilization on a task-by-task basis, the PURR cycle counts assigned to each task are maintained within each task—as the task is switched out. This is also true for the Wait State tasks, the sum of their PURR ultimately representing the available capacity.

[9] When using shared processors, the measurement period "M" represents only that time during which an operating system is bound to a set of processors. We should also note that CPU utilization is a concept measured within an operating system; it does not necessarily span an entire CEC. It only counts the utilization of the processors owned by that operating system.

as available processor capacity. This seems an intuitively correct approach since a second work task could be executing instructions during such a delay (in a manner conceptually similar to HMT).

This form of calculating CPU utilization, even on a job by job basis, is what is used in support of the commands **WRKACTJOB** and **WRKSYSSTS**. And clearly, any other scheme is unable to "fairly" assign cycles amongst threads when work tasks are running on both logical processors.

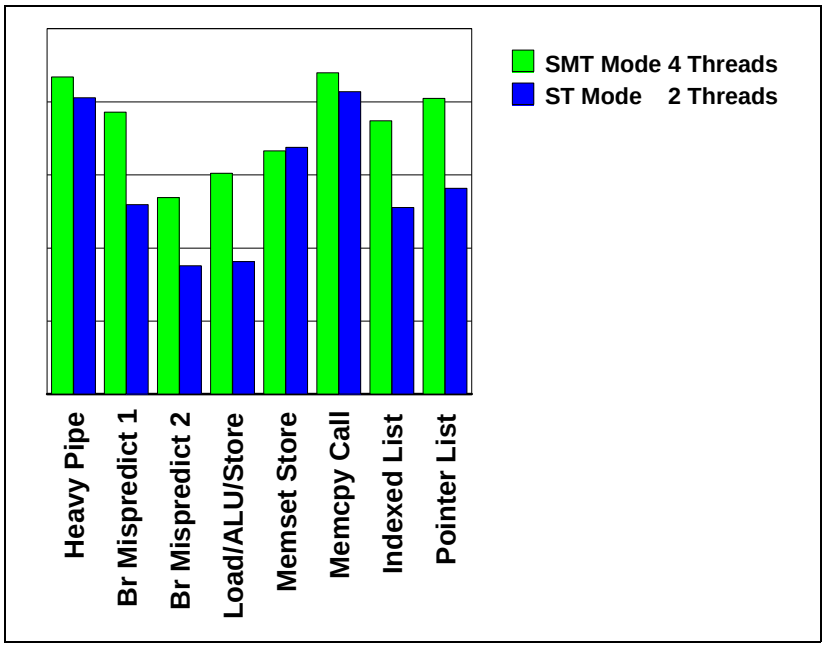
Although better than the alternatives, the PURR-based means of calculating CPU utilization is not absolutely precise either. On a thread-by-thread basis when running two work tasks, this mechanism is attempting to make a estimate of how each thread utilized physical processor resources. From a system wide point of view, for a particular physical processor, measured utilization is clearly absolutely precise when that processor is executing no work tasks; CPU utilization on that physical processor is 0%. Similarly, when both logical processors of a physical processor are both being used, utilization is just as clearly 100%. And these states are relatively common. It is when there is one work task and one Wait State task executing on a physical processor that absolute precision is not possible. Recall that it is the cycles assigned to the Wait State task's PURR that represents available capacity. But it is capacity as perceived by the "gold standard" of the Wait State task. But when a second work task replaces the Wait State task, will it be consuming the same number of cycles perceived as having been consumed by the Wait State task? For this is what "available capacity" effectively means. If a high degree of precision were expected, the answer to the question is really no. Available capacity and therefore measured CPU utilization is a measure of how the Wait State task interacts with a single work task on a physical processor and so is also a function of the characteristics of the work task. Since those characteristics change from moment to moment, momentary CPU utilization of that physical processor will as well. So once again, measured CPU utilization whenever running one work task on a physical processor is a function of the characteristics of the work task. But in a typical environment, anywhere from zero work tasks to more than the number of logical processors can be contending for processors at any moment in time, and this has an averaging effect on any "imprecision" occurring in other states. And, again, even though this is being described as imprecise, it is considerably better than the alternatives.

The discussion of the PURR, of course, relates primarily to SMT mode (*ON). When running in single-thread (SMT *OFF) utilization is calculated exactly like any other SMP; the physical processor either is or is not being utilized. Within a measurement period, the sum over all physical processors of the time that each physical processor was used is the CPU utilization.

## Measured Results

You've suffered through all of this theory. Let's take a look a some simple but actual results.

The purpose of the chart shown below is to study the capacity of a single POWER5 processor chip that consists of two physical processors by executing a set of primitive operations which focus on particular aspects of the dual-processor's design. Of course, in SMT mode, this means that there are four logical processors. The same stressful operation was executed by every thread, two threads with SMT's *OFF and four threads with SMT's *ON. For instance, one of the tests (see Heavy Pipe) had an instruction stream which attempted to use every pipe stage in every cycle. Others (e.g., Memset, Memcpy) attempted to maximize the number of stores. A few more studied the effects of more lightly used pipes, using branch misprediction and operand dependencies (e.g. Load/ALU/Store) to achieve this. Not unexpectedly, where a resource was heavily utilized in ST mode, SMT mode attempted to drive the resource even harder, so SMT mode offered limited additional capacity (over ST mode). But where resources such as pipe stages were available, SMT achieved between 45-65% more throughtput in these simple examples. We have also found that when two differing environments are concurrently running on the same physical processor (say cache misses with heavy pipe usage) you can often expect superior gains.




But these are primitives, operations that you can easily test yourself. More important are the actual workloads, most which ultimately consist of short-lived transitions through operations similar to these primitives. Lacking in these primitives, though, are the cache misses and their resulting long delays seen in actual workloads. As with HMT, much of the capacity advantage of SMT over ST also comes from this effect.

In larger environments, we are seeing that SMT produces ~35% in database transaction processing and Websphere workloads, ~28% in SAP, and ~ 45% in Domino R6 Mail.

## Related Concepts

### Memory Affinity

SMT also has some overlap with the concept of Memory Affinity. In 8-processor nodes (MCMs), SMT effectively creates 16 "local" processors, all sharing the same local cache(s) and main storage. The existence of these extra local processors would tend to increase the probability that a task can execute within its home node. But as we've seen, the performance of 16 threads executing on 16 logical processors is not twice that of 8 tasks executing on 8 physical processors. So suppose that a task becomes dispatchable when two processors happen to be available, one, a single logical processor on this task's home node, and the other, a full physical processor (i.e., no task executing on either of its logical processors) on a remote node. Is it better for the task dispatcher to use the partial processor on the home node, or to use the complete processor on the remote node, especially realizing that the other logical processor of the remote processor might shortly become used? There is a tradeoff between the additional capacity available on the single logical processor and the repeated latency of loading the state of this task into a remote node (and of restoring it to the home node later). For this reason, the task dispatcher tends to first use local logical processors before using remote processors.

### CPU Utilization within SMT's System Controlled Mode

The SMT mode of *SystemControlled adds another twist into the precision of CPU utilization. In this mode, the system has the option of temporarily switching a physical processor to support a single task if the system perceives a performance benefit for that task is possible. The system returns the physical processor to SMT mode when a second work task is to be dispatched there. Clearly, there is no concurrently executing Wait State task when there can be only one task executing on a physical processor. So for the period where the physical processor is only executing the one work task, the utilization of that physical processor is measured as being 100%. If SMT *ON mode were used, that same work task would be executing along with a Wait State task, this Wait State task would perceive available cycles, so the measured CPU utilization would be well less than 100%. So switching the physical processor to execute the single task alone both improved its performance AND increased the measured CPU utilization.

In order to further exemplify this point, consider a 2-way system in SMT mode executing four work tasks on the four logical processors. By definition, CPU utilization for this system is 100%. Next, suppose that one work task no longer needs the processor, leaving three tasks to continue. If in *SystemControlled mode, the system has the option of replacing the lost work task with a Wait State task or of switching, perhaps after a while, that physical processor to run just the one remaining work task. If the system switches to execute only one task on that one physical processor, leaving the other physical processor in SMT mode and executing two tasks, CPU utilization still remains at 100%. Only three tasks are now executing, one now executing slightly faster, but utilization remained unchanged. If instead, the system had switched in a Wait State task, CPU utilization would be measured as being less than 100%. As a next step, let's remove a work task from the other physical processor, leaving one work task on each physical processor. Without a Wait State task on either physical processor, CPU utilization would be measured as

still being 100%. Notice that if these remaining two tasks had instead been running on the same physical processor, the other physical processor running the Wait State task, CPU utilization would be measured as being 50%.

This effect may seem odd. But this example tends to be atypical in the long term. More likely is an environment where many tasks use a processor, each for a short period of time. So on this same 2-way system, 50% utilization is a mix of periods when there is no processor being used, and also when 1, 2, 3, or 4 logical processors are being used. As we've discussed earlier, the utilization measured when there is no processor usage and 4 logical processors being used is absolutely precise. It is less precise for cases of 1 and 3 processors being used, but throughput is up as well. If you were to graph this effect (i.e. CPU utilization vs. Transaction counts), there are cases where the increase in measured CPU utilization is proportional to the increase in transaction counts. There are also cases where CPU utilization will be measured as being proportionally greater than the increased transaction counts, the result being a nonlinear drop-off in the curve.

Lastly, at this writing, there is no difference in the implementation of SMT's *ON mode and that of SystemControlled. The task dispatcher does not yet switch physical processors to single-threaded mode. The design of how to handle SystemControlled is not yet mature enough to minimize these effects to a level that is considered acceptable.

## An HMT Overview

HMT exploited the concept that modern processors are often quite fast relative to certain memory accesses. Without HMT, a modern CPU might spend a lot of time stalled on things like cache misses. In modern machines, the memory can be a considerable distance from the CPU, which translates to more cycles per fetch when a cache miss occurs. The CPU idles during such accesses. Since many applications feature database activity, cache misses often figured noticeably in the execution profile. So as with SMT, the question becomes "Can we keep the CPU busy with something else during these misses?"

HMT created two securely segregated streams of execution on one physical CPU, both controlled by hardware. It was created by replicating key registers including another instruction counter. Generally, there is a distinction between the one physical processor and its two logical processors. However, for HMT, the customer seldom sees any of this as the various performance facilities of the system continue to report on a physical CPU basis.

Unlike SMT, HMT allows only one instruction stream to execute at a time. But, if one instruction stream took a cache miss, the hardware switches to the other instruction stream (hence, "hardware multithreading" or, some say, "hardware multitasking"). There would, of course, be times when both were waiting on cache misses, or, conversely, applications that hardly ever had misses.

Generally, in most commercial workloads, HMT enabled gives gains in throughput between 10 and 25 percent, often without impact to response time.

In rare cases, HMT results in losses rather than gains. Although the system value QPRCMLTTSK also controls SMT, it was originally introduced in order to turn HMT on or off. This could only take affect when the whole system - not just a partition - was IPLed and controlled all partitions.

Not all prior models have HMT. In fact, some recent models have neither HMT nor SMT. The following models have HMT available: 270, 800, 810, 820, 830, 840. The following have neither SMT nor HMT: 825, 870, 890. Earlier models than the 270 or 820 series (e.g. 170, 7xx, etc.) did not have either HMT nor SMT.

## Conclusion

Simultaneous Multithreading delivered by IBM in POWER5 processors is an example of innovation that matters. Extensive microprocessor architecture research, development and testing has resulted in workloads running on POWER5-based IBM eServer iSeries platforms potentially experiencing a 35-40% performance capacity increase. Results gained depend on specific workloads, and on use of the default mode of "System Control," or changing to *OFF or *ON if benchmarking as testing so indicates.

## Disclaimer - System Performance

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Information is provided "AS IS" without warranty of any kind. Mention or reference to non-IBM products is for informational purposes only and does not constitute an endorsement of such products by IBM.

The material included in this presentation regarding third parties is based on information obtained from such parties. No effort has been made to independently verify the accuracy of the information. This presentation does not constitute an expressed or implied recommendation or endorsement by IBM of any third party product or service.

The information in this presentation is based on publicly available information about Microsoft and the Windows 2000 products, Compaq products, and Sun products. The information contained in this presentation is believed to be accurate by the author on the date it was published. However IBM does not warrant or guarantee the accuracy of the information.

The information presented regarding unannounced IBM products is a good faith effort to discuss iSeries plans and directions. IBM does not guarantee that these products will be announced.

## Trademarks


References in this document to IBM products or services do not imply that IBM intends to make them available in every country.
The following terms are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both:

| | | | |
|---|---|---|---|
| ADSTAR | Client Series | JustMail | PSF |
| Advanced Function Printing | DataGuide | Net.Commerce | SanFrancisco |
| AFP | DB2 Universal Database | Net.Data | SmoothStart |
| AIX | e-business logo | Netfinity | SystemView |
| AnyNet | IBM | NetView | WebSphere |
| Application Development | IBM Logo | OfficeVision | |
| APPN | IBM Network Station | OS/2 | |
| AS/400 | Information Warehouse | Operating System/400 | |
| AS/400e | Integrated Language Environment | OS/400 | |
| AT | Intelligent Printer Data Stream | PowerPC | |
| BrioQuery | IPDS | PowerPC AS | |

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information in this presentation concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources. Sources for non-IBM list prices and performance numbers are taken from publicly available information. Including vendor announcements, vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdraw without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.