Operating system exploitation of the POWER5 system

P. Mackerras
T. S. Mathews
R. C. Swanberg

The POWER5™ system incorporates several features designed to improve performance by eliminating bottlenecks and accelerating common functions used in operating systems. This paper discusses how two of the supported operating systems for POWER5—AIX® and Linux™—make use of these features to deliver improved system scalability and performance. In particular, the overheads for synchronizing translation-lookaside buffer (TLB) invalidations between processors, and for ensuring that the instruction cache is kept coherent by software, have been removed. The POWER5 simultaneous multithreading (SMT) implementation has features which allow operating systems to optimize the system for the kinds of applications being executed. We discuss how the operating systems approach the problems of scheduling tasks across the system, of determining when to switch processors between single-threaded (ST) and SMT mode, and of accounting accurately for CPU usage when in the SMT mode.

Introduction

The POWER5* microprocessor [1] introduces many innovations and improvements, in comparison with earlier [2] PowerPC Architecture* [3] systems, which improve the performance and scalability of operating systems and applications. These include features such as larger caches with lower latency, an on-chip memory controller with lower latency to main memory, and faster I/O buses.

These improvements require no changes in the software running on the system in order to exploit them. However, POWER5 also introduces a range of improvements which do require changes to software in order to be exploited. These include simultaneous multithreading, hardware instruction cache coherence,[1] hardware synchronization of translation-lookaside buffer[2] (TLB) invalidation operations, and the hardware barrier synchronization register. These new features, and the changes made to AIX* and Linux** to exploit them, are explained in detail below.

[1]The term *cache coherence* refers to the protocols and mechanisms used in a multiprocessor system to ensure that processors do not see inconsistent contents in main memory as a result of keeping local copies of data.

[2]A translation-lookaside buffer is a local cache which stores recently used address translations.

Simultaneous multithreading

POWER5 introduces simultaneous multithreading (SMT) to the POWER* line of PowerPC* implementations. With SMT, each microprocessor core can fetch from two instruction streams concurrently, and from the point of view of software has two complete sets of registers. Thus, each core appears to software as two complete logical CPUs (subject to some minor constraints that are visible only to the kernel), which we call *threads*. The POWER5 SMT implementation also allows software to render one thread "dormant." This "single-threaded" (ST) mode gives all of the chip resources to the remaining thread.

Supporting SMT efficiently introduces operating system complexities in the areas of configuration, scaling, scheduling, hardware thread priority management, and measurement of CPU utilization.[3]

Configuration

For both AIX and Linux, SMT mode is enabled or disabled at the system image level and is enabled by default. Using the *smtctl* command, AIX allows system administrators to switch dynamically between SMT

[3]Since 1998, i5/OS has had support for multithreading [4]; it supports SMT without modification.

0018-8646/05/$5.00 © 2005 IBM

and ST modes, or to switch modes at the next reboot. Dynamically switching modes causes logical CPUs to move offline or go online for the running AIX image, dependent on the mode selected, and invokes the Dynamic Reconfiguration Application Framework (DRAF) to enable applications to be notified of and react to the configuration changes.

Linux provides a global SMT enable/disable facility via a kernel command-line option that can be specified at boot time. Although there is no single global control to enable or disable SMT while the system is running, individual threads can be brought online or taken offline at runtime. If one of the two threads in a core is taken offline, that core is switched to single-thread mode by the hypervisor.[4]

Logical CPU scaling

The first effect of SMT is that, because it doubles the number of logical CPUs, operating systems must be capable of supporting more logical CPUs, depending on maximum hardware configurations.

AIX currently supports up to 128 logical CPUs. For Linux, the maximum number of CPUs supported by the Linux kernel is specified at compile time. This limit is set to 128 in SUSE SLES 9,[5] to 32 in the Red Hat RHEL3 QU3[6] release, and to 64 in the Red Hat RHEL4 release.

Affinity scheduling

SMT introduces complexities into the kernel scheduling decisions that are somewhat akin to the complexities of scheduling tasks on a machine with Non-Uniform Memory Access (NUMA) characteristics. Nonuniformity comes into play with SMT because threads on the same core share the core Level 1 (L1) cache and TLBs. As on NUMA machines, decisions on where best to schedule tasks may be driven by the presence of data relevant to the tasks in various levels of the memory hierarchy and the nature of memory sharing among the tasks being scheduled.

Both the AIX and Linux kernel schedulers provide affinity scheduling to optimize for nonuniformity. This takes two forms that are relevant to SMT. The first is affinity domain scheduling. Affinity domains consist of collections of nearby hardware resources (i.e., processor, caches), as defined by the memory hierarchy. They are hierarchical in nature, and the lowest-level affinity domains represent hardware resources that are closest together and comprise a single logical processor. Preference is given in hierarchical order, lowest to highest, to scheduling a task in the same affinity domain as when it last ran in order to leverage the performance of task-related data that may exist in various levels of hardware caches or buffers. With SMT enabled, the lowest-level affinity domains consist of a single thread. The domains at the next level above this consist of the threads of a single core and allow affinity domain scheduling to exploit the sharing of the L1 cache and TLBs between the two threads of a core.

Process-to-thread affinity is the other form of scheduling affinity provided by AIX that is of key importance for SMT. Process threads of the same process share a common address space and usually share data within this address space. Process-to-thread affinity attempts to schedule the process threads of a process such that shared hardware resources, such as caches, are leveraged for performance benefit in the sharing of process data. For SMT, process-to-thread affinity exploits the sharing of the L1 cache and TLB between the two threads.

Whether a partition is running in SMT or ST mode, affinity scheduling is performed only for dedicated processor partitions, not for shared processor partitions. In a shared processor partition, the processors seen by the operating system are virtualized by the hypervisor and mapped to physical processors in a fairly dynamic fashion that does not allow affinity scheduling. In fact, the hardware topology information required by AIX and Linux in order to determine the affinity domains of a partition is provided by the POWER5 firmware only for dedicated processor partitions. In contrast, a dedicated processor partition has a fixed mapping of virtual processors to physical processors, allowing the operating system to make meaningful affinity scheduling decisions.

Thread-sensitive scheduling

Under SMT, the two threads of a core interact with each other far more than two separate CPUs do. At the most basic level, one thread executing instructions slows down the other thread owing to contention for resources such as execution units and register rename slots. More subtly, because some resources are partitioned rather than shared, a thread executes slightly slower in SMT mode than it would in ST mode, even if the other thread is running in a simple kernel idle loop.

With SMT enabled, AIX and Linux exploit the fact that tasks run faster in ST mode and use a combination of thread-sensitive scheduling and hypervisor interactions to run individual CPUs in ST mode under light task load conditions. Under these conditions, the AIX and Linux schedulers attempt to schedule tasks so as to use only one thread in each core. A thread which has no scheduled tasks executes an idle loop within the kernel, where it loops waiting for work. If no work materializes after some period of time, the idle thread executes an H_CEDE

[4]The hypervisor is the layer of software that controls the access of the operating system to processor, memory, and I/O resources. It provides the ability to run multiple operating system instances on one machine.
[5]Novell; see *www.novell.com*.
[6]See *www.redhat.com*.

hypervisor call to indicate to the hypervisor that it has no work to do. If the other thread of the core is active, the hypervisor places the calling thread in the dormant state, placing the processor in ST mode and allowing the other thread to run as fast as possible.

The idle loop delays making the H_CEDE hypervisor call in the hope that work will soon become available and to avoid the performance overhead of unnecessarily placing the thread in a dormant state. A dormant thread is made active when it is presented with an interrupt or when it is the target of an H_PROD hypervisor call executed by another thread. The delay is tunable at boot time for both AIX and Linux, and also at runtime under AIX.

As the load increases on a lightly loaded system, the AIX and Linux schedulers begin to distribute work across all of the logical CPUs, thus scheduling tasks to secondary threads. Internally, the AIX and Linux schedulers maintain a queue of runnable threads for each logical CPU, since this is more scalable than having a single global queue and simplifies the implementation of affinity scheduling. Distributing work across all of the logical CPUs thus involves rebalancing these run queues; as a result of this rebalancing, the secondary threads of the CPUs are made active as work becomes available for them to do.

Hardware thread priority management

SMT supports hardware thread priorities which can be used by software to prioritize performance between the two threads of a core. The difference in priority between related threads dictates the ratio of processor decode slots allotted to each of the threads. If one thread is prioritized over the other, it is likely to run faster because it is allocated a larger proportion of the instruction decode slot resources. The lowest priority that can be set by the operating system also serves as a power-saving mode. A default *normal* priority is set for each thread by the hardware at system reset and on any interrupt or exception. AIX and Linux maintain the normal priority setting for most execution contexts.

User programs can set their own thread priorities within a restricted range, for which the normal priority is the highest level. However, the thread priority is reset whenever an interrupt occurs, and the kernel does not save and restore the thread priority. The use of priorities other than the normal priority is intended for short stretches of code and for tight loops, for example when waiting for a spinlock to become available. In these circumstances the resetting of the priority to the normal level on interrupts is acceptable.

AIX and Linux exploit SMT hardware thread priorities in an effort to provide better overall system performance. Thread priorities are lowered in places where a thread is doing work that could be characterized as nonproductive. This provides the sibling thread with the opportunity to use more execution resources and obtain better performance. AIX also raises thread priorities in order to complete more quickly work whose duration can affect system performance and throughput, e.g., when a high-priority spinlock is held.

AIX and Linux both use spinlocks for serializing access to various data structures. In its simplest form, a spinlock consists simply of a word of memory which is non-zero when some CPU is holding the lock, or zero when the lock is not held by any CPU (i.e., it is unlocked). To acquire the lock, a CPU waits until it is zero and then performs an atomic test-and-set operation that sets the word to a non-zero value if it is still zero at that time, as a single atomic operation. Such atomic operations can be implemented using the PowerPC `lwarx` and `stwcx.` instructions.

If a thread is waiting for a spinlock to become free, it is continually reading the lock variable, thus using up execution resources and slowing down the other thread. For this reason, AIX and Linux both lower the priority of the thread waiting for a spinlock to become free, and set the priority back to normal priority when the lock is successfully acquired.

AIX and Linux provide a similar optimization for the idle loop, which is executed by a thread when it has no scheduled tasks. On AIX, the idle loop spins in a loop continually examining the head of the thread run queue data structure for an indication that tasks have been scheduled to the thread. On Linux, the idle loop spins in a loop examining a single-bit flag indicating that a task has become runnable on that logical CPU. The priority of the thread is lower while it is spinning in this loop; it is restored to normal priority when the loop is exited. In contrast to the case of spinlocks, the thread priority is reduced to the lowest setting. Use of the lowest priority here provides the benefit of making the largest possible number of decode slots available to the sibling thread and of placing the idle thread in power-saving mode. It is likely that a relatively significant amount of time may pass before tasks are scheduled to the thread, so slowing down the discovery of the presence of scheduled tasks as a result of running at the lowest priority is not of concern. In comparison, adding latency to the discovery of free spinlocks through the use of the lowest priority would have a negative impact on system performance.

AIX provides spinlock primitives that raise the hardware thread priority of the thread holding the lock. More specifically, the thread priority is raised by the lock primitives at the time the lock is acquired and restored to normal priority by the unlock primitives at the time the lock is released. These priority-boosting lock primitives are provided for kernel locks that are in heavy contention

in order to minimize lock hold times and reduce contention of these locks. The set of locks which use this technique is set statically in the source code.

Neither AIX nor Linux currently sets the hardware thread priority differently for tasks with different software-scheduling priorities; all tasks are scheduled at the normal hardware thread priority level.

Measurement of CPU utilization

Being able to accurately measure the utilization of resources by a particular task or group of tasks is important in environments in which workloads are managed. For example, it is often important to ensure that sufficient resources are allocated to important processes so that they complete within a certain time or have the required level of performance. Accurate measurement is also important when users are charged according to the amount of computing resources used.

Both SMT and shared processors introduce complexities into the process of measuring CPU utilization. Historically, the CPU utilization of a process was simply measured as the elapsed wall clock time while the process was running, and it was sometimes approximated by counting the number of timer interrupts that occurred while the process was running. SMT complicates this, because the number of instructions executed by a thread during a given interval of time vary depending on the activity of the other thread. In other words, with SMT, a process cannot be considered to have a "whole" CPU while it is running if the other thread is active.

Shared processors introduce complexity because the virtual processor is not always executing on a real processor. Because the kernel does not know when or for how long the virtual processor is dispatched on a real processor, the elapsed time while the process is scheduled on the virtual processor is not an accurate measure of the amount of CPU resource it has received.

To help solve these problems, the POWER5 processor includes a per-thread processor utilization of resources register (PURR), which increments at the timebase frequency multiplied by the fraction of cycles on which the thread can dispatch instructions. In other words, the PURR counts at the same rate as the timebase in single-threaded mode; in SMT mode, if each thread has equal priority, the PURR counts at half the rate of the timebase register. If one thread has a higher priority than the other, its PURR will count faster than the other thread's, but the sum of the PURR values will always increase at the same rate as the timebase register. (The PURR does not count CPU cycles, because the timebase counts at a lower frequency than the CPU clock.)

The hypervisor virtualizes the PURR so that each virtual processor sees a PURR value that increments only while the virtual processor is dispatched on a real processor. Thus, the change in the PURR value while a process is running reflects the number of CPU cycles that were used by the process reasonably accurately. (With SMT, the notion of "CPU cycles" becomes somewhat fuzzy, though, because the proportion of the chip's total execution resources used by a thread may not be exactly equal to the proportion of cycles on which the thread could dispatch instructions.)

AIX contains many facilities for which the accurate measurement of CPU utilization is crucial, such as the workload manager (WLM), software priority management, charge-back accounting, and statistics provided through performance tools. In the past, these facilities have used elapsed time or timer-interrupt-based sampling as the basis for measurement, but they have now all been converted to use the PURR.

In fact, the AIX infrastructure for measuring CPU utilization has been re-engineered for POWER5. The AIX kernel now reads the PURR on each context switch, on each interrupt/exception entry and exit, and on each entry to and exit from user mode. This provides accurate measurement of the CPU time used by each task in user mode and in the kernel, and the CPU time used in handling interrupts. The per-task information is accumulated on a per-process and per-logical-CPU basis. Accumulation of per-process information occurs once per second in order to avoid the scalability problems that would be encountered if the per-process information were updated every time the PURR is read. Because the per-process information could potentially be accessed by many CPUs in the system under a multithread application, it is important to limit the rate at which the per-process information is updated. Otherwise, the lock protecting access to the per-process information could easily become subject to major contention.

The AIX facilities that depend on the measurement of CPU utilization use this new infrastructure. For example, AIX charge-back accounting facilities, including the fairly standard Unix Accounting Utilities and the newly introduced Advanced Accounting feature of AIX, use the infrastructure to obtain per-process CPU utilization for the purpose of accounting. Similarly, WLM uses process-level information provided by the infrastructure in providing share-based workload management and managing CPU, memory, and I/O bandwidth usage for classes of processes based upon system administrator-defined policies.

In addition to providing value in supporting SMT, the new AIX CPU utilization measurement infrastructure provides more accurate CPU utilization information than was available on AIX in the past. In many cases, timer-interrupt-based sampling was used as the basis for measurement. This method of measurement yields

information that reflects the CPU utilization but is not an accurate measurement of it. Additionally, the new infrastructure separately measures and accounts for CPU utilization by interrupt handlers. This was not the case in the past, and interrupt handler CPU utilization was assigned to the interrupted process instead of being treated as system overhead.

The Linux CPU utilization measurement infrastructure is currently still using the old method of counting timer interrupts to determine process CPU usage. The Linux kernel reads the PURR at regular intervals and makes the values available through a file in the `procfs` virtual file system.[7] This can be used to estimate how much of a real processor each virtual processor has been allocated.

Hardware page copier

The POWER5 system has a facility that is designed for efficient copying of memory. This hardware-based facility efficiently copies 4,096-byte pages from source locations to destination locations in real memory. Because the hardware page-copy facility uses real addresses, only the hypervisor can use it directly. Operating systems can use the facility through a hypervisor call, which requires the operating system to specify logical real addresses for the source and destination pages.

AIX uses the page-copy hypervisor call in several situations in which a whole page of memory has to be copied—for example, when a process forks.[8] (In fact, as an optimization, the copy for each page of the process memory is deferred until either the parent or the child first writes to that page.)

Because AIX must specify real addresses rather than virtual addresses for the source and destination pages, it uses the page-copy hypervisor call only in situations in which the real addresses for the pages to be copied are guaranteed to remain constant over the page-copy operation—for example, where the source and destination pages are fixed in memory. Use of the call is further limited to cases in which the real addresses of the source and destination pages are readily known and available and the real addresses do not have to be determined through costly lookups based upon the virtual addresses. The requirement for fixed pages with known real addresses means that the page-copy hypervisor call is used primarily by the AIX virtual memory manager.[9]

Hardware instruction cache coherence

In the PowerPC Architecture [3], implementations are permitted to have an instruction cache that is not automatically kept coherent with updates to memory. Instead, instructions are provided for enforcing coherence when necessary. In particular, the `dcbst` instruction is used to write back modified cache lines from the data cache to memory, and the `icbi` instruction is used to invalidate cache lines in the instruction cache. When new instructions have been placed in an area of memory, either by direct memory access from an I/O device or by store instructions executed on a CPU, software must perform a sequence of `dcbst` and `icbi` instructions for that area of memory before attempting to execute the new instructions.

This is relevant to operating systems such as AIX and Linux that demand-page executables. When a program goes to execute an instruction from a page that is not present, the operating system maps the page into the process address space. If the operating system does not currently have the page in memory, it must first read it in from the executable file. Before making the page accessible to the process, the kernel must ensure that the instruction cache does not contain stale contents for that page. (The kernel cannot expect the user process to do this, because the user process has no way of knowing that the page has just been read in.)

A similar situation exists when a page contains both instructions and data and is subject to copy-on-write. When a process executes the fork system call under Linux or AIX, the writable pages of the process image are not immediately copied, but instead are made read-only and shared between the parent and the child. If either the parent or the child attempts to write to the page, the store causes a page fault and the kernel then copies the page and maps the copy into the process address space. If the page is executable, which is the default under Linux, the kernel must once again make sure that the instruction cache does not contain stale contents for the new page.

However, as described elsewhere in this issue, the POWER5 microprocessor maintains coherence of the instruction cache in hardware. The instruction cache hardware observes all stores to memory, either by a processor or an I/O device, and automatically invalidates any matching lines of the instruction cache. Therefore, the kernel does not have to do anything to ensure that the instruction cache does not contain stale contents. This saves time and reduces complexity in the kernel. Linux exploits this by branching around the instruction cache coherence code on POWER5 machines. AIX also exploits this, but in a more indirect fashion. When AIX requests that the hypervisor create a mapping for a page, it can also request that the hypervisor make the page coherent with the instruction cache. The POWER5 hypervisor is able to ignore this request, since the coherence is maintained by hardware.

[7] The `procfs` file system exports information about processes and other kernel internal structures in the form of virtual files that can be accessed using the normal open, read, and write system calls.
[8] The fork system call duplicates the current process, creating a new child process which has a memory image identical to that of the parent (but which is not shared with the parent).
[9] Linux does not use the page copier.

A similar issue arises in programs that generate instructions at runtime, such as a Java** JIT (just-in-time) compiler. On other PowerPC implementations, the `dcbst` and `icbi` instructions must be used before newly generated instructions are executed, but on POWER5 they can be omitted.

Hardware synchronization of TLB entry invalidation

In previous PowerPC implementations such as POWER3* and POWER4*, only one processor in a multiprocessor system was permitted to perform a TLB entry invalidation at any one time. (TLB entry invalidations are performed using the `tlbie` instruction and are broadcast to all processors in a multiprocessor system so that they act on all of the TLBs in the system.) The constraint that only one TLB entry invalidation can occur at any one time requires software to use a global lock variable to serialize the execution of the `tlbie` instruction between processors.

POWER4 introduced a local form of the `tlbie` instruction, called `tlbiel`, which acts only on the local TLB and does not require synchronization. Linux uses the `tlbiel` instruction on POWER4 when it is not running on a hypervisor and when it can determine that the mapping being invalidated has been used only on the local processor. When Linux is running under a hypervisor, only the hypervisor executes `tlbie` instructions, because the hypervisor has control of the virtual-to-real address mappings. On POWER5 systems, the hypervisor need not use a global lock to serialize `tlbie` instructions across the machine. This removes one potential bottleneck from the system and simplifies the hypervisor software. (Note that running under a hypervisor is the only mode of operation supported on POWER5 systems.)

Barrier synchronization registers

POWER5 introduces a set of barrier synchronization registers (BSRs) that provide fast, lightweight barrier synchronization between CPUs. This facility is intended for use by application programs that are structured in a SIMD (single-instruction multiple-data) fashion—that is, where the CPUs are executing similar or identical instruction streams but processing separate sets of data. Such programs often proceed in phases, with processing phases alternating with data communication phases. Generally the program is required to wait at the end of each phase until all CPUs have completed that phase. The BSR is designed to accomplish this efficiently.

Support for the BSR facility is implemented in the latest releases of AIX 5.2 and 5.3 at the time of writing, and is planned for the near future for Linux. AIX and Linux do not plan to use the BSR facility internally but simply to expose it to applications via an interface to be determined.

Conclusion

The POWER5 system introduces several new features that together improve performance and scalability in comparison with earlier POWER systems. AIX and Linux have been modified where necessary to support those features and make them available to applications.

Acknowledgments

David Engebretsen and his team designed and implemented much of the architecture-specific Linux kernel code for supporting shared processors and SMT, and in particular the code to cede the virtual processor to the hypervisor when idle, and the initial version of the shared-processor spinlocks.

References

1. R. Kalla, B. Sinharoy, and J. M. Tendler, "IBM POWER5 Chip: A Dual-Core Multithreaded Processor," *IEEE Micro* **24**, 40–47 (March–April 2004).
2. J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," *IBM J. Res. & Dev.* **46**, No. 1, 5–26 (January 2002).
3. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, C. May, E. Silha, R. Simpson, and H. Warren, Eds., Morgan Kaufmann Publishers, Inc., San Francisco, 1994.
4. J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel, "A Multithreaded PowerPC Processor for Commercial Servers," *IBM J. Res. & Dev.* **44**, No. 6, 885–898 (November 2000).

Received August 19, 2004; accepted for publication January 4, 2005; Internet publication August 10, 2005

Paul Mackerras *IBM Linux Technology Center OzLabs, 8 Brisbane Avenue CA03, Canberra ACT 2611 Australia (paulus@au.ibm.com)*. Dr. Mackerras is a Senior Technical Staff Member in the IBM Linux Technology Center. He received a B.Sc. degree in mathematics and computer science and a B.E. degree in electrical engineering with first-class honors in 1982 from the University of Queensland, Australia, and a Ph.D. degree in computer science from the Australian National University in 1988. He joined IBM in 2001 at the Linux Technology Center "OzLabs" facility in Canberra, Australia. Dr. Mackerras has overall responsibility for the parts of the Linux kernel that relate to running on PowerPC-architecture machines, and is recognized in this role by both IBM and the Linux kernel development community.

Thomas S. Mathews *IBM Systems and Technology Group, 11501 Burnet Road, Austin, Texas 78740 (tmathews@us.ibm.com)*. Mr. Mathews is a Distinguished Engineer in the AIX Product Development organization within the IBM Systems and Technology Group. He received a B.Sc. degree in computer science from the University of Texas at El Paso in 1984. He joined IBM in 1985 and has been a member of the AIX development team in Austin, Texas, for seventeen years, developing AIX product capabilities in the areas of file systems, memory management, system scalability, virtualization, and hardware exploitation. Mr. Mathews is currently has overall architectural responsibility for the AIX kernel, libraries, and commands and utilities.

Randal C. Swanberg *IBM Systems and Technology Group, 11501 Burnet Road, Austin Texas 78758 (rswanber@us.ibm.com)*. Mr. Swanberg is a Senior Technical Staff Member with the IBM Systems and Technology Group in Austin, Texas. After beginning his career working on defense navigation systems for the U.S. Army at Fort Hood, Texas, he joined IBM in 1989 and has worked on several operating system projects including AIX, OSF, Monterey, and Linux. Mr. Swanberg received a bachelor's degree in computer science from Baylor University.