Bret R. Olszewski

Mysore Srinivas

Greg Mewhinney

Introduction

This paper describes hardware multi–threading as implemented on AIX 4.3.3 and RS/6000 hardware. The paper also discusses the performance and operational trade–off's made when using hardware multi–threading.

Hardware Multi–threading Concepts

Hardware multi–threading (HMT) is a concept wherein two or more logical processors share physical hardware resources. While a number of hardware architectures have in the past used similar structures for a variety of reasons, academic research in recent years has focused on HMT as a means of improving system throughput by compensating for increasing large memory latencies. The RS64–II and RS64–III microprocessors are HMT–capable [STO98].

While microprocessors have continued to increase processing performance more or less in keeping with Moore's Law, the underlying components of the memory system (RAM,DDR, RDRAM) have not kept pace. Rather, these components have tended to emphasize density and manufacturability over absolute performance. This trade–off has dramatically increased the amount of memory and reduced overall system cost, but required system developers to invent increasingly complex system structures to maximize overall hardware performance.

As an example, consider a hypothetical four–processor system running at 500Mhz. Such a system could reasonably expect it's memory subsystem to be clocked at one–fourth of the processor speed, or about 133Mhz. On a four–processor systems such as our model, the memory bus is shared for control and data by all of the processors. This "shared bus" implementation is typically cost efficient and high performance for low–end SMP systems. On such a system, one could normally expect a memory read operation to take on the order of 60 processor clocks (or cycles). This figure is hypothetical, exact timings are very implementation dependent. Larger systems use more complex memory structures, such as cross–bar switches, because the shared bus design cannot supply enough memory bandwidth to scale to a large number of processors.

Approximately forty percent of instructions on a RISC–based processor are loads or stores. Obviously, if a processor had to wait 60 cycles for each one, the processor would spend far more time waiting on the memory subsystem than actually executing instructions. So, processors take advantage of the fact that usually there is some locality to the access of data by using caches. A cache is a copy of memory which is quicker to access than memory. All modern processors contain at least one on–chip cache (usually referred to as L0, L1, and sometimes L2) which provides nearly instantaneous access. The size of the on–chip cache is limited by real estate (the maximum size of the chip) or manufacturing yield (the bigger the chip, the more of them that are not useful due to manufacturing defects). While the amount of on–chip cache has tended to increase with higher chip densities, most systems also include a much larger off–chip cache (usually referred to as L2 or L3).

The net effect of caches is to reduce the number of requests to memory. However, caches do not completely remove the need for accessing memory, just reduce the frequency. If caches remove the need for 99% of accesses to memory, the processor still ends up spending about 0.6 cycles of each average instruction waiting on memory. Consider that a processor such as the RS64B–III can execute up to four instructions per cycle! The memory delays account for far more execution time that the time per instruction in the processor. In practice, delays in cache misses, serialization, translation misses, and other effects tend to result in average instruction latencies in the range of two to four cycles per instruction on multi–processor systems.

Hardware multi–threading provides a mechanism of improving overall system throughput by overlapping memory accesses with other

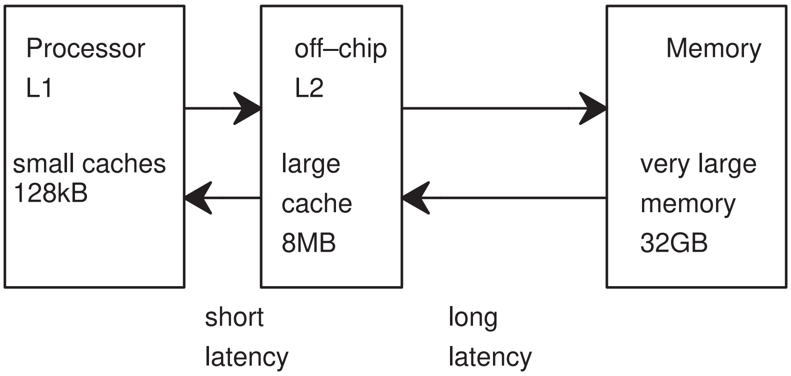

Figure 1 – The memory hierarchy

computation. The implementation in the RS64–III family of processors has two contexts (logical processors) per physical processor. At any processor clock, one and only one of the contexts is active and the other is inactive. The physical processor switches between the contexts, typically at a fairly rapid rate, to give the illusion of two processors to software.

The mechanisms to switch between contexts are are controlled by the operating system software. Three basic classes of switch events exist:

- Hardware events – including cache misses and virtual memory translation misses
- Time out – prevents one logical processor from starving another
- Software hints – allow software to favor the priority of one logical processor over another

The operating system sets the switch events at system initialization. The hardware events are the primary triggers of switching between logical processors. The most frequent switch comes on a cache miss event, as shown in Figure 2. The figure initially shows logical processor one executing. It tries to execute a load instruction (a memory reference). The instruction causes a cache miss, which causes the microprocessor to switch contexts from logical processor one to logical processor two. The switch takes three or four processor cycles. Following the switch, logical processor two begins executing instructions. Typically, when the data requested by logical processor one is returned, the microprocessor will switch context back to allow that processor to continue processing. Note that it is also possible to have both logical processors waiting on cache misses. In this case, HMT doesn't provide any additional parallelism.

Some of the processor resources, such as registers, have a copy for each logical processor. But most resources on the physical processor, such as cache, are shared between the logical processors. From the perspective of cost of the hardware this is a positive. The size of the microprocessor increases only a little by adding HMT. From the perspective of performance, the sharing of resources can improve or degrade performance depending on the characteristics of the workload.

AIX Support

AIX 4.3.3 supports HMT on the RS/6000 model M80 with APAR IY10778. It is enabled by the command "bosdebug –H on" and activated on subsequent reboot. When invoked, AIX presents the system as having twice as many processors (observable by bindprocessor –q or other commands) as are physically installed. Remember that each processor reported by AIX is actually sharing approximately half of the resource of a physical processor. As such, the system is composed of twice as many processors, but each has typically a little more than one half the performance of a non–HMT system.

The modifications to AIX to support HMT are fairly modest. The first is to enable the second logical

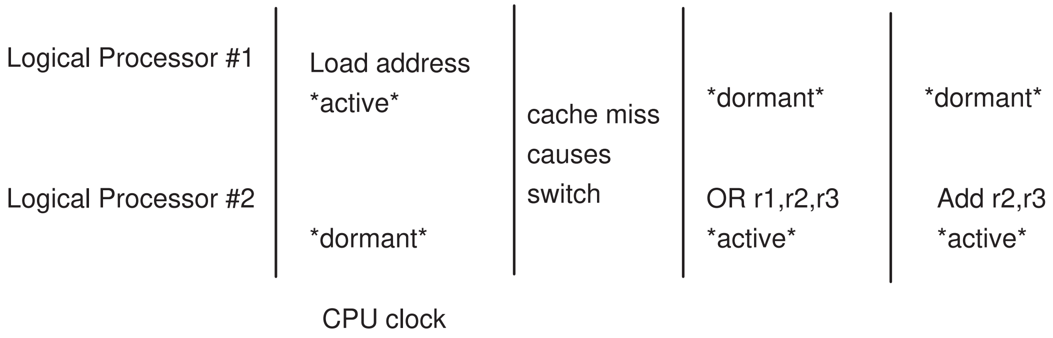

Figure 2 – Logical processor switch

processor. This is done fairly late in system initialization. Second, software was modified to use software hints for low priority operations such as the idle loop and lock spinning. The idle loop modification essentially allows an idle thread to consume virtually no processor cycles, allowing the other logical processor the full resources of the processor. The modifications for lock spinning have the same effect, a logical processor which is waiting on a lock yields the physical processor to the other logical processor. This tends to improve overall throughput even when software lock contention occurs.

On the currently supported HMT platforms, external I/O interrupts can only be routed to one logical processor of each physical processor. This asymmetric characteristic does not typically reduce throughput, as thread priority software hints are used by AIX to ensure that sufficient resources are available to service interrupts.

The RPDP (CPU guard) feature of AIX is not supported when HMT is enabled. The CPU guard feature predicts the failure of a CPU and disables it without shutting down the operating system. When HMT is enabled, an error log entry is generated, but the logical CPUs associated with the CPU guard event are not disabled. Support for CPU guard running under HMT may be added in a future AIX release.

Performance Benefits

Quite a number of workloads have shown a performance advantage using HMT. The performance advantage of HMT tends to be greatest on systems with a modest number of physical processors. In general, results are good on 8–way systems and below. On systems with a larger number of CPU's, the software scalability of the application usually becomes the limiting factor.

One workload extensively analyzed is SDET. The SPEC SDM SDET workload mimics a number of users doing software development sharing a system. The workload executes sets of AIX commands in shell scripts. For more information on this workload, or SPEC, consult their web page at http://www.spec.org. The metric of performance is scripts per hour, or the number of scripts that could be executed in one hour. Normally, the benchmark is run numerous times to reduce the effect of run to run variation in the measurements. The benchmark exercises the AIX operating system in very useful ways. In particular, the benchmark stresses the AIX file system, processor management, and the virtual memory manager.

A series of measurements were performed on an RS/6000 model M80. The system contained 8 500Mhz CPU's, IBM SSA disk drives, and 32 giga–bytes of memory. The SDET workload was run to peak performance using one, four, and eight physical processors. Chart 1 shows the relative speedup for each CPU configuration using HMT.

The SDET performance results show speedup from HMT for all cases. The improvement varies from 4.2% on one–way to 11.5% on four–way,

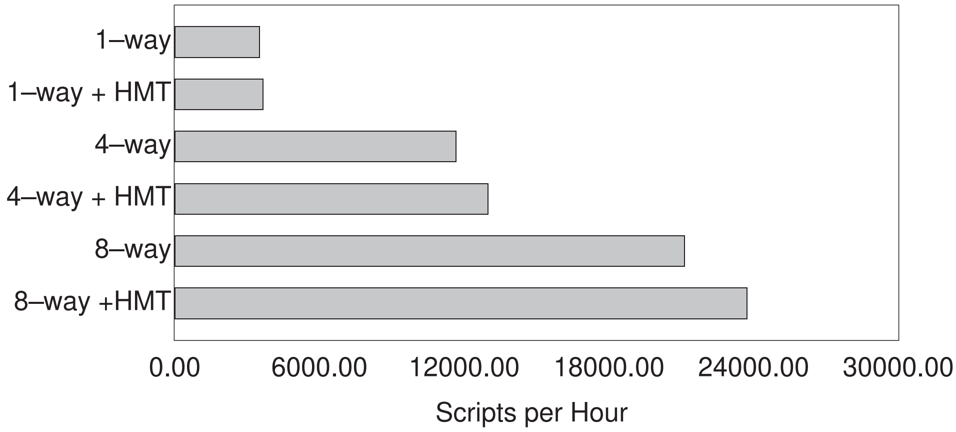

Chart 1 – SDET Throughput on M80

peaking at 12.3% at eight–way. Note that the improvement varies quite considerably in this workload. This is a function of the workload and is noted on other workloads as well.

We can look more deeply into the improvements by using a hardware performance monitor. The monitor counts hardware events, such as cycles, instructions executed, and the number of cache misses that occur while running the workload. What we would expect to find is that the number of instructions executed per unit time (instructions per cycle) should be greater with HMT than without it. That is because most of the time that one logical processor is waiting on a cache miss, the other logical processor can continue to execute instructions. Chart 2 shows the rate of change in instructions per cycle. The chart shows the improvement in SDET throughput is very closely tied to the improvement in instructions per cycle. One interesting effect is that the improvement in throughput at the eight–way point is actually a little bit better than the improvement in instructions per cycle. In this case, the actual number of instructions per script decreases running under HMT. This is counter intuitive, as HMT should experience path length growth due to locking because of the doubling in processors observed by software. The explanation is in the software hints used by AIX to “hide” lock spinning by putting spinning logical processors into low priority.

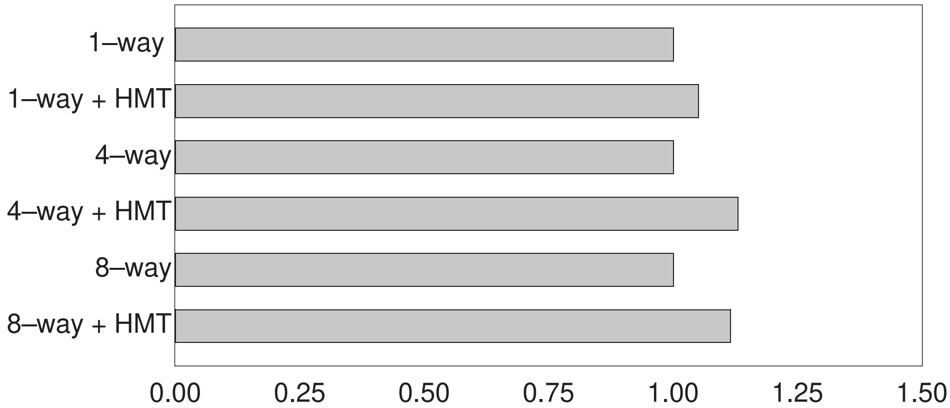

Chart 2 – SDET M80 improvements in instructions per cycle

Another workload carefully studied was an online transaction processing (OLTP) workload using a commercially available relational database. Workloads of this kind tend to mimic the behavior of a number of customer environments, such as enterprise resource planning (ERP). These workloads have characteristics which tend to make them perform very well with HMT. These characteristics include high multi–programming levels, relatively high cache miss rates, and good workload scalability.

Once again experiments were performed on an eight–way M80 system. This test showed an improvement in throughput of 22.0%. The improvement in instructions per cycle was 27.0%. The fact that the overall throughput improvement was less than the increase in instructions per cycle is a reflection of less than perfect scaling in the application. Improvements in similar applications on systems with far greater number of processors is typically less, on the order of 5% to 10% in the best cases.

Additional tests in web serving environments has shown gains similar to SDET and OLTP. These environments have similar characteristics which allow HMT to show a performance benefit.

Performance Cautions

The nature of HMT performance enhancement assumes a highly multi–programming workloads. When one task is waiting, another task runs. The hardware and AIX attempt to provide fairness between the threads. While this balance proves very effective for transactional environments, it is not optimal for situations where a single thread of execution needs to complete in the shortest period of time. Under no circumstances will a single thread of execution have a shorter elapsed run time under HMT than without it. Consider that whenever a task takes a cache miss, the hardware will switch to the other logical processor. This switch is not free, it takes 3 or 4 processor cycles. When the data is returned for the miss, the processor frequently needs to switch back to the dormant thread. Thus, even if the system is fully idle except for one thread, that thread will run longer under HMT than without it.

This behavior was been noted in a business intelligence (BI) workload which measured the run time of a single query. The query performance was typically slower by a factor of ten to twenty percent under HMT. The same series of tests, when measuring the throughput of a number of long running queries in parallel showed throughput improvements consistent with SDET, though with increased elapsed times for each query. Thus, the benefits of HMT for highly transactional, high multi–programming works must be weighed against its drawbacks for time critical single tasks. If a single task, such as a single query, backup, or database reorganization is the critical path in a business process, running without HMT may actually prove more efficient.

Another potential hurdle with HMT is related to the fact that the logical processors shared physical resources, such as caches. An example of this is the SPECint95 rate benchmark (www.spec.org). This benchmark, now retired, runs multiple copies of the same program in lock step. The benchmark gives a measure of throughput of a multi–processor system. However, when this benchmark is run with HMT the results are worse than without HMT. Table 1 shows the results of this benchmark obtained on an M80 system.

#CPU’s	#copies	HMT on	Result
1	1	No	186
1	2	Yes	171
4	4	No	630
4	8	Yes	434
8	8	No	1489
8	16	Yes	1259

Table 1 – M80 SPECrate int95 results

The workload data shows that less throughput was obtained with HMT at every measurement point. The decrease in throughput ranged from 8.8% to 45.2%. The variability of the timing for the benchmark runs was very high, the outer bound of decrease is likely a worst case. Hardware counter analysis showed that the workload suffered from a large increase in level one data cache misses. This is a result of running two identical programs, accessing similar data addresses, at the same time against the same cache. Similar behavior has been noted running programs such as SETI@home (www.setiathome.ssl.berkeley.edu) as a test of system throughput. While cache thrashing is fairly unlikely in complex environments, it would most probably appear on systems with low multi–programming levels and long running jobs.

Enabling HMT has the effect of doubling the number of processors that AIX presents to software. This increase can expose problems with workload scalability. Though many applications, such as those by the major database vendors, have been proven to scale to a large number of processors, there still exist many applications designed to run efficiently on systems with at most a few processors. Applications which do not scale are likely to perform worse under HMT, due to contention inherent to their serialization (e.g. locking). Scalability problems are most likely to be noticed on systems with large numbers of processors. If you intend to run an application which is not proven to scale, performance benchmarks should be undertaken with and without HMT to determine the throughput possible in both environments.

Another side–effect of two logical processors sharing a physical one is an increase in the variability in elapsed and CPU time to complete a task. Since AIX treats the two logical processors as priority equals, the amount of resource available to one logical processor can be affected by what is running on the other processor. If a thread is running on one logical processor and the other logical processor for that physical processor is idle, the non–busy thread will execute with nearly all of the CPU resources. However, if two non–busy threads share a physical processor, each will execute about one–half of the time. Thus, the elapsed time to run any task may vary on the order of a factor of two. This effect is shown in Chart 3. The chart shows the elapsed run time for a test case using a compression program. The first measure, "no HMT" is the elapsed time on a single processor system without HMT. The second measure, "HMT compress & idle", shows the elapsed time by enabling HMT. In this case, no other thread is running on the second logical processor. The measure "HMT compress & loop" shows the elapsed time when running the test case at the same time as a tightly spinning loop. The final case shows the effect of running two copies of the compress test at the same time. This effect could be summarized as each logical processor provides a variable amount of performance, based on the current load on the other coresident logical processor.

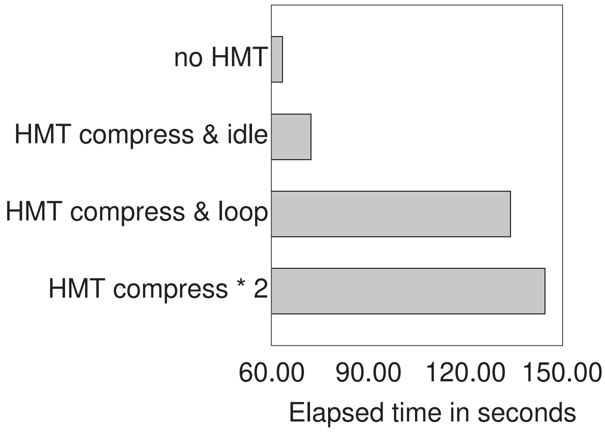

Chart 3 – Run time variability

Another issue with the current HMT implementation is CPU utilization. AIX records CPU utilization with a sampling technique. One hundred times per second each processor samples what task is running. This tends to result in a reasonable approximation for CPU time accounting. From that perspective, there isn't so much a problem with CPU utilization as there is with the capacity implied by CPU utilization. For example, if a system is running at fifty percent CPU utilization, one may assume that approximately twice as much work could run on the system before the CPU was totally consumed. This of course assumes that some other bottleneck, such as memory, I/O, or workload scalability didn't limit throughput. Now consider what happens with HMT. Imagine a thread running on an otherwise idle system with one physical processor and two logical processors. Each time the thread has a cache miss, the hardware switches to the other logical processor. If no other thread is running in the system, this shows as idle time. Now, if a second thread is also run on the system, does the throughput of the system increase linearly with the idle time that was previously observed? Not necessarily, as the two threads may both end up blocked on cache misses from time to time.

We can observe this effect using the SDET benchmark. By running different numbers of scripts, we can arrive at throughputs with varying CPU utilizations. We can then scale the throughput by the CPU utilization to estimate CPU limited performance at 100% utilization. Chart 4 shows the curve for SDET running with HMT. The chart shows throughput at measured utilization on the "measured" line. The extrapolated line results from scaling the measured line to assumed throughput possible at 100% CPU utilization. If measured utilization was linear with capacity, the extrapolated line should be linear at purely horizontal ending at the measured peak. Chart 4 clearly shows that the extrapolated line is not linear. Chart 5 shows the same measures for the workload with HMT disabled. While the extrapolated line is not horizontal for all points, it is much closer than that measured with HMT. The overall capacity error is the different between the extrapolated line at any utilization and the peak measured performance. In the HMT case, the extrapolated line reaches values above 37,000, where as the peak measured performance is only on the order of 24,500. With HMT disabled, the peak extrapolated performance is around 26,600 compared with a peak measured performance of about 21,700. Also, the non–HMT case shows fairly accurate extrapolated performance at utilizations above thirty percent. As a general rule, it is always difficult to extrapolate performance at low utilizations, because the sampling–based mechanisms used to measure utilization tend to be less accurate at low CPU utilizations.

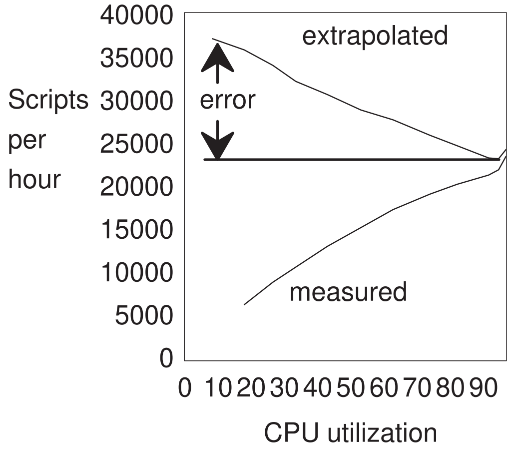

Chart 4 – HMT on M80 throughput/utilization

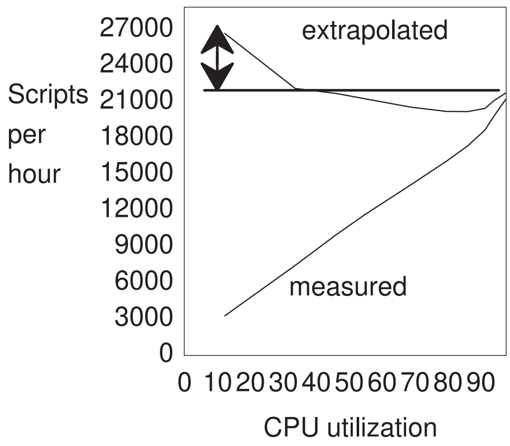

Chart 5 – HMT off M80 throughput/utilization

Thus, the CPU utilization observed with HMT is somewhat less precise as a measure of available capacity. If highly accurate CPU utilization is required, HMT might not be an option.

Conclusion

Performance gains on workloads with high multi–programming levels of ten to twenty percent are possible with HMT. The gains tend to be highest on systems with eight or less physical processors. These potential gains may or may not be possible in any given customer environment. Issues with performance of long running critical tasks, workload scalability, cache thrashing, and the requirement for certainty in CPU utilization should be considered when evaluating applicability of HMT in your environment.

References

[STO98] "Commercial Multi–threaded RISC Processor", with S. Storino, A. Aipperspach, J. Borkenhagen, R. Eickemeyer, S. Levenstein, and G. Uhlmann, 1998 International Solid–State Circuits Conference, Feb. 1998, San Francisco, CA.

Trademarks

AIX and RS/6000 are registered trademarks of International Business Machines Corporation.

SPEC, SPECrate int95 are trademarks of Standard Performance Evaluation Corporation