Internet Nuggets

Mark Thorson
eee@netcom.com
June, 1996

This column consists of selected traffic from the *comp.arch* newsgroup, a forum for discussion of computer architecture on Internet—an international computer network.

As always, the opinions expressed in this column are the personal views of the authors, and do not necessarily represent the institutions to which they are affiliated.

Text which sets the context of a message appears in italics; this is usually text the author has quoted from earlier messages. The code-like expressions below the authors' names are their addresses on Internet.

Architectural Potholes

Mark Rosenbaum
mjr@netcom.com

That is an interesting contrast. It seems that with the large caches and the high hit rates that most SMP systems have that even SMPs might be considered logically shared physically-distributed.

That's an interesting topic; I've never felt that I had the "killer explanation" for the difference ... but I think there's a fundamental difference, in practice. So, let me try, and hopefully somebody can offer a more succinct version.

[Note: references would be Lenoski and Weber's Scalable Shared-Memory Multiprocessing and Greg Pfister's In Search of Clusters.]

1) "SMP-classic":

a) CPUs with coherent caches.

b) UMA shared memory, i.e., with same access time from every CPU, and memory access time much longer than cache access time.

2) "CC-NUMA-classic":

a) CPUs with coherent caches.

b) NUMA shared memory, with a more complex access time function:

Local memory access time much longer than cache access time.

Remote memory access time much longer than local memory access time.

and possibly:

Remote memory access time may or may not vary much according to where it is.

At some point, if the remote memory access time is much longer than local memory access time, people will use the system as 3) message-passing system, even if the memory is logically shared, simply because the penalty for acting like everything has the same access time gets too high.

Excellent point, any guess as to what the ratio might be? 10X? 100X? Also, doesn't the communications also play a part here. Even if the remote memory is very close (say 2X away) but congest easily wouldn't that force you to treat the machine as distributed memory?

Now, what's different: layout of data for caches tends to be a short-term, small-size, dynamic issue; layout of data for multiple memory systems tends to be a long-term, larger-granularity issue ... and apparently much more difficult in the general case, as evidenced by the relative numbers of 3rd-party parallelized applications. (I.e., many for SMP, not many for the extreme case of message-passing systems. There's plenty of data for the extreme points, not so much for CC-NUMAs, although if you look at NCSA's software application pages, there may be some indication.)

While it is true that the number of MPP (distributed memory) applications is small, I'm

not sure the explanation is that MPP is too hard to program. My experience in the industry was that the SMP systems were used for multi-user timesharing systems and the MPP systems were designed for a small number of large applications. This is highlighted by the fact that on many of the MPP systems (nCUBE, Intel, TMC, ...) users could not directly logon to the nodes but rather only the host. This would require any application ported to these systems to be parallelized while a simple (non-parallel) port to SMPs could be done *without* parallelizing the code. The non-parallel versions for things like word processing worked fine.

This created a software base which drove sales which created a business reason to then go back and parallelize the code. As far as I can tell the jury is still out on how well that effort is going. A look at the RDBMSs yield mixed results. There are still some RDBMSs that don't scale well even on SMPs. On the other hand, there was no technical reason why MPP could not have been used as timesharing machines. Netcom (my service provider) uses a Network Of Workstations (NOW) to provide UN*X timesharing. Also, I believe that the IBM SPX can also do this. Given the way things went one would have to question the business model that these companies were using.

A point of interest about distributed memory systems is that they can be designed to be fault tolerant (currently, they typically are not).

For data base applications where the disk seek time will cover most latencies, this architecture offers some significant advantages. It will be very interesting to see how the market changes.

Architectural Potholes

John Mashey
mash@sgi.com

Excellent point, any guess as to what the ratio might be? 10X? 100X? Also, doesn't the communications also play a part here. Even if the remote memory is very close (say 2X away) but congest easily wouldn't that force you to treat the machine as distributed memory?

Seems likely.

While it is true that the number of MPP (distributed memory) applications is small, I'm not sure the explanation is that MPP is too hard to program. My experience in the industry was that the SMP systems were used for multi-user timesharing systems and the MPP systems were designed for a small number of large applications.

This may be a matter of viewpoint, i.e.:

a) As Mark notes, many SMPs were straightforwardly used for throughput of multiple single-thread programs that were probably derived from uniprocessors anyway, and as noted, with varying degrees of success for commercial DBMS.

b) On the other hand, in the technical world, people were certainly routinely doing small-N-way parallelization of codes in the late 1980s and early 1990s. While I'm not sure how much of this was going on on other vendors' systems, I do know it was happening on SGI PowerSeries systems. I was still at MIPS, but over the years since I've talked with lots of scientists and engineers who had done this, certainly not to do algorithms research, but just to solve their problems faster. I especially recall one who had an 8-CPU system: he was running a research job that took weeks, and that every once in a while looked at a file to see how many CPUs it should be using. When he went home, he'd let it have all 8. When he came back, he'd tell it to keep 4, so he could have 4 for his interactive work.

It is also the case that Power Challenges acquired a reasonable number of parallelized ISV applications fairly quickly during 2H94.

I/O Potholes

Adrian Cockcroft
adrian.cockcroft@sun.com

I'm taken by the fact that nobody has mentioned locality of I/O adapters in this thread. In most SMP architectures, all I/O adapters are about the "same distance" from all of memory—both architecturally and in the implementation. In NUMA systems and message-based massively parallel systems, this is not in general true. Placement decisions must try to achieve both locality to the processor and locality to the "relevant" I/O adapter(s). This is a significant complication that may, in part, explain why optimizing for NUMA and message based systems seems "harder" than SMP.

I don't think "distance" from the processors in a NUMA system makes very much difference, except for devices that must be stroked at very precise timing intervals. Hopefully there aren't many

of those left outside of real-time and control systems. Why: The latency involved in most I/O operations is extremely long compared with the memory-to-memory latencies being discussed, so much so that the memory-to-memory latency is noise in comparison.

The latency from the CPU to the I/O bus and I/O devices needs to be short for good performance. The reason is not obvious, but it became clear when comparing the performance of some SBus I/O cards in desktop and server systems, that the extra latency over the server's backplane (less than a microsecond of extra latency) made a big difference in performance. The reason is that unfortunately not all I/O devices are as clever or as perfect as many people think. There are devices that contain uncached memory—graphics frame buffers, some I/O cards that can't tolerate DMA latency under load (e.g. to avoid dropping Ethernet packets), and all cards that have control registers to twiddle. Access to the cards is done via "programmed I/O", where the CPU does uncached reads and writes to the I/O space. The bulk transfers happen via DMA, and are not latency sensitive.

It turns out that very many cards are developed in low-latency desktop systems (SBus desktop Sparcstations, PCI-bus desktop PC's) and the same cards are used on servers. Lower volumes for servers mean that only the most popular card types are optimized for use on servers, and real customers buy many cards from many vendors and then wonder why they don't all work so well.

The lesson Sun learned from this was that I/O bus latency does make a significant difference. The CPU to SBus latency in the Ultra Enterprise server range is actually less than the latency in the older desktop system, and the programmed I/O datapath was carefully optimised.

If you are looking at a NUMA system, try to figure out which CPU is going to run the device driver code. Then start to worry about how many round trip PIO's are required to setup each data transfer. It can actually take longer to set up a DMA access (which involves a tiny amount of control data transfer) than the DMA of the bulk data itself.

I/O Potholes

Zahir Ebrahim
zahir.ebrahim@sun.com

In addition to the "programmed I/O" latency considerations that Adrian Cockcroft mentioned, there is one other "subtle" consideration that dictates location of I/O in a distributed server that may not be visible to folks who don't actually build these systems for a living but only use them (for a living). This is the memory model consideration.

So consider this as a brief prelude to the subject that only points out some of the related issues.

Memory model mainly has two dimensions:

1) as seen by the I/O device doing DMA, and

2) as visible to PIO from the issuing and viewing processors.

For 1), it is the race between "last DMA transfer" and "completion notification" (notification to a local processor vs. remote processor, DMA to local memory vs. remote memory).

Aspects of this are managed in system hardware (not the I/O device), and other aspects are bound in the I/O device driver. The issue definately has an impact on the system design consideration for I/O.

For 2), it is a) the "side effects" of PIO operations on the I/O device, b) ordering of any semaphore updates in main memory with respect to last PIO transfer to a shared I/O device, and c) ordering of PIO transfers between I/O devices.

By ordering, I mean at least 2 things: i) the order in which the effects of the PIO from a processor becomes visible to other distributed processors (if one sees X then Y, then all see X then Y, not that some see Y then X), and ii) the order in which the I/O device sees the PIO transactions from the issuing processor.

Again, some aspects of this are managed in the system hardware (not the I/O device), other aspects are managed in the device driver interface to the kernel, and some others are handled in that I/O device driver.

Where the boundary is determines the performance vs. complexity tradeoff for supporting I/O devices exhibiting those characteristics in that system.

As one can observe, the global order considerations for PIO can be somewhat stricter than for memory, with the additional complication that I/O devices (registers) have side affects, and that latency of PIO have at least four additional perfor-

mance-related impacts that warrant considerations:

1) it effectively throttles the processor doing a PIO "load" to a remote device thus wasting processor cycles,

2) under memory models weaker than sequential consistency, the processor may have to interject barrier instructions thus throttling throughput,

3) the PIO thread may even have to be bound to a certain processor thus limiting its relocatability (which may or may not be such a bad thing), and

4) for PIO streaming devices such as frame buffers and other memory-mapped devices/interfaces which may be non-cached and "store intensive" rather than "load intensive", PIO throughput is important.

Although system software folks generally wrestle with these issues, they are also important for the application software as they determine the effective system throughput, CPU utilization, and overall perceived quality of the application running on that product.

In many of these cases, simply the proper placement of the device driver and the application thread (as opposed to the I/O device) can sometimes help improve the situation. But that needs system software assist.

And finally one can easily imagine all of the above necessitating special hardware design characteristics for I/O in a distributed NUMA cluster to even approach symmetric big-iron I/O performance, especially if there is also some issue with software compatibility (have your existing device drivers written for big-iron also run on distributed servers without change), and if there is some marketing story to tell that your IO-ops/sec is actually higher (if not downright lower) in your distributed product family than in your symmetric product family (and Adrian provided a good example of how it can subtly degrade going even from a desktop to a big-iron server, let alone going to a fundamentally different architecture).

Interpreting Benchmarks

Brad Carlile
bradc@cray.com

The paper is "STiNG: A CC-NUMA Computer System for the Commercial Marketplace" by Lovett and Clapp. I've been told that much the same data as in the paper is available at http://www.sequent.com /public/solution/numaq/l

Background: StiNG is based on the Intel SHV Pentium Pro 4-way SMP, with a NUMA interconnect based on SCI.

Here's the question: They talk about getting traces on an (UMA) SMP to drive their simulation. But the tabulated simulation parameters make it seem like the code was modified for NUMA-style locality, or something else was done to increase locality—like simply assume particular levels of locality. Which was it? This wasn't discussed.

I've also looked at this paper. The two main workloads they looked at were TPC-B (since all the transactions are simple and isolated you don't need to increase locality). And TPC-D Query 6 is embarrassingly separable as well (as long as your tables are well partitioned).

Both of these are *very* simple workloads. You can avoid the partitioned problems that affect NUMAs and MPPs in real workloads.

For others who don't have that paper, some sample results from their simulation: 12-13X speedup on a 32-way system for TPC-B (yes, B); nearly 20X for TPC-D Query 6. (Neither is a bad result, in my opinion.)

But I don't think either of these speak well for NUMA. They should be seeing near linear on these simple problems. On a 40-CPU UMA SMP system we've demonstrated 39X faster on queries using Oracle.

So why do you want to go to a NUMA that offers only 20X out of 32?!? Sequent talks about going to hundreds of processors, but at this scalability rate you'll never be able to use them.

The NUMA architecture is an improvement on the MPP architecture because it added coherency. But to get good performance you still have to performance tune it like an MPP. The latencies make you partition your database onto different local nodes—this is a pain. This is a database performance tuner's nightmare. Look at the latencies below....

— average L2 miss latency around 2 us for TPC-B

within that,

— average local latency 200-300 ns (from text)

— average remote latency 10 us (33X-50X local)

— average L2 miss latency around 1.2 us for TPC-D

within that,

— average local latency 200-300 ns (from text)

— average remote latency 5-6 us (17X-30X local)

Will NUMA die before it is born?

Register Windows

David Chase
chase@centerline.com

Having compiled for SPARC in a previous lifetime, I appreciate how these fixed-size register windows can help with procedure calls. But I never did like the fact that you got a fixed-size chunk of registers whether you needed them or not. I haven't looked at any detailed study to back my claim—but looking at C++ code, there seems to be a large variance in the number of registers used. Besides, the usage seems clustered towards the lower end—there's a better chance I use <8 registers than the contrary.

If you're willing to pony up for the compiler work, register windows can still end up a win in these situations.

First of all, you don't have to use the save/restore instructions; the ABI does not really require it. There's a little hair involving setjmp and longjmp and their common (ab)use, but I think that was getting fixed around the time I left Sun in late 1993. Given that you've paid the hardware cost of register windows, it doesn't often pay to avoid using them, unless the procedure in question is part of a deep call chain. This is doubly true on Sparc version 9 (e.g., UltraSparc), where (if I remember correctly, if the plans worked out properly) the trap model was cleaned up to allow much faster usual-case response to register-window overflow.

Second, if you can see who calls whom (that is, if you are able to form a call-graph, not always possible with calls through procedure pointers) then it is possible to do interprocedural register allocation, in the style proposed by Tom Murtagh many years ago (POPL 1984, also TOPLAS of July 1991).

Third, there's a number of low-level tricks that can be used to avoid use of a register window. There's shrink-wrapping (Fred Chow, PLDI 1988), and there's aggressive use of tail-call and leaf-routine optimizations (compilers for Sparc tend to be good at these) that help avoid excess thrashing of register windows.

In the case of clearly recursive routines, you can perform inlining along the recursive call—that is, replace

```
fib(x) = x < 2 ? x : fib(x-1) + fib(x-2)
```

with

```
fib(x) = x < 2 ? x : (x-1 < 2 ? fib(x-2) +
fib(x-3)) + (x-2 < 2 ? fib(x-3) + fib(x-4))
```

This cuts the call chain depth in half. (To my knowledge, this was more or less simultaneously observed by people working at Sun, and Mary Wolcott Hall and Anne Holler, who were respectively working on dissertations related to inlining.) Note, too, that since "fib" is clearly (provably) functional, this notoriously inefficient implementation of fibonacci numbers could be slightly improved by caching the conditionally common subexpression "fib(x-3)".

Register Windows and Delay Slots

Paul W. DeMone
pdemone@tundra.com

In the specific case of delay slots, I've been dealing with high-end processors and for those I'd have to say delay slots have become both awful and useless. With wide-issue processors not only do you need more than one delay instruction, but you need a variable number depending on where the branch happened to fall in the issue window. For example, with 4-wide issue and a single delay cycle (i.e., 1-cycle i-cache) you need anywhere from 4 to 7 delay instructions depending on where the branch was. So the solution is usually to add pc prediction hardware which removes the need for delay slots in the first place. The awful part has been already mentioned. Delay slots add a great deal of special cases which must be designed and tested. And like everything else, it gets worse as more buzzwords such as speculative execution are thrown in.

Yes, branch delay is a trick that works well for simple pipelines but makes life difficult for more aggressive implementations. And it also increases average code size because a NOP will have to be

stuffed in after a branch if the compiler cannot otherwise fill the slot with a useful instruction.

The Alpha derives a lot from the MIPS architecture but DEC architects avoided its delayed branches to help "future-proof" Alpha. The POWER and PowerPC architectures also avoid branch delay slots because of the distinct and almost autonomous nature of its branch unit (it's actually on a separate chip from the integer and FP execution pipelines in the POWER and POWER2 implementations).

The same kind of idea applies for processors with non-interlocked load delay slots (early MIPS). Average code size goes up and if future processors get rid of the delay slot then programs either can't take advantage of it or are not backwardly compatible.

In regards to the discussion of register windows and their status as "pothole" or win, it's nice to see that they can be useful. because they are certainly a pain to implement, especially on a speculative execution machine with register renaming. If nothing else there are many pages in the Sparc version 9 architecture manual referring to windows, window control registers, window traps, etc., which all have to be designed and tested and which other architectures don't need. Whether windows appear to be a win or lose seems to depend a lot on which side of the hardware/software fence you primarily inhabit.

Hard to say. Register windows occupy extra silicon area and slow down context switches on one hand. On the other, automatic integer parameter passing via window overlap can reduce the frequency of loads and stores in the code. Clever circuit design and layout (e.g. UltraSPARC) seems to be able to reduce the silicon penalty to a minor issue while the use of powerful compilers with interprocedural register allocation can reduce the cost of not having register windows. I think the pros/cons balance fairly well and it becomes more of an religious issue.

Here's something I haven't seen mentioned yet:

Storing 2 single-precision floating-point values in 1 double-precision floating-point register.

This will bugger up most RISC compiler register-based parameter passing conventions. Also forces extra logic and routing in the FPU to extract the single-precision value from the upper or lower half of the 64-bit physical register. In my opinion, the Alpha handles this best by stretching out single-precision values into a double-precision compatible format when in the floating-point registers.

The single-precision load and store instructions handle the transformation and its inverse so it's essentially invisible to the programmer. The FPU datapaths can be made more regular and therefore faster.
