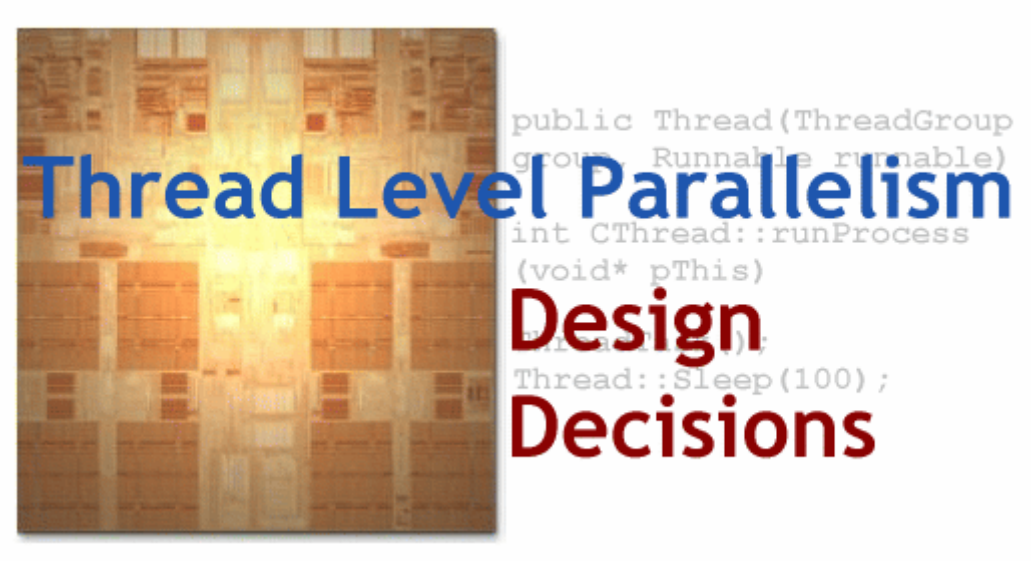
Niagara: A Torrent of Threads

By Chris Rijk – November 2004



Introduction

In my first article in this series, I analysed why taking advantage of Thread Level Parallelism would be better for server workloads. In the second article I took a deep look at Sun's Niagara design, being the first mainstream processor with heavy TLP optimisation. This last article in the series is more of a collection of thoughts on a wide variety of topics, including features already mentioned by commercial companies and some ideas of my own. The main focus is of course how best to improve performance, though there is also consideration for how system designs, OEMs, ISVs and customers could be affected.

General Thoughts on Developing Thread-Optimised Processors

To start off with, let's consider some of the design issues. Developing new processor architectures specifically for high throughput is simple in concept, but there are many side effects, both good and bad. Here I give some more basic or higher level issues.

Binary backwards compatibility still vital

This might seem like a strange thing to say, but it's all too tempting to talk about entirely new ISA designs as ways to improve performance and efficiency. (I have a couple of ideas myself) However, for server tasks, any design that is significantly more efficient than today's servers will either need super-huge caches or massive main memory bandwidth to cope with the amount of data the processors will demand. There is no way around this - higher throughput processors will process more data in the same time, which means more bandwidth.

Multi-processor systems will still need careful design to get good scalability. The more throughput a system can handle the more valuable it becomes, so high reliability will become increasingly important. There's only so much a hot new ISA design can do in face of system, storage, software, service and support costs. This is particularly important for the server market, where verification and support become more time-consuming and expensive the more architectures that need to be supported. So remember the Itanium lesson and provide hardware that can run customers' existing software even better.

Additions to existing ISAs would not be a problem, unless heavily relied upon to deliver most of the performance benefit of the new design.

The changing goals of caches

Caches exist to reduce the burden on the main memory system. Today, caches are mostly to help reduce latency and partly to reduce bandwidth requirements. However, TLP optimised designs mean stalls due to latency become less of a problem. For such chips, caches will be mostly to help reduce bandwidth requirements and partly to reduce latency. This should lead to designers choosing smaller, denser cache designs, but with higher latency. The bandwidth demands on the caches will also increase significantly, especially for any cache shared between multiple CPU cores.

The trade-off between complexity and efficiency

For processor designs where the main consideration is single threaded performance, a higher performance design generally means a more complex, but less efficient design. For highly TLP optimised designs, efficiency is more important, in terms of performance per transistor, per watt or for a given die size. While simpler designs are generally more efficient, this is not always the case. For example, sharing logic or cache between multiple CPU cores is generally more efficient, but also more complex.

Power and heat become easier in some ways, harder in others

TLP optimised designs should be much more power efficient. However, this doesn't necessarily make power consumption less of an issue. Many cores per chip means much more logic per chip and less SRAM (which is low power) so using lower power logic becomes more important. Some aspects of chip cooling should become simpler because instead of a few large hot spots (as with a single core chip), there are multiple small "warm" spots per chip, so the average temperature should be more even.

However, typical power consumption per chip will likely rise, possibly quite dramatically. Cooling a single core chip where the core consumes 50W is pretty easy, but cooling 4 on one chip (200W) would be a serious challenge. The lower power each CPU core is the more tempting it will be to put more on a chip. So the clock rate of large multi-core designs may be less transistor or interconnect limited, and more power consumption limited.

Share resources, within reason

For on-die caches, it would generally be better to share them between all cores. Even L1 caches could be shared between 2 (or more) CPU cores - more on this later. Functional units, particularly large ones which are less latency sensitive or rarely used could also be shared been CPU cores. Sharing can help reduce die size and power consumption without much of a drop in performance. More sharing requires extra logic to organise the sharing, and extra wires to shuttle data to and from the shared unit, which is less likely to be local, so there are limits.

Real work is better than speculation, speculation is better than nothing.

When a running program on a single threaded chip encounters a conditional branch, it is much better to speculate on which direction it will go rather than stall the whole pipeline. However, with multiple threads per CPU core, speculation for one thread can get in the way of real work for other threads. So getting good performance without relying on speculation would be better, though zero speculation would be overkill in almost all applications - it would be better to be able to prioritise real work over speculation.

Look for new concepts in CPU core, cache, memory and system design, and re-consider older concepts

Simply focusing on putting lots of relatively "simple" CPU cores on a chip is just the basics. Because some bottlenecks are significantly reduced, features from "fat" CPU designs to help mitigate them are less necessary, and in some cases can be removed. Because some new bottlenecks appear (or rather, some existing problems become far more serious) research in some areas will need to be increased, and some existing ideas that have not yet been implemented could be sped to market.

It's not necessarily just the processors either. Whole concepts in system design, chip packaging design, memory sub-system design and maybe I/O design might need to be re-thought. Some more efficient designs may come from "going back to square one" and starting with a minimal implementation and then re-analysing modern features, only adding them if they truely help throughput orientated processors.

Ideal CPU Core Size, or Cores Per Chip?

There would be no fixed number of cores or threads which is "best". This is partly because technology changes all the time, and also because different designs have different goals and options. The maximum number of CPU cores possible per chip would follow Moore's Law - if average chip sizes stay the same, then the number of cores per chip will continue to increase.

Though there would be no ideal, most CPU cores with multiple hardware threads will likely have either 2, 4 or 8 threads. That is, unless radically different threading technology is used. With SMT or the more coarse grained threading of Niagara and others, more hardware threads can take up any spare resources, but they also add to the costs and complexities of the chip - the marginal benefit of each extra thread falls, while the marginal cost is constant. Too many threads would actually reduce performance.

Multi-Core Drawbacks

The most discussed issue in the news is software licensing, because many ISVs charge software licenses per "processor", which many are including to mean "per CPU core". In other words, a customer deciding to buy a server with 2 dual-core processors would have to pay for a "4 processor" license, putting the overall system cost quite a bit above a system with 2 single-core processors. At the moment, it is somewhat of a fringe issue, but when Intel and AMD release their dual-core x86 processors, it will become more serious.

One problem with providing customers with multi-core processors is that there is less flexibility in what they can buy and upgrade. A 4P system (4 sockets for processors) with dual-core processors can be upgraded from one to four processor chips and two to eight CPU cores, but an 8-core processor can only have eight cores. A work-around is to sell some of those 8-core designs as 4-core ones, in cases where only four can ever be used. This is practical because when manufactured, some chips will not have eight perfectly functioning cores but will have at least four that would be perfectly fine for customer use. It's certainly a lot better to do this than simply throw away chips with 4-7 working cores.

One potential issue that is speculated about is the clock speed chosen for multiple CPU cores. After all, different processor chips from the same manufacturing process get qualified at different clock rates, so in a multi-core chip, wouldn't the chip be limited to the slowest CPU core? In practice however, this is unlikely to be much of an issue. Chips from the same wafer tend to have much more similar characteristics than chips between different batches or manufacturing lines. Two chips manufactured next to each other would be even more likely to have very similar characteristics, and qualify at the same speed. So all the CPU cores in a multi-core chip would be likely to qualify to the same speed grade anyway.

A Quick Revisiting of Sun's "Niagara"

First Niagara Chips

Since my previous article, some more information on Sun's 8-core Niagara processor has come out. Though Niagara-like chips will likely be the exception (in terms of single-threaded performance vs multi-threaded performance), it is still interesting to look at since it's the only highly TLP optimised server processor that we have a number of details on. This is not to belittle, for example, IBM's POWER5, which has 2 cores and 2 threads per core, but it's not yet at the stage where throughput is clearly more important than single threaded performance.

Niagara has reached first silicon, and is running in Sun's labs. The maximum power consumption is just 60W, but it can take 16 2GByte DIMMs, a total of 32GB of main memory, and over 20GB/s of main memory bandwidth. The 8 cores share a 3MB L2 cache, and each core has a 16KB L1 instruction cache and a 8KB L1 data cache. The pipeline is just 6 stages long and there is no branch prediction unit.

Looking at the photo, the chip has nearly 2000 pins (or rather pads) for power supply and I/O. Most of them would be for the main memory system - AMD's Opteron processor has a dual-channel DDR main memory controller on-die, but Niagara has 4 dual-channel DDR2 main memory controllers on-die.

While there has been no recent updates on the die size, based on suggestions from 2003, the chip is probably around 350mm². Compared to Intel's fastest 90nm Pentium 4, Niagara consumes about half the power, but the die is about three times bigger, which makes the power density six times smaller. Niagara should be relatively easy to cool.

Sun is already working on a "Niagara-2" processor, which includes built-in 10Gbit ethernet and is expected to arrive in a similar timeframe to Sun's higher-end Rock design. While the Niagara series is aimed at high density network heavy applications with little CPU overhead per thread, Rock will have much more serious single-threaded performance, even more than was planned for Sun's "fat" UltraSPARC V design, and apparantly has a similar number of hardware threads to Niagara.

Alternatives to Multi-Core?

All the major processors used in business servers have already gone multi-core (and multi-threaded) or will do shortly. The main companies left in this market are AMD (x86), IBM (POWER and mainframe), Intel (x86 and Itanium) and Sun (UltraSPARC). There are a few specialists, but it's almost impossible to get good information on their designs. The rest appear to be doing only incremental improvements to existing processor designs and not brand new designs.

So apart from the above 4, does anything else stand a chance, for general purpose servers? Are there any alternatives to multi-core and multi-threaded versions of existing ISAs? I think the short answer is quite definitely "no". If even Intel with support from major OEMs and ISVs is having clear difficulties with Itanium, it seems doubtful anyone else stands a chance.

So though there's lots of interesting "alternative" designs out there, I don't think they're worth covering. There's plenty of possible variance and directions with multi-core designs anyway.

Multi-Core Desktops?

With AMD and Intel battling it out in the PC desktop and portable markets, neither can afford to relent on single threaded performance or price. In addition, most PC software gets little benefit from multi-processor (or multi-core) designs, for much the same reasons that Hyperthreading has little benefit in general on desktop applications. Very few games benefit from TLP, for example, and though Photoshop and video encoding do, they're more like workstation tasks and are not mass market. Dual-core will be a hard sell to consumers, or for business desktops and I am surprised at the confident predictions of some that dual-core will become mainstream (though not dominant) within a few years in these markets.

Adding a second CPU core adds significantly to both production costs and to power consumption at the same clock rate. It's also a bit more difficult to run a larger chip at a higher clock rate. So dual-core chips will run at lower clock rates than their equivalent single-core parts. Basically, do a multi-core chip and you hurt single-threaded performance, price and power consumption. Though there is merit in this for more workstation-like tasks, it is very hard to see dual-core processors becoming mainstream on the desktop. Though a boon to the small proportion of power users and enthusiasts, it seems hard to justify in general. The minimum extra costs will certainly put dual-core chips out of the lower end of the market.

For software optimisation, we have a classic chicken and egg problem: for client software, ISVs don't go to much effort to make software benefit from multi-processor (or multi-core) systems because few customers would benefit, and customers buy few systems due to the lack of benefits. The "low hanging fruit" here is the games market, since many games are developed from scratch making it easier to add multi-threading capabilities from the start. It also helps that games have an insatiable demand for more processor power, and that they are reasonably well suited to being made multi-threaded. However, modern game engines still have long development times and going multi-threaded late in the day would be painful.

SMT can increase multi-threaded performance a bit and will likely become prevalent in a few years, but the overall benefit is minor for desktop and portable users. Still, in the long run this could slowly encourage ISVs to make more multi-threaded software, making dual-core more useful.

There is still a basic problem for the CPU companies though - it would be very hard to sell a more expensive multi-core processor that had lower single threaded performance compared to single-core designs. That companies are working on dual-core chips for desktops is really a side-effect to doing them for servers, and will be used more to test the market I think. I expect there will be a lot of press on this for many years to come, but that it will not become anywhere near mainstream this decade.

One company to watch though is Sony - their PlayStation 3 seems certain to require advanced parallel programming to get maximum performance. But with high volume, high market penetration and increased use of HDTVs, Sony could successfully push it up as a consumer desktop. They could custom develop "good enough" consumer software for the most commonly needed applications, as well as push the home entertainment "convergence" angle. This would be a wholely different sales and market model to the PC, and the main attraction to consumers would be lower prices and simplicity.

Single-Core to Dual-core Design Options

A processor design company wanting to make a dual core version of a currently single core processor has many choices to make. Let's consider the main ones.

One of the most important is the power budget - if the single-core version consumes (say) 100W at 2.5GHz, since most of the power consumption is due to the CPU core, a dual-core version would consume nearly double the power. A 200W chip would be very expensive to cool, so this simply isn't an option as an upgrade. Maybe increasing it to 120-130W would be. The smaller the power budget of course, the lower the clock rate will have to be to compensate.

Another reason to keep the power budget low is that 2 processor 1U servers today are actually too hot for many datacenters. Increasing the power budget increases the chance customers will got for lower clock rate (lower margin) models, or server blades. Heat density is increasingly becoming a serious issue for customers.

Designers can tweak the dual-core implementation to be more power efficient though, since it will be running at a lower maximum clock rate anyway. Reducing the target clock rate means less aggressive and more power efficient transistor designs can be used. In other words, due to these design changes, a CPU core in a single core chip would likely consume more slightly power than one CPU core in a dual-core chip, when both are running at the same clock rate and at the same voltage.

What cache design to go with is quite significant too. The L1 caches would never be shared between different cores because the changes required would be too significant to be worth it, but what about sharing higher level cache? For server programs, there would almost certainly be a good benefit from (say) sharing a 4MB L2 cache rather than having two cores with 2MB of L2 cache each, because there would a useful amount of data (and code) shared between the two cores in most cases. Single threaded programs would benefit from the larger cache too. The downside with sharing is that it would probably hurt the minimum latency slightly. Also, with smaller caches shared between two cores, there is a greater chance of contention over the same part of the cache. The short answer to sharing is that the more cache you have, the better it is to share it.

IBM's POWER4 and POWER5 processors share the on-die L2 cache (1.5-2MB) and multiple processors can share the monstrous L3 cache. Sun's first UltraSPARC IV design has separate external L2 caches (8MB per core), but the second generation version has 2MB shared on-die L2 cache as well as a 32MB shared off-die L3 cache. Sun's "Gemini" processor, recently canned, had two separate 0.5MB L2 caches, one per core.

AMD seems to be planning to have a 1MB L2 cache per core in their dual-core Opterons later in 2005. I think that's about the limit of the size where it's best to split the L2 cache. For Intel's dual-core Itanium's, neither the L2 caches or L3 caches are shared, which seems a bit of a waste though maybe the extra design effort wasn't worth it. What their dual-core Xeons will look like is unclear, but maybe they won't share any cache either.

How TLP Could Affect Server Competition

So far, IBM is the only company to have significantly benefited from dual-core processors, since they've been shipping them for so long. In a few years, most server CPUs sold will be dual-core (or more) and also 64-bit. The immediate effect of all this in terms of market share might not be that significant - if everyone improves at the same rate, then the overall competitive position might not change that much.

On the other hand, multi-core processors and multi-threaded cores will lead to actual performance increasing at a faster rate then before. This should lead to a decrease in the average number of processors per server: many customers will buy dual/multi-core single processor systems instead of dual-processor systems, buy dual/multi-core two processor systems instead of 4 single core processors and so on. That helps pull the market lower down, more into the territory of the x86 processors. With AMD's and Intel's x86 processors going 64-bit, that will further help them.

For designers of Itanium, POWER, and SPARC, the development costs could be reduced. After all, simpler cores means less design work to do, and also simpler testing and manufacturing in some ways. The designs could also last longer before needing to be replaced by a completely new design. Finally, they would be less at a manufacturing disadvantage, since multi-core isn't that useful on the desktop, so Intel and AMD wouldn't be able to leverage their desktop designs and volume as much as before.

Intel still seems to be going for fairly general purpose designs with future Itaniums, but IBM and Sun seem to be more willing to do "server only" designs. This doesn't mean they are abandoning the workstation market, or going for minimal single threaded performance, but that they can be less concerned about designing good workstation, desktop or portable processors. This could include major changes to the system design for IBM and Sun since they do the processor and system design together. Intel seems to be wanting to harmonise Itanium and x86 server system designs in the future, making the two lines socket compatible, though their new CEO seems to be changing their priorities.

So while multi-core will expand the lower-end of the market helping x86, it should become easier for IBM and Sun to produce more cost effective and competitive low-end systems - because of their lower volume, the design costs are much more significant for them. For Sun, doing the Niagara processors would likely not have been practical otherwise - the very simple CPU cores should be far cheaper to develop and bring to market. Sun also seems to be quite specifically going to high performance/watt for Niagara to get an extra edge - system power consumption for 1U servers is becoming a serious problem in many datacenters.

OS Scheduling and Optimisation

Having multiple cores per chip and multiple threads per core certainly will affect OS design. Two threads on the same chip or CPU core can synchronize far faster, which helps in some ways. However, with many more active threads at once, probably running slower individually, the chance of threads stalling on a lock will increase. Using finer grained locking, or better yet, lock free data structures and code will help here. Alternatively, a single thread stalling on a lock would degrade performance less when there are multiple threads per core anyway.

From the OS and application side of things, software that today can scale well to (say) 64 single-threaded CPUs should also scale well to a small number of highly TLP optimised processors. With TLP designs being more bandwidth intensive, it'd be easier to run out of system bandwidth though, so NUMA optimisation to increase memory locality will become more important.

For CPU cores that can only run a single thread, regardless of how many CPU cores there are per chip, the OS can schedule tasks pretty much in the same way. However, for CPU cores with multiple threads, in some cases it may be more optimal to switch a running thread from one CPU core to another. For example, if one CPU is running 4 threads, while another is running none, it would be better to run two on each. That requires OS support, and maybe some support on the chip.

Software Optimisation

Reducing the number of instructions that need to be executed to complete a task may help overall efficiency - leave more resources for other threads. However, that would likely be at the expense of the performance of a single thread.

Since TLP optimised designs would likely become more bandwidth limited than latency limited, using more compact data formats or other techniques to reduce main memory bandwidth needs could help. This should also improve the cache hit rate. But this might make the code more complex, and increase the number of instructions that need to be executed. There don't seem to be any easy answers to software optimisation.

Many developers may decide that since single-core and single-threaded processors will be around for some time, the effort required to make a big difference to TLP optimised processors would be too much to be worth it - many server programs are very complex these days.

For business software, there is little chance that the programs will be significantly changed in the short term to better optimise for multi-core and multi-threaded processors. This also means that the processor designers would not be able to assume that existing code would be changed to suit their purposes. In other words, processor designers should look to make existing code run as fast as possible.

Automated Parallelisation

If you have a single threaded application, one way to improve performance is to manually make it multi-threaded, and use multiple CPUs. How much this can improve performance depends on the nature of the application and the skill of the programmer. Done manually, this is typically either done across large blocks of code or large blocks of data - splitting things up at a high level. However, it is possible for compilers to attempt to parallelise code automatically. Most research I've seen seem to look at fine grained optimisation - speeding up a single loop or function. This is most likely because developing a compiler that can handle such optimisations across a wide scope would be too complex.

To date, this is almost unused in production systems - the benefits are too small. However, multi-core processors change this somewhat since the main reason performance benefits are so small is because communication between separate chips is so slow - with on-die interconnections, the latency is reduced by an order of magnitude. The local parallel optimisations compilers can make have very closely coupled threads - one might be feeding data to the other directly. Because they are so close, synchronisation between them is critical, so very low latency on-chip signaling is the fastest way. Or alternatively, different hardware threads on the same CPU core.

The biggest downside for this technique in server software is that such software really needs to be optimised for older platforms because that is where the customers are, and these optimisations wouldn't help traditional processors at all. Having separate binary versions would add considerably to development and support costs. Debugging software compiled with auto-parallelisation may also be a nightmare. Given the effectiveness of manual parallelisation in comparison, I don't expect automated parallelisation to become common in production systems. HPC software will likely continue to be the most common exception.

More Advanced Reliability Features

Computer reliability is a serious issue, and unfortunately it is not possible to have 100% reliable hardware or software. However, a lot of reliability problems are as a result of poor setup and administration, though vendors should also be responsible in making it simpler and easier to use their products reliably.

At the more paranoid end of the computing market, where computer errors could result in people dying, hardware and software really does have to be highly reliable. One problem is that high energy particles can cause "soft errors" in chips - cause a signal to change. SRAM in caches is relatively susceptible to this, and it is increasingly becoming a problem as circuits get smaller. Modern caches have error detection and correction codes to reduce the problems to an acceptable level. But what about calculation errors in logic? These are extremely rare but they do happen.

The best way to detect errors in calculations is perform the same operation twice, with different circuits, and compare the results. With duplication, errors could be detected, and the instruction could be re-run. If multiple duplications are used, voting can be used to determine the correct result instead of a re-run.

The simplest way to duplicate the operations is to run the same program on two different CPU cores. With single core processors, this is slow - the two CPU cores need to be synchronised over a slow external connection. With multi-core chips, this synchronisation can performed efficiently on-chip.

To date, hardware and software supporting this has been rare, but I expect it will become much more common by the end of the decade. Then users will have the option of running software at maximum performance or at maximum reliability.

Heat and Power Consumption

Though TLP optimised designs should be quite a bit more energy efficient, this doesn't necessarily mean their total power consumption or heat dissipation would be lower than current designs. In fact, the performance benefit from increasing the power budget should be higher, as TLP designs would not be pushing the bleeding edge in terms of clock rate and because they would be less latency sensitive. In contrast, with "fat" CPU designs, a 100% increase in the power consumption budget may only yield a 20-30% increase in performance.

Companies like IBM and Sun might choose to go for one processor per motherboard - a large, highly integrated high performance design. Having just one would make cooling it simpler, and make it easier to support a high power budget. Still, cooling chips consuming 150W or more is not easy at all.

One design aspect does become easier however. With a single core design, typically the hottest section for the chip would be the floating-point unit. A multi-core chip would not have a single large hot-spot, but several warm-spots spread around the processor. Multi-core chips would have more logic and less cache (which doesn't consume too much power) so the actual usage of power would be spread more evenly around the chip. This makes cooling somewhat easier.

Since TLP optimised designs would have smaller, simpler cores, it would be natural to assume that the pipeline lengths would be significantly smaller. Certainly the smaller and simpler a CPU core is, the shorter the pipeline can be for a given clock rate. However, using a longer pipeline for the same clock rate allows the CPU core to run at a lower voltage or use more power efficient transistors, reducing power consumption. The lower the power of each core, the more can be put on a single chip.

Cores Per Chip

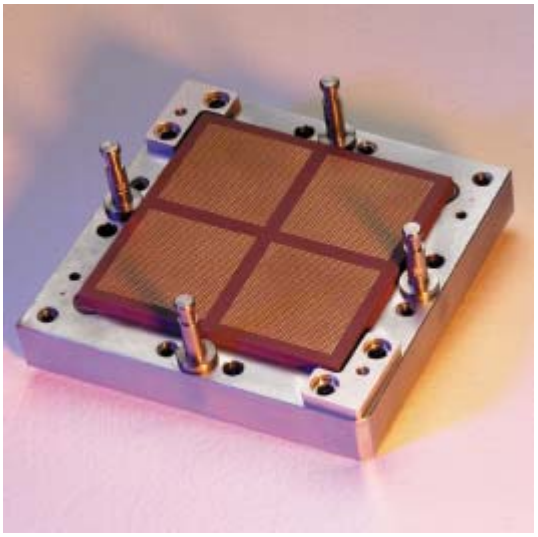
Larger chips are more difficult and expensive to manufacture. They're also more difficult to design well. On the other hand, larger chips also have the highest performance. Larger chips also means more CPU cores can be used, and typically, higher power consumption. Though it's not the absolute limit, it is very rare for processor chips to be larger than 450mm², which places an upper limit on the number of CPU cores per chip on any given process technology. But is there a sweet spot?

Compare two systems, one with two dual-core processors and one having 1 quad-core processor, with the same core design, the same amount of total cache and memory and I/O bandwidth, and running at the same clock rate: the quad-core version would be somewhat faster and have slightly lower power consumption, compared to the two dual-core chips in total. This is because the connections between CPU cores would be both faster and lower power, by being on-chip.

However, the quad-core chip would be more expensive to manufacture than two dual-core chips. However, a motherboard with one processor socket would be cheaper than one with two processor sockets, and that might make up for the difference in manufacturing costs. This would also have implications for the memory system and I/O system. Choices, choices.

Choosing how many CPU cores per processor chip is also affected by the size and power consumption of the individual CPU cores of course. The CPU cores on Sun's Niagara chip are so small, it would probably be possible to build a 32-core version at 90nm. However, the shared L2 cache and I/O logic would have to be moved off-chip, which could be done using a Multi-Chip Module. By putting chips close together in the same package, but still using one processor socket, MCMs have better electrical characteristics than completely separate chips, so the connections can run at higher clock rates and lower latency.

What this means in the end is that there is no real "sweet spot" for the number of cores in general. But there would be one for each given CPU core design, which would depend on the size and power consumption of each core, and how expensive and power hungry the manufacturer wants to make each processor chip.



POWER4 MCM

My expectation is that AMD's and Intel's x86 processor designs will not quickly move to quad-core, or large chips. I think this would put too much of a strain of their current system designs, as well as not being that efficient given the limits on the power budget. I expect IBM and Sun to take a more radical route, using new processor designs with many cores, together consuming a lot of power. I also expect them to use more radical system designs, compared to x86 systems and their own current systems. Intel is more willing to do very large chips with very high power consumption for Itanium, so they may do quad-core Itaniums before quad-core x86 chips.

In the longer term for IBM and Sun, I'd expect them to move towards using advanced chip packaging technologies to split large processors with many cores into several chips with few cores. That could be used to reduce manufacturing costs and also to increase performance compared to the alternatives.

In the longer term for the x86 server chip designers, there's a tricky trade-off - by 2010, even 16 x86 cores could fit on a single chip without too much trouble. The problem is power consumption and optimisation: a 4 core chip could use 50W cores that are also used in 100-120W desktop processor designs, giving a 200W processor. Doing much more than 200W may well be impractical for volume systems. However, 16 cores with a 200W power budget means each core is limited to about 12W. If, as seems possible, most desktop processors are still single core by then, that means limiting a 100-120W desktop processor to just 12W. It would be much more efficient to design a different "server only" CPU core specifically for 12W, since it would be smaller and simpler. Such a design could be used for portables too though.

Alternatives and Headaches for Big Cores

Today, IBM use more advanced packaging in their POWER4 and POWER5 processors (and other products), almost as an alternative to a more conventional motherboard. As noted previously, the same technology could also be used to as an alternative to having a large single chip - for example, by using four quad-core chips instead of one 16-core chip. This could help reduce manufacturing costs, and have other benefits. It is also more practical to do this with multi-core chips, since the ways in which a single core chip can be split are rather limited.

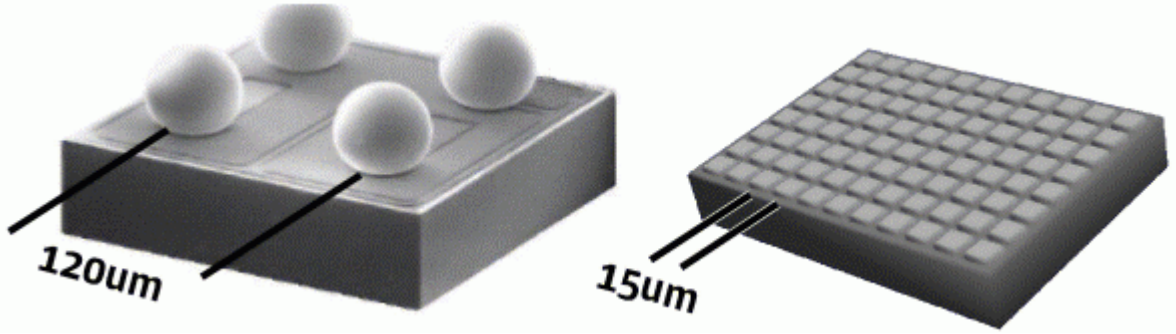
If all the chips in a MCM aren't identical it could mean having more flexible design and upgrade choices. For example, some of the chips could have 4 cores plus a small L2 cache shared across the chip, connecting to a different chip which has main memory controller logic, I/O logic and a large L3 cache. This means the two chips could be upgraded independently, and could also use different process technology.

Using MCMs could also make life easier in other ways - with larger chips, sending signals and power to very exact margins is more difficult, and they are harder to manufacture. Putting it another way, smaller chips are easier to run at higher clock rates as well as being naturally cheaper.

However, a group of small chips in a MCM would not be as fast as single chip equivalent would be. The latency and bandwidth between chips would be better than separate processors across a motherboard, but on-chip would still have a clear edge.

In 2003, Sun announced an alternative way of sending signals between chips called proximity communication. The image below compares it to current methods of wiring up chips - before the actual wires are attached to the ball bonds. For more information on this see the "Proximity Communication" presentation and audiocast from this SunLabs day.

This technology holds out the promise of signaling that is almost as good as on-die connections in terms of latency, bandwidth, size and power consumption. Though this technology has been demonstrated in the labs, it will likely take until 2010 to bring it to market. If the technology could be realised it would certainly make a large difference to system design, since there would be a much greater incentive to couple chips closer together. The biggest benefit would be for memory limited applications, such as HPC and high-performance 3D graphics.



Area Ball Bonding vs Proximity Communication

Asynchronous Logic

In theory, using asynchronous would help alleviate a number of nasty design issues. No more clock distribution, which would save both power and design time. Blocks of functionality become easier to re-use and port to new process technology due their self-timing nature. Overall efficiency should improve. These are not the only potential benefits, and they're certainly not minor, so why isn't asynchronous logic used more? Most designs have no asynchronous logic, and the few mainstream designs that do only have very little.

Basically, asynchronous is very hard to design and test, currently. Part of the problem is lack of experience, and part of the problem is lack of tools to build reliable, competitive asynchronous circuits. To put it another way, asynchronous logic requires a lot of bleeding edge R&D to be useful in high-performance CPUs. Normal logic gets massively more R&D naturally since it is already widely used, making it difficult to catch up.

So don't expect any major chips to be mostly or entirely asynchronous any time soon - none are being developed currently for a start. However, there is some low-hanging fruit - areas in which the design and test effort would be relatively small, but the payoff particularly large.

In a multi-core design, a large but subtle method to make the design easier and possibly faster would be to isolate each CPU core, the L2/L3 caches and I/O logic from each other using asynchronous logic. So you have small blocks of synchronous logic, separated by asynchronous logic. Taken to an extreme this would mean each synchronous block could run at an entirely different clock rate and also out of phase with the rest of the chip. Using asynchronous logic to communicate between the blocks means the different clock domains can still communicate reliably.

Since this would be mostly about passing data around, the amount of actual logic would be small, making it easier to design. Once this technology matures, the end result of all this should be some significant savings in design time, since the clock distribution should become simpler. This should also make it easier to run the design at higher clock rates.

Actually making all these long wires between the different blocks run fast is not a trivial task. It also just happens to be something that asynchronous logic could do a better job of, as this paper indicates.

In the longer run, making a fully asynchronous CPU core would be easier with simple CPU cores. Making a 100% asynchronous version of something as simple as a CPU core on Niagara would be far easier than a large super-scaler design with out-of-order execution. As well as having the super-simple Niagara design, Sun is also quite aggressive about developing asynchronous technology, so it'll be interesting to see what they get up to, though they are by no means alone.

Threads Per Core

Nearly all SMT designs to date have had just 2 threads - why don't they use more? More threads per core means more logic and power consumption, and may put pressure on clock rate or number of pipeline stages with the extra complexity. So if single threaded performance is a serious concern, more threads per core don't help. Although more threads means the core is less latency sensitive, it also means a lower cache hit rate on average, which puts more demand on system bandwidth and can limit overall performance.

More significantly, the performance gain from each extra thread would also tail off quickly. One extra thread on the Xeon adds about 30% performance, but going up to 4 might add just 20% more. However, supporting 4 threads would likely force some design changes which would reduce performance. How performance increases with each extra thread depends on how efficient the basic CPU core is. More "braniac" CPU cores which can issue many instructions per cycle rarely do so in practice, leaving gaps that other threads could fill. "Speed demon" designs would leave fewer small gaps per cycle, but would have larger gaps due to more processing stalls due to cache misses.

In comparison, TLP optimised designs may well have 3-way or 4-way issue cores like "fat" cores today, but they would have fewer features to improve IPC per thread. Simpler out-of-order execution logic, simpler branch prediction, simpler instruction issue and execute logic. Instead, they would rely on extra threads to fill all the execution slots. This would make it both simpler to support more threads and increase the benefit of them.

I expect that "fat" or fairly fat CPUs processors will be 2-thread SMT for the next few years, but eventually move to 4-threads. I expect most TLP optimised designs to start with 4 threads and some to eventually move towards 8. I don't expect 16-thread CPU cores for general purpose servers this decade. Though for some more specialised designs and embedded tasks, this has already happened. For example, Cray's 128-thread "MTA" processor, which switches to a new thread every clock cycle.

Advanced Prefetching

Since programs on single-core and single-threaded processors are often latency limited, doing something to reduce average latency would obviously help. However, unlike with HPC code or multimedia code which has regularly structured data and is often accessed serially, business logic is quite messy, so hardware cache pre-fetching isn't effective. At least, today it isn't, but maybe "intelligent pre-fetching" will help.

On a multi-threaded CPU, one hardware thread could work along-side the main thread, helping to improve cache hit rates. By running the second thread ahead of the main thread, but doing no real processing, just fetching data, the second thread could help "warm up" the caches. If the second thread can exactly predict what data the main thread will need then it could be quite efficient - static or hardware pre-fetching today can be overly optimistic and actually hurt cache hit rates in some cases. I doubt the second thread could handle all possible cases, because it would not be able to mimic everything, but being 90% accurate may well be possible. So long as the second thread does no write operations, it can be entirely safe to run it.

Sun has a technology called "hardware scout", which was described by Marc Tremblay in our interview as "Your ultimate fetch that has runtime information. Blows away what we can do with software pre-fetch." Sun also have a technology called "execute ahead" which sounds related but we know only the name. Since Marc is working on the Rock design, that may well be the first Sun processor to use these ideas, though there is nothing official. Intel has also talked about "warm up threads" and similar ideas, and maybe their 3rd generation Itanium design, Tukwila, will have something like this.

Sharing Logic Between CPU Cores

The relative positioning of some parts of the CPU is very critical. For example, the register files and basic integer instructions all need to be very tightly coupled together for maximum efficiency. However, some caches and the FPU (and multi-media instructions) are easier to separate and move around, and in several cases used to be on separate chips. I mention this since the easier it is to move parts of a CPU core without hurting performance, the easier it would be to share them.

Consider two CPU cores right next to each other. Normally, each would have a "private" FPU, even though in nearly all cases, the units average use would be well below peak, even with multiple threads per core. So, why not share them - two CPU cores share one FPU unit. This would reduce the amount of logic required, the chip size and the power consumption. Some extra design work would be needed to handle the sharing though, and could place some constraints on the arrangement and layout of the CPU core.

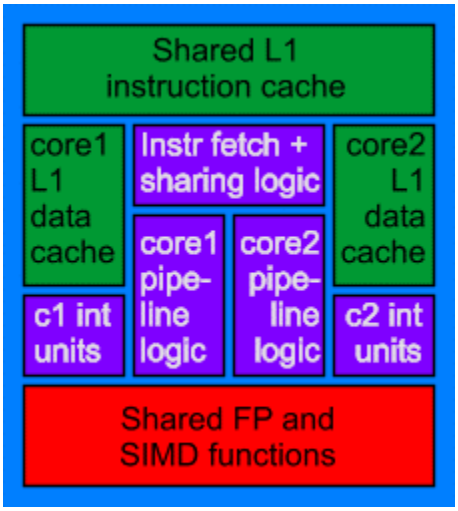
The overall reduction in performance should be minor, so this is a way to improve overall efficiency. As an alternative, instead of simply sharing 1 simple FPU between 2 cores, how about sharing 1 advanced FPU instead - use the same amount of logic, die size and power as before, but increase performance.

I think it would be worthwhile to consider sharing the L1 instruction cache too. The effect on the pipeline should be minor and doubling the size of the shared instruction cache should provide a useful improvement in the cache hit rate. I think sharing the L1 data cache would be much less practical, since it's in a much more critical part of the pipeline and the sharing benefit would be less. However, since multiple threads per core makes CPU cores less dependent on latency, it would become more practical to share L1 data caches than without it. More on this later.

An illustration of how sharing could be implemented is shown on the right. The two CPU cores are basically mirrored, apart from the shared parts. This mirroring is very logical but hard to explain simply - the basic reason is that this makes the sharing simpler and more efficient to implement since the equivalent parts of each core are directly facing each other.

An obvious question at this point would be, could this be extended to four CPU cores? Sharing between two is good, so sharing between four should be better right? Well, maybe not. The main problem is related to the mirror issue above - with two CPU cores, the whole length of the line that divides the CPU cores is a naturally efficient place to handle the logic of sharing. With four CPU cores, in a 2x2 tile, the only equivalent would be the point at the center. Maybe sharing one functional unit could be done in the center, but even then you'd end up with some asymmetry in the CPU cores. This requires extra wires and logic to work around, which generates inefficiencies. This doesn't mean it wouldn't be better, but that sharing has diminishing returns.

Some parts of the design, like the highest level cache and any I/O logic, are probably best shared between all CPU cores. Separate caches means multiple copies of the same data will exist, which is less efficient. Nevertheless, having 16 multi-threaded CPU cores sharing a single L2 cache may be too difficult to design well. It might be better having several L2 caches, each shared between some of the cores, or for the L2 cache to be split into seperate parts like on Niagara.



Two cores with shared instruction cache and FPU/SIMD units

Pipeline Design

TLP optimised designs will use simpler, smaller cores, run at lower clock rates, so they're going to have short pipelines, right? Some certainly, but not necessarily all. In general, a longer pipeline (with otherwise the same features) can enable the CPU to run at a higher clock rate at the same voltage on the same semiconductor process. Alternatively, it could be run at the same clock rate, but at a lower voltage.

In other words, a longer pipeline can be used to reduce power consumption. The transistor-level implementation of a "low power long pipeline" design and a "high performance long pipeline" design would be rather different though.

With multiple threads per core, the cost of a branch prediction miss can be reduced, making longer pipelines less of a performance issue than before. However, miss-predictions would still be a drag on performance.

Pipeline lengths have been gradually increasing because the designs are becoming more complex as more features are added. But it's also because the interconnects (wires between transistors) are increasingly becoming the bottleneck for the clock rate. This is because it's harder to improve interconnect speed than transistor speed. Basically, long wires have to be cut up into separate pipeline stages when before they could have just been one pipeline stage.

Not everything points to longer pipelines however - in the previous article, I briefly discussed caching the results of some pipeline stages (some parts of instruction decoding for example) in the instruction cache. That could certainly take out a lot of logic between the instruction cache and the execution stage of the pipline, improving efficiency. If such instruction caches were also shared between cores, that'd further help the efficiency.

Cache Design Trade-offs

Consider a TLP optimised core with 4 SMT threads which can issue 4 instructions per cycle. Typically, about 20% of instructions would be memory loads, so the demands on the L1 cache in a normal design would be quite high. Being able to do 2 loads per cycle would probably provide a useful performance boost. However, doing a full dual-ported cache (two reads and two writes to any location per cycle) would approximately double the physical size of the cache, or halve the capacity for the same physical size. Alternatively "psudo dual-ported" cache can be used - requests to separate banks (associative blocks) can occur in parallel, which would probably give most of the performance benefit for much less area cost.

The size of the L1 cache also has to be balanced with the L2 cache design - the better the L1 cache hit rate, the easier the L2 cache design becomes. However, a better L1 cache hit rate generally means a larger (more expensive) cache. Also, the bigger the L1 cache and smaller the L2 cache is, the less overall cache sharing there is, which hurts efficiency and makes the memory design (and any L3 cache design) harder.

There is one simple low-cost way to improve the L1 cache hit rate though - use denser SRAM. Often the densest SRAM design available on a particular manufacturing process is about 2-3x denser than the L1 cache SRAM used. This is for the simple reason that (current) L1 cache designs need to be very low latency, which means fast SRAM must be used. Generally the only way to improve SRAM speed with a particular design and process technology is to increase the size, which increases the signal strength. TLP optimised designs are less latency sensitive, so slower but denser SRAM could be used.

There is still a bandwidth problem to solve though.

Time for a "L0" Cache?

I think a good way to ease these trade-off problems would be to introduce a slightly unusual cache - let's call it "level 0" or L0 for the sake of argument. (Some older references to L0 cache really mean L1 cache though) The idea is to cache just a few blocks of memory for each hardware thread - 8-32 blocks of 16 bytes for example. These of course would have a 1 cycle access time. Since this would be on a per thread basis, each thread could do a L0 cache access at the same time, meaning the CPU core could do as many loads per cycle as it has threads.

But what would the cache hit rate be? If the L0 cache hit rate is poor then there's little point in the whole exercise. I think in many server programs it would be about 70%, though I haven't managed to find any papers which analyse something quite like this, so this is just a guess. This may sound unrealistically high, but even with spaghetti server code, a lot of memory accesses would simply be reading a few elements from an array, or fields from a particular object. Even if this is only a short array, that's still a number of sequential accesses, which would have a good hit rate with this "L0" cache.

If the L0 average cache hit rate is 70%, that reduces by 70% the number of requests that go onto the L1 cache. Though small, relative to the chip as a whole, the L1 cache would be far more complex than the L0 cache, so using the L1 cache less would actually save some power too. The average memory latency would also fall, improving performance. In addition, by reducing the dependence on the L1 cache, the L1 cache design can be changed in more flexible ways. In other words, it becomes much more practical to use high density SRAM or to share the L1 caches with another CPU core.

A L0 cache should work particularly well for instructions and have hit rates of around 80% or more. Enough to cache typical inner loops. The instruction cache is also more suited to sharing between multiple cores, as mentioned previously. The L0 instruction cache could also be used to cache partially decoded versions of the instructions, and so shave some stages off the pipeline between the L0 instruction cache and the execution stage.

Because L0 caches would reduce the demands on the L1 caches, it would become more practical to share the L1 caches and to use denser but slower SRAM. How easy and practical it would be to share a large L1 instruction cache between 4 CPU cores though depends on how much performance it has to provide. The more instructions per cycle each core can execute and the lower the L0 instruction cache hit rate, the greater the demands on a shared L1 instruction cache. It would certainly be easier to share a L1 instruction cache between 4 tiny single-issue Niagara type CPU cores than 4 4-way issue cores.

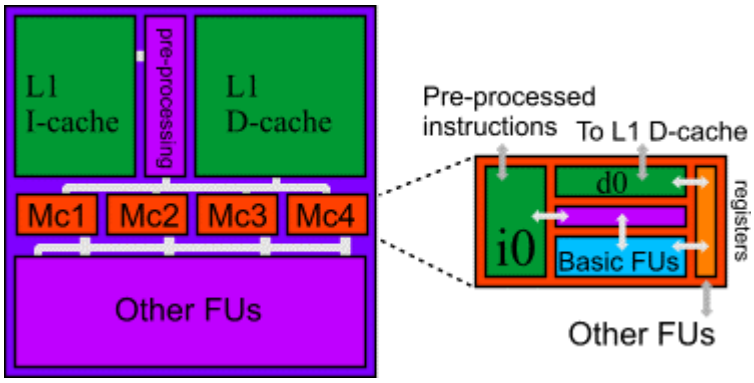
An interesting thought occurs - if using L0 micro caches means the L1 caches can become bigger and shared, then maybe there's some merit in making it more like a L2 cache in that instruction and data entries are unified, rather than split. For example, maybe two cores could share a 256KB unified L1 cache, while having separate L0 instruction and data caches per hardware thread.

Putting Together Several Ideas

I've put a number of ideas in this article about what designs might happen in the future. However, that is mostly just what we can imagine today, and I'm sure we will see a number of other interesting design ideas as time goes on. While working on this article, I had a strange idea, which is rather subtle and counter-intuitive - but I decided to describe it here anyway to help stimulate some discussion.

It's probably better to describe this idea as a next-generation CPU design based on something like Sun's Niagara processor. Consider the Niagara CPU core design today - a very small, low power and simple core that executes just one instruction per cycle, but can swap between 4 threads. Then consider a theoretical second generation design where all 4 threads can be active and executing instructions at the same time, but sharing the same functional units. This would require a copy of parts of the pipeline for each hardware thread. This increases the maximum number of instructions per cycle per CPU core to 4, though the maximum per thread would still be 1.

In this next-generation design idea of mine, not only are parts of the pipeline copied per thread, but some of the execution units are too - the smaller, more basic, more commonly used ones. This results in micro CPU cores within the larger CPU core, and each micro core has just one hardware thread. Compared to the 4-thread design suggested above, we effectively have 4 micro-cores, taking up only a small fraction of the total size, but which handle much of the processing - perhaps 90% on typical server code. The more complex and less frequent execution units are still shared though.



A simple CPU core with 4 micro-cores

With the first generation design, at most 1 instruction per cycle could be executed per cycle. The second generation design allows for a peak of 4 instructions per cycle. The next generation design also has a peak of 4 per cycle, but since each micro-core is self-contained and very small, it should be quite practical to run them at 2x to 4x the clock rate of the "main" CPU core. Smaller circuits are easier to run faster, and would also consume less power compared to running the whole CPU core at 2x to 4x the clock rate. A follow-up design could make the micro-cores asynchronous - since they'd be so small, this would be much more practical than otherwise.

Since some of the execution units would be copied, there is also less chance of conflicts over execution units for common instructions. Execution units for other common instructions can be quite small: add, subtract, logical operations, compare instructions and branches. Though branch prediction logic can get very complex, a very simple core could largely dispense with all this, partly because the pipeline should be very short.

This also makes a good case for using tiny "L0" caches as explained earlier in this article - this is particularly helpful since loads and stores are quite frequent. In addition, by pre-decoding the instructions after being read from the L1 instruction cache before being placed in the L0 instruction cache, each core would need less copied logic, could have a smaller pipeline and generally be smaller and more efficient.

An extreme implementation could use a super-pipelined add/subtract unit - instead of doing a full 64-bit addition in one cycle, the implementation could do 16-bits at a time, with early termination. The idea is to maximise efficiency, which means optimising for the average case. Today, processors are mostly designed for the worst case.

However, any floating-point execution units are large and would be shared, so this isn't going to improve any HPTC benchmarks. In fact, with more power going to the integer only micro-cores, the rest of the CPU core might have to be clocked a bit slower.

It's hard to say just how small such a "micro-core" could be, particularly since it would have to be 64-bit and run at high speed. The basic architecture would make a large difference too, mostly in how many registers there are. The original 32-bit ARM-1 processor had about 23,000 transistors. A 64-bit version would obviously require more - approximately double. However, modern CPUs have a lot of little add-ons, from reliability features to extra data on the CPU state. Doing a 64-bit "micro-core" with less than 50,000 transistors might be impractical, though probably manageable with 100,000.

One of the critical reasons to get the transistor count low is to reduce power consumption. A chip with 4 micro-cores per "real" core, and 8 cores in total would have 32 micro-cores. If each micro-core consumes 1W, that's 32W just for the micro-cores.

It's interesting to compare the difference in maximum and average number of instructions per second between the 3 generations, assuming that the same process technology is used. With the first generation, each chip can manage a maximum of 1 instruction per cycle, but the average would be close to that maximum. With the second generation, the maximum quadruples, though the average would probably only double. The cost would likely be small, with size and power increasing by maybe 10% per core, though the size increase of the whole chip would be rather less. Doing high-speed micro-cores running at (somewhat optimistically) 4x the original clock rate, further quadruples the maximum

instruction rate, relative to the original. Again however, the average increase (ie actual gain in real-world performance) would be less, perhaps 2x.

In total, that'd give a 4x improvement in performance over the first generation without increasing die size or power consumption much. Well, that's the optimistic estimate at least. One major issue would be that the much faster CPU cores would put far more pressure on the L1 caches, the shared L2 cache, the memory system and I/O system, all of which would need to be upgraded.

Whether this idea is really workable or not, I hope it gives a sense that there's much more to highly TLP optimised processor designs than putting a few CPU cores on the same die and adding a hardware thread or two to each core. I would expect future designs to have interesting and entirely new features and layouts, and less of "more of the same".

A 2P (2 processor socket) x86 system is definitely "low-end" - since just about anyone does them. 4P systems are not that much of a stretch and many OEMs sell them, though the market is dominated by Dell, HP, IBM and Sun. Slicing the market into "low-end" and "high-end" is somewhat arbitrary, but since Dell have moved away from doing 8P systems, it could be said that 1-4 processor chips per system is the extent of the "low-end" or "volume" market. Price and performance are both important factors for low-end systems.

Given how high-margin higher-end systems can be, it may seem surprising that more companies don't do larger systems. However, while 2P is now common, designing a 64P system that scales well, performs well and has good reliability and usability features is a serious technical challenge. While 2P systems would often be running just one application, high-end systems will often have multiple applications running on the same system, generally with the aid of some hardware or software partitioning.

One way to describe "big iron" is that it is a hardware solution designed to make the software side easier and simpler. In this view, a cluster would be the opposite - the software is more complex so that the hardware can be simpler and cheaper.

Multi-core chips, and highly TLP optimised designs in particular, will affect the system design, so it's worth considering how different system types may change.

Single Processor Systems and TLP

Today, 2P servers outsell 1P servers by about 2-to-1 despite being more expensive. This is probably because the price/performance and density of 2P systems is better. With TLP optimisation significantly increasing the performance per chip, 1P systems will become higher end and more expensive on average. But because 1P dual-core systems will be more like 2P in terms of price and performance, I would expect the unit volume share of 1P systems to increase, and the share of 2P systems to decrease. Putting it another way, while the average number of CPU cores per system will increase, the average number of processor chips per system will decrease.

A 1P system has some natural advantages over a 2P system: no processor-to-processor connection, no split or multiple memory controllers, a straight-forward motherboard design and so on. I would expect some companies to do processors that are 1P only, including having a unique socket design, instead of trimmed down versions of SMP capable processors like today. This should lead to system-on-a-chip like designs, which would be particularly effective for low-end systems or high-density systems.

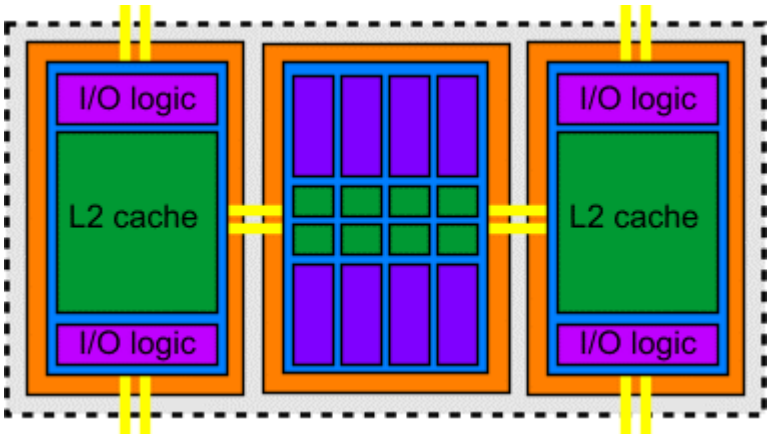
An obvious example of this Sun's Niagara chip, which seems to have just about every major motherboard component on-chip. This can save a lot in chip packaging costs, motherboard design and manufacturing costs, and also testing costs. It also reduces power consumption (less I/O between chips) and can increase performance with faster on-chip connections. But design costs increase. A 1P only design doesn't need a SMP interconnect, or the logic on the processors to handle cache coherency between chips. However, a multi-core chip does still need some cache coherency logic between the cores on-chip, and with the I/O system.

How much difference this all makes in the end is hard to say, and there are no actual shipping examples to compare with. For low-end systems, cost is critical and in theory a highly integrated solution would be best. However, there is a couple of potential issues. Perhaps the biggest is simply having a "all your eggs in one basket" main chip which has to be developed and tested all at once, while development of today's systems is more spread around. In addition, on most motherboards, only the main processor uses bleeding edge chip manufacturing technology, while the other chips use cheaper, more proven technology, though slower. An all-in-one solution has to use bleeding edge process technology for everything.

There is also a reduction in flexibility. A multi-chip solution can be upgraded, adapted or customised with new chipset parts. An all in one design is rather more fixed - doing different designs is possible, though rather more expensive, since the whole new chip has to be tested and so on. Of course, a Niagara-like design does not have to be a pure system-on-a-chip, and for minor, low-performance parts, off the shelf components can be used. In addition, a lot of chipset features are pretty standard, so a reduction of flexibility is not necessarily a major loss. Most blade servers make a similar trade-off between flexibility and suitability.

There is one final issue - larger highly integrated chips are more difficult to manufacture, and also more difficult to run at high clock rates. Since chips have to be clocked by their slowest part, the more variation the worse average performance will be. This is a problem that will get worse as feature sizes get smaller. As noted in a section above, asynchronous logic can help split up the chip into multiple domains, which should help alleviate some of the problems.

Maybe there is an another alternative solution that can overcome these downsides - still have a multi-chip solution, but all in one package, instead of spread over the motherboard. The traditional way of doing this would be with a MCM, but there are new concepts such as stacking chips or Sun's proximity communication technology. Of these, proximity communication is the most unproven, but also has the best potential since in speed and power consumption it should be very close to on-chip connections.



Processor with auxiliary chips

However, a multi-chip package would not necessarily be split like multi-chip chipsets today. To the right is an idea of how to split a Niagara-like chip: the group of CPU cores in the middle stays the same, but becomes a chip by itself. The shared L2 cache and all the system I/O are moved to a pair of identical chips. Since there would be a general purpose switch between all the CPU cores and the rest of the chip anyway, this would require little in the way of new logic, aside from the chip to chip connection itself.

As well as resulting in smaller, more easily manufactureable chips, this gives a small amount of flexibility - the I/O and L2 cache chips could be upgraded separately to the CPU cores, and since there are two, a lower-end processor package could have just one, instead of both. Finally, the L2 cache and I/O parts would not (necessarily) need bleeding edge technology, and could use slightly more mature technology compared to the CPU cores, without a significant reduction in performance.

What this means overall to costs and performance depends of course on the actual solution chosen. MCMs and other packaging technologies are not necessarily that expensive. How proximity communication will compare is hard to say, since it is so new.

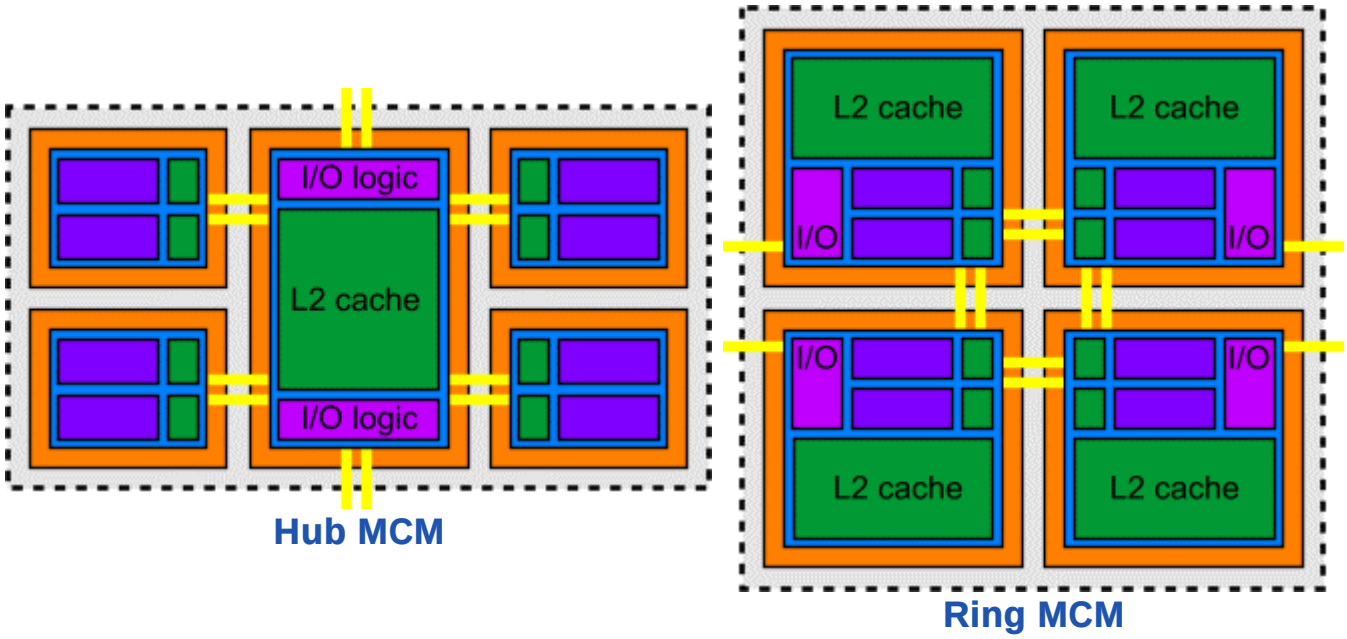
Low-end Multi-Processor Systems and TLP

CPU design companies developing dual-core chips for this market plan to keep their system designs identical, or very similar, to their current single-core system designs. This puts more pressure on system bandwidth and power consumption. The dual-core versions will also be more expensive than the single-core versions, on the same process technology. Like with 1P systems, the average price of a 2P system or 4P system will increase, but some customers will buy a 2P instead of a 4P and some will buy a 4P instead of an 8P, and so on.

As processors become increasing efficient with heavier TLP optimisation over the coming years this could force some re-thinking on system design. A rapid increase in TLP optimisation and number of CPU cores per chip will result in vastly more powerful processors in a short period. Such cores will either demand large external caches or massive amounts of main memory bandwidth. Very large caches are expensive, so are unlikely to be a common solution.

Using commodity main memory there are limits to how much main memory bandwidth you can have with on-die memory controllers - Sun's Niagara design seems to be close to the limits. The bandwidth demands would likely be too much for bus based designs as well. In addition, main memory requirements would rise in line with performance, and unless much denser DIMM chips become available in volume much sooner than anyone expects, single processor chips will start requiring 16 or even 32 DIMM slots each. If that happens, main memory daughter cards will become more common, because there just wouldn't be enough space on the motherboard anymore.

There are alternatives of course. The simplest is to keep the number of CPU cores per processor low, meaning performance does not increase so rapidly. So far, this seems to be what AMD and Intel are doing - incremental steps forwards. Staying with 2 CPU cores per processor chip would likely be enough that current system designs do not have to change significantly.



One interesting alternative is to use a single processor module that uses just one processor socket, like with the 1P system example above, but with multiple separate processor chips. There's a number of different ways to achieve this. One would be to have several identical chips, each with some CPU cores, some shared L2 cache, some main memory controllers and some I/O. Instead of a "ring" of identical parts, another option would be to have a central hub ASIC with some shared cache, main memory and I/O bandwidth. Around the hub would be some simple processor chips with two or four CPU cores and maybe some shared L2 cache and a connection to the hub.

However, this really becomes just a variant of the 1P socket MCM above. But it does lead to an interesting point - doing very large chips or complex packages for 1P systems would lead to a "1P" system that was actually pretty high-end, and might require a whole motherboard of main memory to supply it. In that case, a system with 2 such processor modules would likely be expensive and large enough that it could not be considered "low end" or "small".

Large Systems and TLP

It's hard to really nail down the difference between a cluster and a large SMP system, and it's not becoming any easier. Saying a "big box" doesn't really cut it, since a server blade system is still the same as a number of 1U servers from an application point of view. The term SMP (Symmetric Multi-Processor) is somewhat redundant these days, but basically means a group of processors where all are equal and all have direct (hardware) access to all the physical main memory and I/O. This is the easiest model for software developers. In large SMP systems, instead of one "motherboard" there are many "CPU/memory" boards, typically with 4 processor chips. The main point then becomes how different CPU/memory boards communicate with each other.

With many database applications on large SMP systems, about 50% of main memory requests will be to remote CPU/memory boards, even with data placement optimisations. That is a huge amount of traffic, which really requires a custom hardware solution - a "backplane". Ethernet is about 100x slower. Even Infiniband would not be able to handle that, despite having much better latency and bandwidth and requiring much less OS overhead.

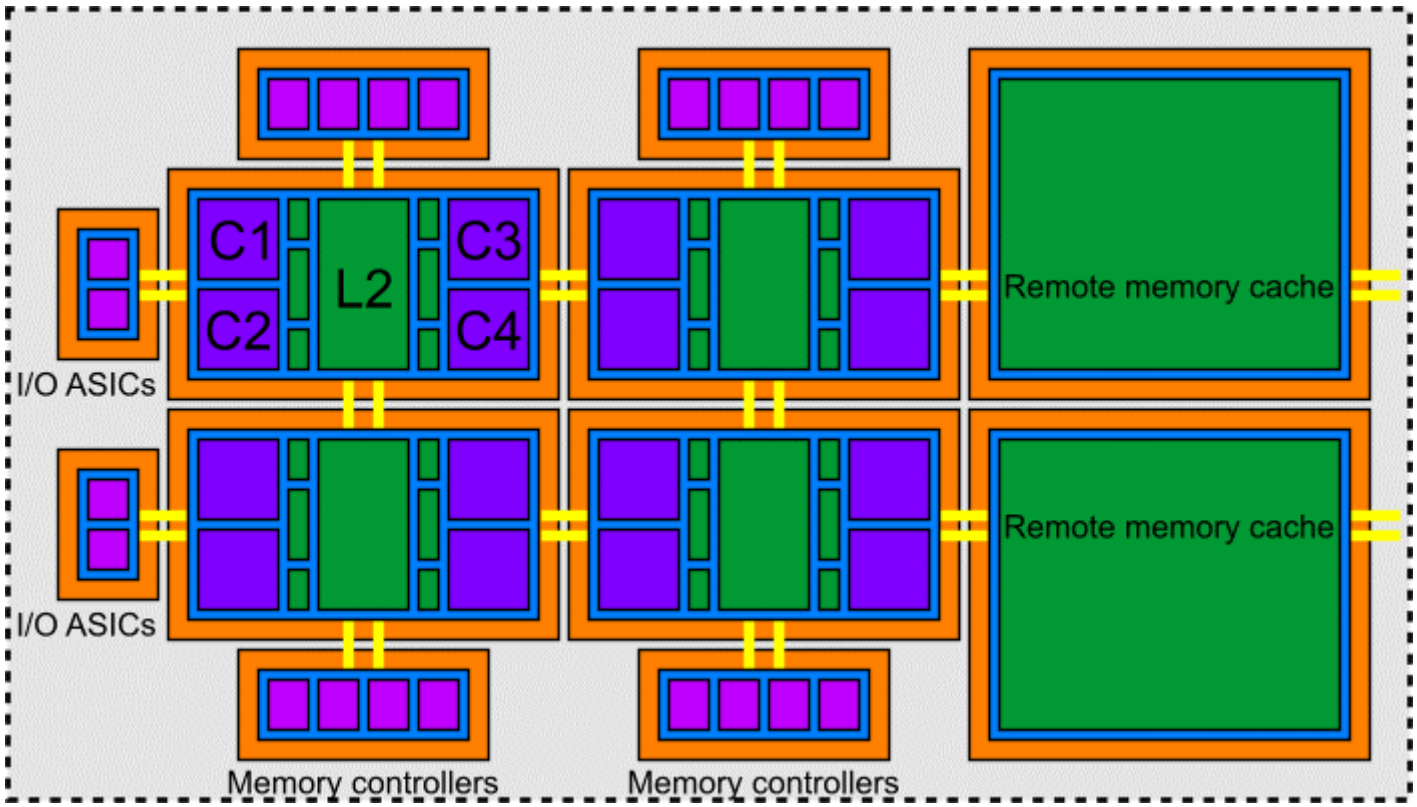
Most HPC programs have greater locality and so have far fewer remote memory requests, so scale much more easily. But it can require a huge amount of custom development to make the algorithms have that much locality. A big cost for many large HPC projects is not the hardware costs, but the software development costs. Putting it another way "big iron Vs clusters" is more like "expensive hardware Vs expensive software".

In general I have argued that highly TLP optimised processors will get more processing logic and less cache, pushing up memory bandwidth requirements very significantly. However, for large systems this is much less likely to be the case, since the bandwidth requirements between CPU/memory boards are already huge. Unless radical new technology is used, it will likely be hard to increase cross system bandwidth as much as per processor bandwidth. So it is much more likely that large caches will be used to reduce bandwidth requirements on large systems.

Today, most processors have an individual cache, on-die or directly attached. However for large systems it might be better to put a large cache in the chipset, basically dedicated to reducing remote memory requests. For local memory access the processors will be less latency sensitive and should have lots of bandwidth, so there is less need for local memory caching. That should also make it easier to design processors that are well suited to small systems and large systems.

So how big should these shared caches for remote data be? That depends on how much system bandwidth is available, how fast the processors are, the maximum number of processors, how scalable the system needs to be, and how upgradeable the system needs to be. The last point is particularly important, since large systems will often be used for at least 5 years in production, and customers are much more likely to upgrade them compared to low-end systems. Given the high development and testing costs, the longer the shelf life the better. One way to help this is to have a huge amount of system bandwidth at the start, reducing the need for large remote caches, but rapidly increase the cache sizes as faster processors are introduced. These large systems would also be more readily able to afford large caches, and I wouldn't be surprised to see 1 GByte caches by 2010.

As in the above examples, MCMs could be used to make manufacturing simpler, and the large caches could become part of this. Higher-end systems can afford more advanced packaging too, and maybe putting several high-speed processor chips per MCM and having just one such MCM per "CPU/memory board" would be the best option, as shown below.



Large MCM for High-End Servers

For business applications on large systems, main memory latency is the number one issue affecting scalability today. This is particularly hard to address on large systems, because latency is largely a matter of distance - bigger systems are more spread apart. Today's large systems generally scale very well, but are also very expensive. Large systems designed for processors with heavy TLP optimisation are unlikely to be any cheaper per processor (or CPU/memory board), but will likely be much faster.

If bandwidth demands can be kept low enough, good scalability should be easier to achieve, because the TLP optimisation will make them much less latency sensitive. It should also help a lot with I/O, which frequently becomes a performance bottleneck in large systems.

A large increase in bandwidth demands between CPU/memory boards will increase the need for a more radical alternative to transporting such data. Fortunately, since the systems will be less latency sensitive, this will probably become a bit easier.

The most obvious "alternative" today is the use high-speed custom designed fiber optic connections, which can offer huge amounts of bandwidth, and is already in use in some cluster interconnects. Fiber is also not affected by magnetic fields and other issues which can cause problems for high speed, high power electrical connections, which is important for mission critical systems.

It is possible that fiber could allow for much more flexible system designs - in SMP systems today, the connection between CPU/memory boards is a fixed "backplane". But if fiber was used instead (while keeping the protocols between CPU/memory boards the same), the connection would no longer have to be fixed. The difference between this and a traditional cluster is that from a software perspective this would seem like a large single SMP system. Having a flexible instead of a fixed SMP interconnect is not a few idea, but it hasn't been achieved yet.

HPC Systems and TLP

Despite all the press x86 clusters get these days for HPC solutions, most of the market (by revenue) is actually for large SMP systems. It is expected though that x86 clusters will overtake SMP systems in a year or two. Across the whole HPC market, there is no one size fits all ideal system architecture.

Some HPC programs are effectively limited by how many operations they can execute per second, typically floating point. Doubling the clock rate of the chips or doubling the number will nearly double actual real world performance. This category is the simplest to solve.

Then there are HPC programs where local main memory performance is the biggest limitation. These are well suited to "grids" or x86 clusters connected via cheap networking, and scale fairly easily.

Finally there are HPC programs where "network" performance is the main bottleneck - remote memory requests are common. Such programs are the hardest to scale and big SMP systems are well suited to this. But for problems where the largest SMP box is still too small, clusters of some kind still need to be used.

The above three categories are effectively different classes of hardware solutions - individual programs do not always fit neatly into one of the above three. Another way to look at it is the typical distance data travels to the CPU core: mostly a few millimeters from cache (first category), mostly several centimeters from local main memory (second category), and mostly a meter or more over the network (third category).

Some HPC projects also need serious amounts of I/O bandwidth (or total storage) or very high-end graphics systems.

For TLP optimised designs, the first category is well suited since transistor efficiency is the main issue. For the second category, a multi-core, multi-threaded system-on-chip design with a lot of local main memory bandwidth, little cache and high-speed networking (Ethernet or Infiniband) would make an almost ideal compute node.

Because TLP optimised processors focus more on efficiency and memory bandwidth, that should help HPC in general. Solving the third category is the hardest - the speed of light actually becomes a noticeable limitation for the largest HPC problems. Some companies are working on very densely packed groups of custom multi-core chips using exotic cooling - keeping the system small makes the networking easier.

Final Thoughts

A major side-show to the whole multi-core processor discussion is how software will be charged. Customers don't like this since it would likely make moving to dual-core more expensive. Naturally, the hardware vendors would prefer cheaper software. Some software vendors are moving away from a per-CPU or per-socket model anyway, and some already charge per-socket not per-CPU. For Sun's 8-core Niagara, this is a particularly big deal, but for their own software Sun already charges per-socket and provide software subscription options as well. Microsoft have announced per-socket pricing, so maybe this issue will not be a drag on the market moving to dual and multi-core.

In the x86 server market overall, changes will probably be fairly incremental in nature. This will make the transition easier for OEMs, ISVs and customers. However, they also have a transition to 64-bit software to navigate.

Meanwhile the main RISC/Unix vendors are already offering dual-core, and we should see much more radical designs shipping in a year or so, starting with Sun's Niagara processor. However, the short-term effect on the market may be modest - customers don't change buying patterns that quickly after all.

However, I'm not predicting any particular companies will be winners or losers based simply on the move to TLP optimised processors - there are many other factors, including time to market, execution, marketing and positioning, and for IBM and Sun, a host of system related aspects.

Regardless of what happens in the market, we have a lot of new technology to look forward to.

This is the end of my "TLP" series of articles. This final article covered a large number of topics, but only briefly. I may do some follow-up articles that take a more in-depth look on some of the sub-topics, depending on reader feedback.

Further Reading

- TLP and the Return of KISS - Chris Rijk, *Ace's Hardware*
- Niagara: A Torrent of Threads - Chris Rijk, *Ace's Hardware*
- Architecting the Future: Dr. Marc Tremblay - Brian Neal, *Ace's Hardware*
- SML2004-0062: Long wires and asynchronous control - Ron Ho, Jon Gainsley, Robert Drost, *Sun Labs*
- Sun's Niagara falls neatly into multithreaded place - Charlie Demerjian, *The Inquirer*
- POWER5: Second Generation Dual-Core Processor - Chris Rijk, *Ace's Hardware*
- IBM's POWER5: A Talk with Pratap Pattnaik - Jon Stokes, *Ars Technica*
- Sun CPU Roadmap (Rock Diagram, Page 26) - *Sun*
- Contrarian Minds: David Yen - Interview, *Sun*
- Robert Drost (DARPA Project) - Interview, *Sun*
- Cascade Project Homepage (DARPA Funded HPCS Program) - *Cray*
- IBM Wins DARPA Funding - *IBM*