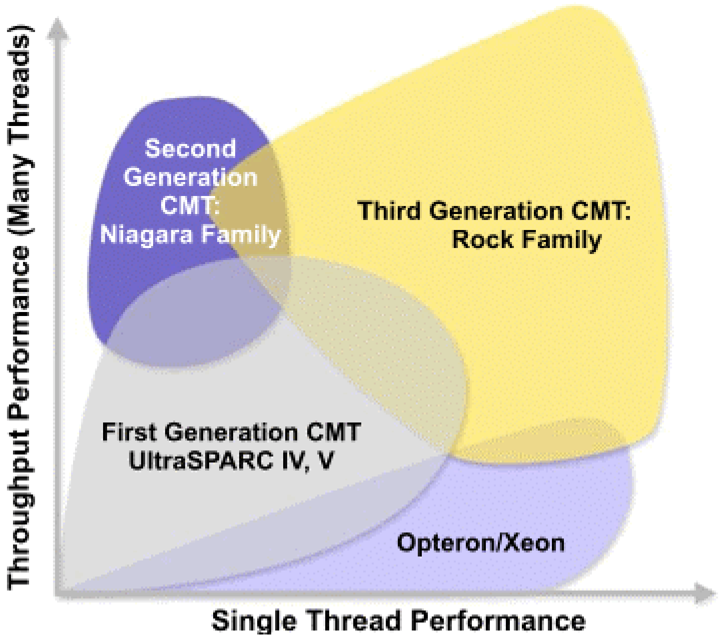
Niagara: A Torrent of Threads

By Chris Rijk – April 2004

Sun's "Throughput Computing" Plans for Servers

For the server market, Sun is effectively betting the farm on what they call "Throughput Computing" or Chip Multi-Threading (CMT), more traditionally known as Thread Level Parallelism, or TLP. They expect heavily TLP optimized designs to provide a significant competitive advantage. First up is Niagara in early 2006, which is the main focus of this article. The chip design is almost complete and is expected to tape-out in Q2 2004.


Throughput Performance (Many Threads)
Second Generation CMT: Niagara Family
Third Generation CMT: Rock Family
First Generation CMT UltraSPARC IV, V
Opteron/Xeon
Single Thread Performance

Until recently, Sun had three entirely separate CPU architectures in the works, the oldest being Millennium, a heavy-duty 6-issue design with out-of-order execution and 2 SMT threads. Despite taping-out sometime in Q1 2004, the design has been cancelled. On release it would have been called the UltraSPARC V. That name may now go to "Rock", a brand new multi-core design that promises 30x the performance of a 1.2GHz UltraSPARC III and 6x that of Millennium. Rock was due about 1 year after Millennium, before the latter was cancelled. The diagram on the left was taken from a Sun presentation showing how they contrast various CPUs (though I edited it slightly).

Several few years ago, Millennium was the planned successor to the "Cheetah" architecture, known commercially as UltraSPARC III. The last Cheetah based design seems likely to be Panther, a dual core version with a number of tweaks, which would have used the same 90nm process as Millennium, and been nearly as fast in server tasks. Between Panther, Niagara and Rock, it seems Millennium got squeezed out - the first casualty of Sun's change in direction on CPU design.

How TLP Could Affect the Server Industry

In TLP and the Return of KISS I explained how processors could be better optimized for multi-threaded workloads by using many CPU cores per chip and other techniques. This is a big deal as including software, storage, services and support, the overall server market is worth over $100Bn a year. IBM and Sun make most of their money from this market, so optimizing for server performance is more important to them than single threaded performance. Though the actual processor chip is only a tiny proportion of that $100Bn market, it makes up a significant proportion of R&D costs and has a big impact on competitiveness.

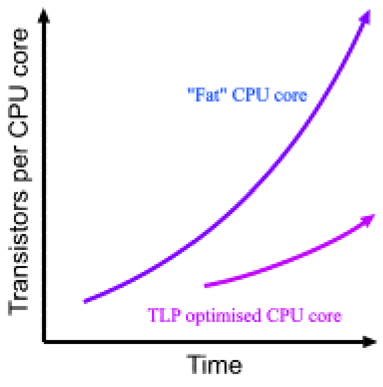
An important point about optimizing chips for server tasks with TLP is that customers will not need to change their software. Moving to TLP optimized systems will not be significantly different to previous generation processors. This is worth noting because software support and customer migration issues is what are dragging down the Itanium. Performance could still be dragged down by a poor OS threading model though.

Some server processors today have some simple techniques that improve multi-threaded performance but make little or no difference to single threaded performance. For example, the Xeon's Hyperthreading, and the dual-cores of the POWER4 and UltraSPARC IV. This could be described as "first generation chip-level TLP" - it is more like simple enhancements to existing designs. These enhancements will naturally get better over time and build on previous work. However, next generation designs with heavy TLP optimization will not be "more of the same", and will reflect a change in priorities - multi-threaded performance will become the primary goal for the entire design, even at the expensive of single threaded performance.

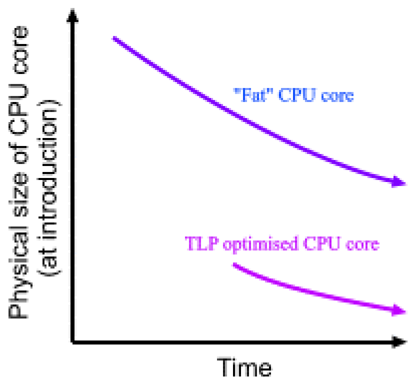
Today when a company introduces a "next generation design", like AMD's K7 to K8, or Intel's Pentium III to Pentium 4, the new design is faster in most or nearly all tasks. For companies who replace conventional "fat" designs with heavily TLP optimized designs, single-threaded performance will not necessarily improve and may even fall, while multi-threaded performance increases massively. This will take a bit of explaining to customers.

Estimated Industry Trends

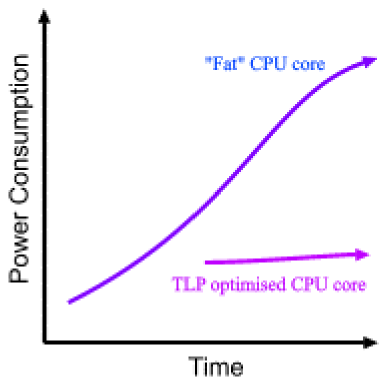
The following graphs and comments are some rough personal estimates of trends across the industry, for server CPUs. They are meant as a simple graphical guideline, which is why they are so basic.



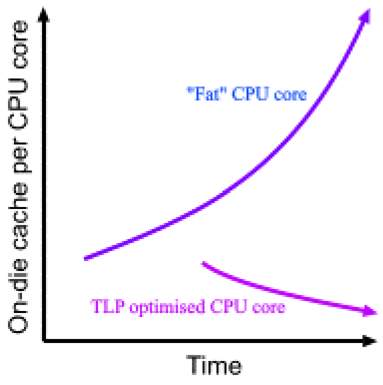
Moore's Law allows for the rapid increase in transistors per core. TLP optimized cores will start out much simpler, and may grow complex more slowly.



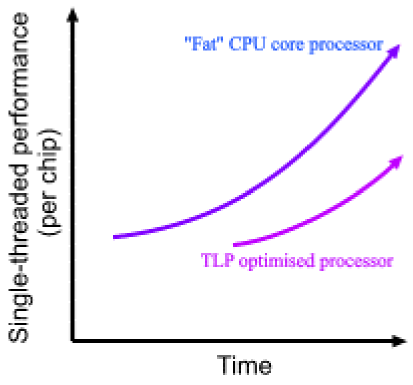
The trend is for chips and CPU cores to get smaller, though TLP optimized ones will start much smaller.



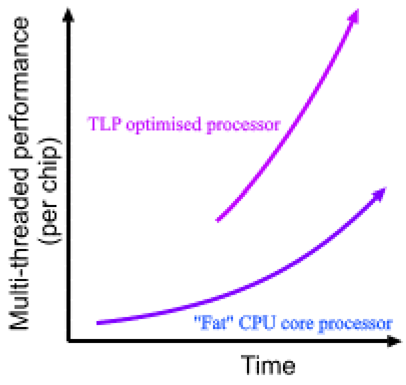
Growth rates in maximum power for "fat" CPUs have levelled off a bit. For "thin" cores, the number of CPU cores per chip will probably increase rather than the power consumption per core.



"Fat" cores need lots of cache to reduce memory latency. TLP optimized designs are less latency sensitive, so less cache is needed.



Better process technology helps both types to increase, though the simpler, slower clocked "thin" cores will be slower on more traditional benchmarks.

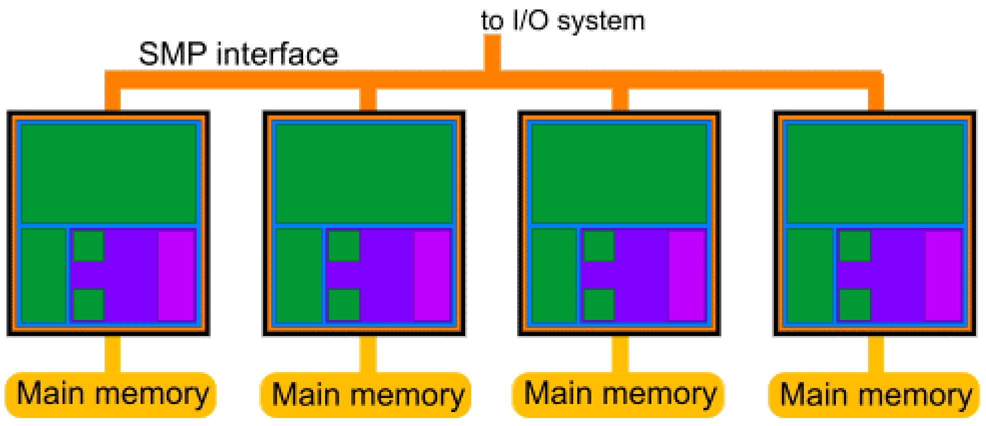


"Fat" cores will benefit from TLP techniques and general improvements, but not as much as "thin" cores.

As noted in the previous article in this series, smaller and simpler cores are more efficient, and using multiple cores scales performance better. This is the basic reason why heavily optimized TLP designs will see huge performance boosts, per chip. However, not only with these "thin" cores be lean and mean, using "lite" versions of existing design ideas, but they will make trade-offs towards multi-threaded code, and make use of new ideas that mostly or entirely benefit multi-threaded code. In other words, TLP optimized designs will not merely be simpler, but will take a different path towards design.

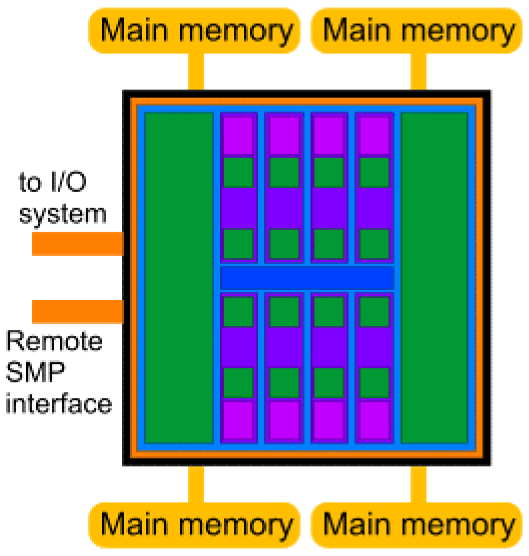
Effects on System Design and CPU Companies

System designs could look very different at the board level in a few years. Part of the reason for this is that how much main memory a server needs is roughly proportional to performance. In other words, if performance per chip increases by 4x, the main memory capacity will likely also have to increase by 4x. In addition, since heavily TLP optimized designs will likely have less cache and also more performance per chip, the bandwidth demands on the memory system will be far higher. Today, main memory bandwidth isn't much of a limitation for server benchmarks, but memory latency is - with heavily TLP optimized designs this situation should reverse.



An illustration of a 4-way system today. The only TLP comes from having multiple chips

A single core from a TLP optimized design could be used in bottom-end servers, "thin and light" portables, or bottom-end PCs, but not much else. Maybe Sun and IBM will stop doing more general purpose cores and concentrate on server and HPC workloads. These may stretch to very high-end workstation use though.



An illustration of a system with a heavily optimized TLP design.

AMD probably cannot afford to have two parallel CPU architectures. To justify the R&D expensive of a separate server-only CPU architecture, they would need to expand their sales of server CPUs very significantly. Until that happens, AMD will have to design cores that must offer next generation desktop performance while looking for opportunities to increase multi-threaded performance.

If Intel does "downsize" Itanium they could easily afford to do an all new TLP optimized CPU core, though that would take time. In the shorter term they could use multiple Pentium M cores though - they are small and power efficient. This would mean their "NetBurst" Pentium 4 design could ignore server workloads entirely, particularly as the increasingly longer pipelines are poorly suited to server tasks.

Intel's official roadmap today is to use a multi-core, TLP optimized Itanium code-named Tukwila in 2006-7, and for it to be competitive with x86 systems. This somewhat implies that Intel are not working on a heavily TLP optimized x86 design, though they likely have a secret back-up plan.

How well heavily TLP optimized cores will compete with conventional designs will depend both on their design and time to market, and also what their competitors are doing. A number of heavily TLP optimized designs are on the horizon, but details are mostly scarce. Fortunately, enough details are available on Niagara to write a decent article on it.

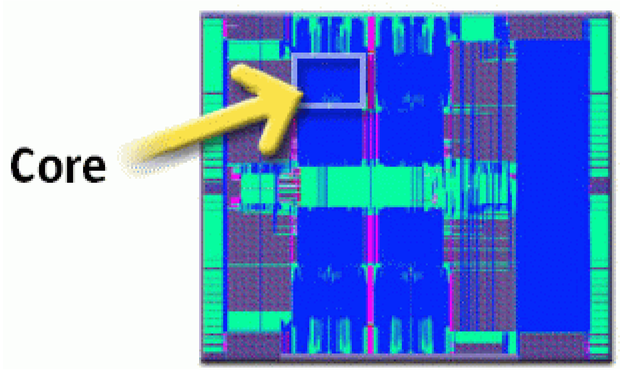
8 Cores and 32 Threads. Low-End?

In July 2002, Sun Microsystems bought Afara WebSystems for $28 million and brought their processor design team in-house. The result of this is "Niagara", a 90nm chip with 8 CPU cores, each of which has 4 threads. Niagara is part of Sun's "h" line of CPUs for horizontal scaling servers - lots of small boxes.

According to this EE Times article the die size is 340mm². According to other articles, each CPU core can issue just one instruction per cycle, in order. Execution of a single thread continues until a stall, upon which the pipeline switches to another thread without pause (so long as one of the 3 other threads is ready). Apart from it being UltraSPARC compatible, few other details have been given about the CPU core design.

Though Niagara is being used here to help explain heavily TLP optimized designs, its design would be more typical of multi-core chips in the embedded market. POWER6, Rock and Tukwila will likely all be super-scalar and use SMT, but we know next to nothing else at this stage. It might be better to think of Niagara as a high-end embedded system-on-a-chip, rather than a more general server processor. The target market for Niagara looks to be network heavy high-density servers with simpler workloads. The main competition would seem to be 2-way rackmount servers and blade servers.

A noteworthy feature of the chip is an integrated TCP/IP Offload Engine, or TOE. This can reduce the CPU overhead massively, while also improving network bandwidth utilization and latency. A rough yard-stick for TCP/IP processing is that a 1GHz Pentium 3 would be required to process 1Gbit/s of traffic. With 10Gbit Ethernet coming soon, the network-processing overhead is becoming a serious issue. However, using a TOE does require OS support - Sun's Solaris 10 is due in late 2004 and has a very significantly improved TCP/IP stack, including TOE support. Niagara seems to have on-die Ethernet controllers, so the TOE will likely be able to make a very significant difference on network intensive benchmarks. Security processing (triple-DES and RC4) is also included, so SSL should fly too.



Niagara floorplan, from presentation at Sun's 2003 analyst conference.

If the networking is interesting, the CPU core design looks rather bland at first glance, being single-issue. Even some embedded designs are super-scalar these days. Based on the floorplan on the right and the 340mm² die size, it looks like each CPU core is a mere 8mm² or so. The clock rate will not be impressive - not much faster than Sun's current 1.2GHz processors according to Sun officials. The grey blocks on the floor plan to the left and right of the CPU cores are probably the L2 cache. It is probably 2MB or 4MB but is unlikely to be very fast, to help keep down power consumption. Floating-point (and HPC) performance seems to be minimal. Sun has said little about the main memory system except to express their interest in DDR2 SDRAM. Niagara also does not have a SMP interface - 2-way Niagara systems will not be possible.

So where's the fun? Sun is currently projecting that Niagara will have 15x the server performance of their current server blade processor, the 650MHz UltraSPARC IIi. That would be competitive with 4-way servers using 130nm Opterons and Xeon MPs, and likely well above them in more network intensive benchmarks. By the time it's released, Niagara will probably be competing with high-end 2-way servers using 90nm processors.

More impressive though is that Niagara is targeted at server blades, which suggests a power consumption of between 30W and 70W . In comparison, top speed Opterons and Xeons in 2005 will likely be consuming 100W each or more. This means Niagara could have 50-75% lower power consumption compared to the competition - 2 x86 servers. This allows data-centers to save costs and pack systems more efficiently. Of course, there is a power consumption overhead from the rest of the system and comparing systems instead of processors is less impressive. Still, this should give Niagara a unique and compelling selling point.

Relative price/performance will probably be less impressive, and vary from benchmark to benchmark as well as depending on release date. With a large system on a chip design, the majority of the system cost will probably be

Niagara, so a 65nm shrink should help costs significantly. Niagara will also need a lot of main memory to achieve its estimated performance, though it would be in the same situation as its competitors. System cost will probably have to be competitive with higher-end 1U and 2U systems with 2 processors.

There is a lot we currently don't know about the design or how competitive actual systems will be. We don't know whether Sun's "15x" is a conservative estimate or a cherry picked best-case scenario. Niagara's design may change somewhat before release, and the release date would affect competitiveness significantly. How competitors will fare, including in TCP/IP optimization is also unknown. However, just about enough information is available about Niagara's design that some predictions about its architecture and operation can be estimated.

The main aim of this article is to explain the architecture and how it gets its performance. I have to make a couple of reasonable assumptions about the design to do this, but is better than nothing. Later on, comparisons are used as a yardstick, the focus is on trying to explain Niagara's probable design and how "fat" designs differ from TLP optimized designs. Some comparisons and market analysis are added at the end though; to help put the design in context.

Nasty Server Code

For the sake of argument, I will assume that Niagara will clock at 1.3GHz, precisely twice the clock rate of the current UltraSPARC IIi. This makes comparisons simpler and is within Sun's hints. 15x performance at 2x the clock rate gives the 8-core Niagara nearly 8x the performance, clock for clock! Or in other words, a single 1-way issue Niagara core and the 4-way issue UltraSPARC IIi would achieve a similar average number of instructions per clock (IPC).

Because the Niagara's maximum IPC per core is 1, that means the UltraSPARC IIi's average IPC in server benchmarks must be below 1, well below its maximum. Based on analysis of other CPUs, this is very realistic, even though the UltraSPARC IIi can execute up to 4 instructions per cycle. Even "brainiac" CPU designs with large caches struggle to achieve IPCs over 1 on server benchmarks, even when not limited by I/O.

There is surprisingly little public analysis of processors running server benchmarks - almost everything goes to single threaded benchmarks, primarily SPECint and SPECfp. This is a shame since a detailed low-level analysis of CPUs running server code is useful to try to explain what Niagara needs to be able to handle.

Still, some studies do exist, which helps to get a rough idea of how server code and code like SPECint is different from the CPU's perspective:

- On SPECint, the OS overhead is almost non-existent and well below 1%. On server code, the overhead is almost always above 10% of CPU usage, 30% being somewhat typical and on network intensive benchmarks it can go over 50%.
- The most active parts of SPECint type code are small, and most fit very well into a 64KB instruction cache. On server code, instruction cache misses are much more frequent because the amount of frequently run code is much larger.
- As a percentage of instructions executed, branches are more common than in SPECint, but might be a bit easier to predict on average. However, because much more code is in use on servers, hits on the branch history table cache will be less common. When the branch predictor does not have any history to make predictions with, it must rely on much less accurate static branch prediction.
- Server code makes heavy use of virtual function caches, dynamic libraries, complex data structures, locking, and little use of floating-point or multi-media instructions.
- All non-trivial programs benefit from a better cache and memory system though server code benefits a lot more than average from larger caches, and is much more latency limited. Bandwidth usage is generally quite light, but cache misses reduce performance by over 60% in most cases. Data loads make up about 20% of instructions executed.

The best study I've yet found of low level CPU statistics on server code SPECint is A Case Study of 3 Internet Benchmarks on 3 Superscalar Machines by IBM and the University of Texas.

How Niagara is Naturally Efficient

It's not hard for a 1-way CPU core to achieve an average IPC closer to its maximum as only instructions that take more than one cycle to complete prevent this – super-scalar CPUs are limited by how much Instruction Level Parallelism is available. On Niagara, the L1 caches are probably small and the latency would almost certainly be 1 cycle, so only a cache miss will cause a data stall. An instruction cache miss would also cause the pipeline to stall. Conditional branches and other control flow instructions can also cause stalls. There are also special instructions and situations that would cause stalls, but that's mostly an OS level issue and not really important here.

Some data processing instructions would require multiple cycles to execute, including some multi-media instructions as well as floating-point instructions and integer divide and multiply. On some CPUs, floating point add and multiply instructions would have a 1 cycle throughput, but Niagara is almost certainly not optimized to that level. Floating-point and similar instructions are very rare with server code though. I'm not including HPC as part of "server code" here though.

The L1 cache miss and conditional branches are likely the two main problems Niagara faces in keeping the average IPC close to the maximum. This is partly because they are very common (about 30-40% of executed server code are loads or branches) and partly because they are inherently hard to deal with. Server code does have many more dynamic library calls and virtual function calls, though for the sake of simplicity I will focus on conditional branches, which would have similar effects on the pipeline.

Niagara has a very simple mechanism to prevent execution stalls for these cases - it swaps to another thread the next cycle and the CPU continues execution. This is of course only possible so long as other threads are active on the system and scheduled by the OS on the particular CPU core. In addition, if all other threads on the same CPU core have stalled, then the whole pipeline will stall.

This is the most common way for in-order designs like Niagara to switch between threads but not the only way. Simultaneous Multi-Threading (SMT) is what the Pentium 4 and Xeons use - all threads on the core can be processed simultaneously, and aren't swapped (except by the OS). A third major alternative is to swap on every cycle, which is what Cray's MTA design does. I will leave an in-depth comparison between these styles for another time.

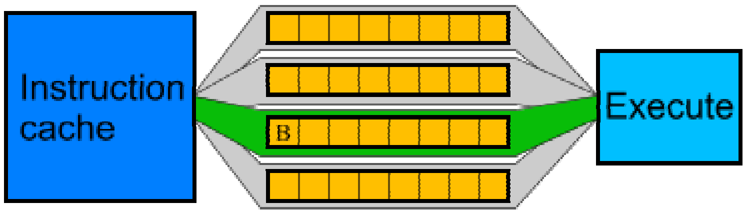
If 32 threads are active on Niagara, then each CPU core will have 4 threads allocated to it. This means that for the CPU core to stall, all 4 threads would have to have overlapping stalls. Reducing stalls efficiently requires balancing resources between the CPU cores and the rest of the system. Figuring how to keep the design simple and well optimized would require a lot of careful simulation to make sure this applies to server code in general.

The goal of the design would not be to try to achieve an average IPC of almost 1. However, the following sections will show how Niagara could come reasonably close, and therefore be competitive.

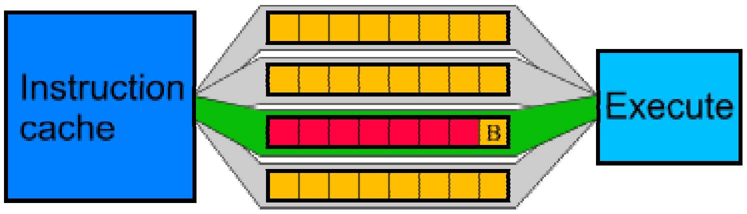
The Inefficiencies of Branch Prediction

For CPUs in general, at the start of the pipeline, instructions are read from the instruction cache, one after another, and fed onto instruction decoding. Those instructions will be executed a number of cycles later, depending on the length and design of the pipeline. However, some instructions change the "program counter", which is the address of the next instruction to load from the instruction cache, or the next one to execute, depending on the ISA. This means that the instruction fetch part of the pipeline must start fetching from the new location, instead of simply the next instruction.

How Niagara may deal with branches #1



Say the current executing thread has just loaded a conditional branch instruction. If all the other pipelines are full, then Niagara may as well use branch prediction to guess which instructions to load next, as shown by the reddish coloured block below:



With some instructions, such as conditional branches, the new value of the program counter will not be known until the instruction is executed, many cycles after that instruction is fetched. This means the instruction fetch unit does not know the next instruction to load when it encounters one of these instructions and would have to stall. Since this can waste a lot of performance, the instruction fetch unit will speculatively guess the next instructions to load when it encounters one of these special instructions. Guess wrong, and all that work is wasted.

As noted above, server code is much larger, making it harder to keep a good history for branch prediction or virtual function calls. Then of course there is the general problem of predicting these accurately. With multiple threads per CPU core, this becomes even harder on the branch history table's cache.

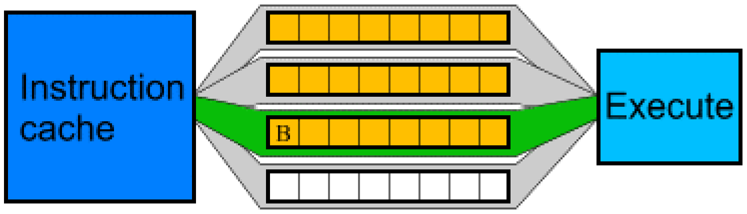
Niagara's branch predictor probably only has simple static branch prediction logic. If there are 4 threads on a core and all are ready to execute then when a branch is encountered, it doesn't hurt to use simple logic to guess which way the branch will go. If the guess is right, execution continues normally, but if the guess is wrong, then execution switches to another thread.

However, not all the other threads may be ready to execute. One may have no instructions prepared at all, due to an earlier branch miss, the thread just being allocated to the CPU or another reason. In such cases, rather than speculating on the current thread, it would be most efficient to start filling the pipeline for that thread. The thread with the branch continues until the branch is reached and then switches to another thread.

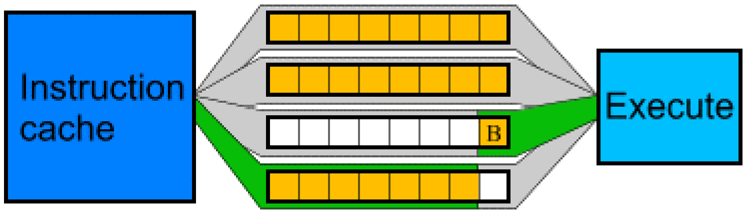
In other words, the CPU core focuses on doing real work - it is better to do something that is *guaranteed* to help performance rather than speculate on something that *may* help performance. Still, it is better to speculate than do nothing at all.

By making use of other threads, Niagara needs only very simple logic to help handle branches well. This has several benefits, including reduced design and test time, less logic, smaller chip size, lower power and a shorter pipeline. Not only that, but the cost of branches becomes very small, improving performance. This is an example of how TLP optimized designs can be naturally efficient.

How Niagara may deal with branches #2



If another pipeline is empty, it would be most efficient to start filling that one with real instructions, rather than the current one with speculatative instructions. In which case the currently executing pipeline will swap to another pipeline when the branch is actually executed. For simplicity, Niagara could use only this method, instead of using #1 above.



What About Branches on Super-scalar SMT Designs?

Though Niagara is the focus of this article, let's take a brief pause to look at how super-scalar multi-threaded CPU cores could compare.

Niagara probably has quite a short pipeline, and a branch miss may only cost 5 or 6 cycles. In addition, since Niagara executes only 1 instruction per cycle, that means only 5 or 6 instructions are lost in pre-fetching, decode and so on. On Intel's "Prescott" Pentium 4 E, a branch miss-prediction costs 30 cycles, and with 3 instructions per pipeline stage per cycle, that's an awful lot of instructions that could depend on one branch prediction. Clearly things are rather different here.

Prescott supports Hyperthreading as well, with 2 SMT threads. Instead of using branch prediction by default, should it switch focus to the other thread like Niagara? Obviously if the other thread is stalled on a memory request, there is nothing to gain by stalling the active thread when it encounters a branch. But what if both threads are active? Should the pipeline work on tens of instructions for one thread, only for it all to go down the drain? After all, the other thread could have used the same resources for its real work.

There is no simple answer to this, and results would depend on what benchmark you use. Instead, a smarter approach is probably best. Some branch prediction history tables have 4 states, about whether the branch will probably be taken or not, and whether this prediction is "strong" or "weak". A "weak" prediction means it's more likely to be wrong. So the smart approach would be: if two threads are active, only use branch prediction for branches that are strongly taken, or strongly not taken. So the less certain the branch predictor is, the better it is to focus on real work. This means single threaded code is not penalized, while multi-threaded code is efficient.

If a CPU core has 4 or more SMT threads, the dynamics change a bit again. If 3 or more threads are currently active on the CPU core, and not stalled, then it's probably best to use little or no branch prediction. Instead, focus all efforts on real work, and not speculation. If 1 or 2 threads are active, it would be best to do the same as the "smart" case above for 2 SMT threads.

L1 Caches and Data Stalls

In the previous section we saw how Niagara swapping to a different thread could help reduce the losses from branch prediction. With a short pipeline and 4 threads per core, branches by themselves would rarely cause the whole pipeline to stall, and certainly for not more than a few cycles. However, the subject of this section is on stalls due to memory latency, mostly data loads, which are a lot nastier to deal with.

Niagara has the same basic solution to this problem as with branches: swap to another thread to prevent pipeline stalls. With a 1-cycle L1 data cache, this would only have to happen on a L1 cache miss. An instruction cache miss would also cause a pipeline stall and on server code this can be very common compared to SPECcpu and data intensive benchmarks. The basic problem might be worse for Niagara since the instruction caches are probably pretty small to keep the individual cores sizes down and efficiency up. In addition, having multiple threads per core makes the L1 cache hit rate worse.

Either way, a L1 cache miss will stall the thread that did the request. If other threads are ready, one can be swapped in so that the CPU itself doesn't stall. How much a L1 cache miss will affect overall performance depends on L2 cache and main memory system. The L2 hit latency is probably high because of the extra complexity of having multiple cores and because it would be optimized for low power, high throughput and high density, at the expense of latency. The L2 cache is almost certainly shared between all the cores, meaning the L2 cache controller can't be put in an ideal position - the closer it is to the L1 cache the better. This is one case in which single core designs have a small advantage.

If the L2 cache hit rate was 100% the overall loss in performance from a L1 cache miss would probably be quite small. In short, with 4 threads per CPU core, it would be rare to have 4 overlapping pipeline stalls in the period of a L1 cache miss (L2 cache hit latency) of say 10-20 cycles. About 1 in 5 instructions would be a load, so even a cache miss rate as high as 25% would mean a L1 data miss every 20 instructions on average. Even factoring in instruction cache misses and branches, pipeline stalls in such a scenario should be quite rare, and would only happen when random chance brings several stalls close together.

Even with 4 threads and a small size, the L1 data cache miss rate would likely be 10% at most, not 25%. So, as long as the L2 cache hit rate is 100%, then with 4 active threads each CPU would likely achieve an average IPC of about 0.95, compared to the maximum of 1. This is just a guess, but seems reasonable. With 100% L2 cache hit rates, most "fat" CPUs would get an IPC of about 1.5-2.0, which is better than Niagara but about 50% of their maximum.

However, due to main memory latency, complete cache misses on "fat" CPU cores often result in an IPC reduction of 50% or more. Putting it another way, main memory latency reduces server performance by 50% or more. Niagara probably doesn't have a L3 cache, so let's see how it could fare when there is a L2 cache miss.

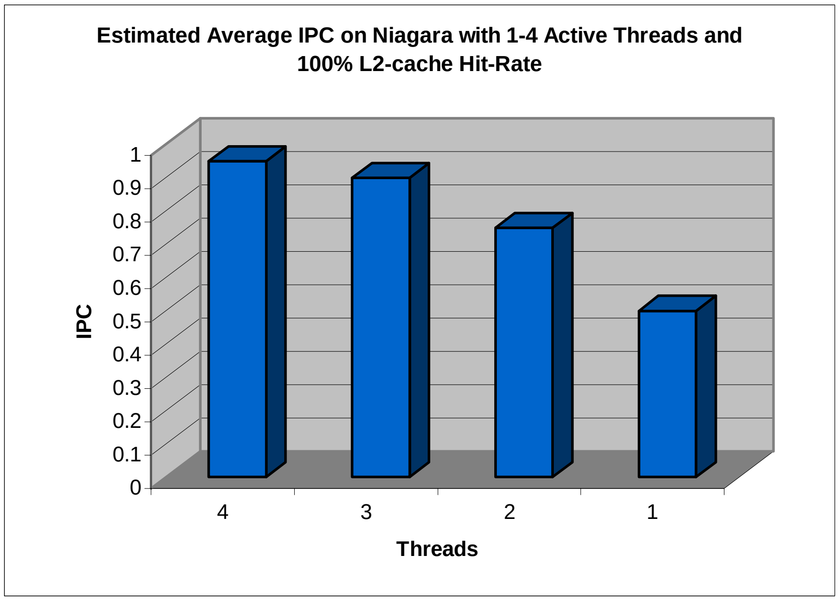
The Effect of L2 Cache Misses on Niagara

Before analyzing how a L2 cache miss could affect Niagara, let's simply consider the likely IPC with 100% L2 cache hit rates for 1, 2, 3 and 4 active threads. This is shown in the simple graph on the right below, which assumes that cache hit rates are the same regardless of how many threads are active. This is just as a yardstick of course, since actual results would depend on the pipeline length, L1 cache hit rates, L2 latency and the benchmark being used.

With 4 active threads, pipeline stalls would be very rare as thread stalls due to L1 cache misses and branches would be uncommon enough that 4 overlapping stalls would be very rare. In comparison, in the case with just 1 active thread any L1 cache miss or branch miss-prediction would cause a stall.

This is useful for estimating the effect of L2 cache misses, since a single L2 cache miss would stall one thread for many cycles, during which the IPC would be like the example in the graph with 3 active threads. So if 50% of the time there are no stalls due to L2 cache misses, and 50% of the time there is just one L2 cache miss, then the average IPC becomes: `(0.5 x 0.95) + (0.5 x 0.90) = 0.925`.

L2 cache misses can overlap however, particularly since one takes so long to complete. So for a more realistic result, let's assume that the percentage of time with 0, 1, 2, 3 and 4 overlapping L2 cache misses is 16%, 45%, 25%, 11% and 3% respectively. The figures were deliberately chosen here so that on average 1.4 threads are stalled at any one time due to L2 cache misses, which I think would be more of a worst case. The average IPC comes out as: `(0.16 x 0.95) + (0.45 x 0.90) + (0.25 x 0.75) + (0.11 x 0.50) + (0.03 x 0) = 0.7995`. If Niagara really could achieve that in real world benchmarks, that's nearly 80% of the maximum IPC, far better than conventional designs.



Of course this is just a "back of the envelope" level guide to help understand how Niagara could perform. There are other factors, which will be discussed shortly.

Niagara's L2 Cache Hit Rate

Ignoring L3 caches for the sake of simplicity, a L2 cache of about 4-8MBytes is enough to get near perfect L2 cache hit rates on most of the SPECint sub-benchmarks. Cache hit rates on server benchmarks are much worse, and a L2 cache miss every few hundred instructions would be common. That means a thread would run for maybe 100-200 cycles before getting another L2 cache miss. On server code, most CPUs spend more time waiting on cache misses than processing data. In theory, SMT should be able to help a lot, but with two active threads per CPU core the cache hit rate goes down.

The above back of the envelope calculations assumed that 1.4 threads would be stalled on average at any one time. So for example, if a load with a L2 cache miss took 140 cycles on average to complete, then a cache miss every 100 cycles would mean 1.4 threads were stalled on average. This is considerably more frequent than a large cache "fat" design, but even so, the effect on Niagara's IPC is much smaller. If the other figures are reasonable this suggests Niagara can tolerate even poor cache miss rates. In practice though, the L2 cache miss latency should be better than 140 cycles as Niagara likely has on-die memory controllers.

In Niagara's case there are 32 active threads, which makes predicting the L2 cache hit rate quite difficult, and we don't even know the exact size. Servers programs basically respond to a request with some data. When a request is being processed, data for the request itself, temporary data and the final response output will be unique and private to the thread handling the request. This request specific data will be referred to frequently and generally have a good cache hit rate. How much is needed is highly dependant on the complexity and scope of the request, and the implementation and configuration of the server software. Both the request and response also need to pass through the OS, which adds some overhead.

The other major memory component of a server process is "global" (or "shared") memory within the server. This would typically be for an in-memory cache of storage data for a database, or in general, data that multiple threads would need access to. Of course, if some required data is not currently cached, the data will need to be fetched, either from the storage system or another server process. For a particular benchmark, if the CPU has enough cache that the request specific data fits in very easily, then more of the cache can be used for global data. This helps the overall cache hit rate, and performance of course.

On Niagara, 32 threads will be constantly active in benchmark conditions, so the average size (and usage profile) of the memory needed to process a request becomes important. If the average size was just 10KBytes, or 320KBytes for all 32, then most of Niagara's cache could be used for global data. If the average request size was 100KBytes, or 3200KBytes in total, little global data would be cached.

Though it is hard to predict Niagara's L2 cache hit rate, since the design has quite a large L2 cache, it is natural to assume the designers thought it was important. Despite Niagara's ability to soak up memory latency, it's certainly not immune. One obvious potential problem is for the large numbers of threads to cause the cache to "thrash" - conflicts over the same cache lines with many active threads could severely impact the cache hit rate and hurt performance.

Another factor is memory bandwidth. With 8 CPU cores averaging a L2 cache miss perhaps every 100ns each, the cumulative demands on main memory bandwidth could be crippling. Each core getting a L2 cache miss every 100ns is 12.5ns per chip, or 80 million per second. If the L2 cache line size is 64-bytes, that'd require just over 5GByte/s of actual bandwidth. If PC3200 speed memory was used (200MHz DDR2 SDRAM), that would mean 4 64-bit channels would be required to meet the bandwidth needs, since DDR SDRAM bandwidth efficiency would be around 50%. So though a larger L2 cache would add to the chip manufacturing costs, it would help reduce the system costs by reducing bandwidth demands.

Other Performance Factors

Threads do not simply run forever with only a pause for main memory. To access unique OS resources (like I/O) or to ensure safe access and updating of shared data, locks are needed. Most are very short lived but still can block a thread for much longer than a L2 cache miss. On Niagara, all locking is performed inside the chip since it is 1-way only. Communicating between different chips is much slower than inside a single chip, and multiple completely separate caches can lead to dirty cache misses and other performance penalties. By being a non SMP design, Niagara can be better optimized, though this does limit its market potential. On the other hand, many of Niagara's main target markets do not need large multi-processor systems anyway - Rock will be designed for them instead.

Swapping threads due to locks, or having more active threads than there are CPUs (and hardware threads) to run on them does suck up a bit of performance too. Exactly how much is hard to say, though the type of environment Niagara would typically run in would have a large number of threads processing simple requests. On fat CPUs this can be a bit of a problem, particularly if the cache is small, as the previous thread is likely to have flushed most of the cache the newly scheduled thread was using when it was last running. So when a thread starts running again, the cache hit rate would be particularly low. With 32 hardware threads on Niagara, each relatively slow compared to a "fat" CPU, thread swaps would be much less frequent. Instead, Niagara has the problem of many simultaneously active threads.

Niagara might have on-chip Ethernet controllers as well as an on-chip TCP/IP Offload Engine. For network intensive applications this could help performance in several ways. Firstly communication between the Ethernet controllers and the CPUs would be internal to the chip, and would not require using system bandwidth. With on-chip buffers, even I/O buffers in main memory could possibly be eliminated (or at least reduced) which would help reduce the burden on main memory bandwidth, as well as improving latency. Niagara also seems to have built-in SSL acceleration, which helps reduce CPU load and improve overall performance. It would be interesting if Niagara has hardware GZIP acceleration too, as dynamically compressing HTML pages for browsers who say they support it can achieve 10:1 to 20:1 compression ratios. This helps reduces loading time, improving the customer experience as well as reducing bandwidth requirements - which is why we use it at Ace's Hardware.

The TOE would provide the main performance benefit though. Processing TCP/IP packets is quite CPU intensive once data rates climb to the gigabit levels. In fact, on network intensive benchmarks, it may take up over half the processor time. So offloading this work onto hardware can significantly reduce CPU load as well as helping saturate all available network bandwidth. In network intensive benchmarks Niagara would likely be limited by the available network bandwidth. Out of the box, Niagara systems would likely have huge network bandwidth performance and low latency because of the faster, more efficient, processing.

Niagara vs Low-End Competition

Ignoring the TCP/IP acceleration and other features, it does not seem unreasonable to expect Niagara to get an average IPC of 0.7-0.8 per CPU core. On that basis, the "15x faster than UltraSPARC IIi" does seem achievable for server benchmarks in general. However, the point of the analysis on the last few pages is not really to estimate Niagara's performance but to show how it can be so efficient. I have had to make a lot of assumptions and broad estimates after all, and could be missing some major performance limiters. Actual performance depends on specifics, and that can change at any time until the products are available.

Still, it is interesting to consider how it could compare with the primary alternative in 2005, 2-way rackmount systems based on Intel's 90nm Xeon DP processors. Other alternatives will be Opteron systems and Sun's own 90nm chips - the successor to the UltraSPARC IIIi is said to have twice the performance, and has 4MBytes of L2 cache. IBM may have some low-end POWER5 based systems too. But Xeons have about 90% of the 2-way market and will be the market-leading alternative.

In 2005, the Xeons will probably be available at 4GHz and up to 5GHz. Xeon's main performance limitation for server benchmarks is main memory latency, and this doesn't seem set to improve much. Though faster FSBs and DDR2 SDRAM are on the horizon, DDR2 SDRAM has poorer latency at the same clock rate. The overall latency improvement will likely be much less than the clock rate improvement, which will hurt IPC on balance. The much longer pipelines of the 90nm Xeons will particularly hurt server benchmarks. All the 90nm Xeon DPs may well have L3 caches to keep up the IPC. IPC on Xeons today are already low - below 0.5 RISC-like micro-ops per cycle seems the norm. Comparing such a value with IPC values on SPARC is better than nothing, but it is hard to say if this is reasonably accurate.

A server with a single 1.3GHz Niagara and IPCs of 0.7-0.8 per core gives 7.28-8.32 billion instructions per second (BIPS). Late in 2005 with large L3 caches and system improvements, a 5GHz Xeon might have typical IPCs of 0.4-0.5, which gives 4-5 BIPS. That's still well below Niagara. 2-way Opterons would likely be close though.

However, this is without taking into account Niagara's on-die network acceleration capabilities, and what system improvements the others get. It's also not known if Intel or AMD will have dual-core x86 server chips for the 2-way market. Tese dual-core designs are expected, but may be for 4-way and higher only, due to the higher costs of making them, and low volume. Either way, the quicker Sun can get Niagara to market, the better it'll be, relative to the competition.<

Niagara in the Market

Before anyone thinks I'm predicting the imminent demise of Xeon, a reality check is in order. Firstly, Niagara will only achieve top speed on network and thread intensive workloads. This is hardly a small market, but is perhaps around 30% of the overall 1-2 way server market. The low single thread performance may also put customers off Niagara systems when they test it internally - only a heavy duty load generator would be able to push Niagara to the limits. In an ironic way though, that aspect could encourage developers to optimize code more since inefficient code and set-ups will be more noticeable in development and testing.

System price is also a critical factor. Niagara is a big chip and will likely be more expensive to manufacture than two Xeons. The development costs also have to be spread among lower volume products - the core of the Xeons are also used in high volume desktops. However, Sun is a systems company - Niagara doesn't need to be profitable by itself, and profitability will depend on Niagara server revenue, and services, support, software and storage for those servers.

However, even if Niagara systems had twice the price/performance and much better power efficiency, this will not cause a rapid change in market share for the rack-mount server market. The basic reason is that customers don't change sides quickly. Customers also seem more reactive (in the short term at least) to lower prices, compared to performance or price/performance.

Most of Niagara's early sales will come from existing Sun customers, and Niagara is well suited to the telco market, which is a strong market for Sun. However, a large jump in price/performance could mean that customers could buy fewer systems instead, which actually hurts sales in the short term. In practice though, most customers buy more systems in such cases. However, simply selling a Niagara system instead of (for example) an UltraSPARC IIIi system isn't really a win for Sun.

Niagara is a different sort of beast to what Sun customers buy today, even if many uses will be the same. It is not simply "more of the same", unlike the 90nm version of the UltraSPARC IIIi would be to UltraSPARC IIIi customers today. Because of this, customers will be unlikely buy in large numbers until they have a good feel of the performance. In contrast, Sun's UltraSPARC IIIi systems were slightly slower than the older UltraSPARC III equivalents, but about half the price, and quickly went from no volume to over 50,000 servers per quarter. How system costs compare with Sun's 90nm UltraSPARC IIIi successor will likely be important.

In general however, customers do not change vendors quickly or frequently. They also go for "more of the same" and "path of least resistance" most of the time. This is particularly the case for large customers, who also make up most of Sun's customers. The only major exception is the HPC market, where price/performance is top priority. In the business server market, change happens slowly. If the market was a real commodity, Dell's lower prices would have gained it market share much faster. Better price/performance does not cause customers to switch quickly in the short term, but if a noticeable performance or price/performance lead can be sustained the effect over several years can be substantial.

Doesn't this mean Niagara might not be very profitable for Sun? In the shorter term, if Niagara does prove to be a naturally more efficient system, this would enable them to have higher margins. In addition, Niagara is quite simple, so the R&D costs would be small compared to "fat" CPU core designs. Being 1-way only also makes the system design much simpler. Selling 50,000 systems might be enough for Sun to break even on the project, ignoring fewer UltraSPARC IIIi sales.

I think Sun will have to go to greater lengths than normal to educate customers about Niagara systems - because they're somewhat different. Issuing some common server benchmarks alone will not convince customers and if they don't understand the benefits, they won't buy. Today, Sun have a number of reference architectures for common applications - pre-developed solutions to common setups basically. If Sun provided a large number of Niagara based reference architectures at launch, and included some benchmarks of the capabilities of such systems in practice (from beta testers) that could make for a convincing argument, with the help of an advertising campaign.

In the longer term, if Sun can keep the system very competitive with second generation designs, then they will be able to win over more customers, improving volume and market share. This means, future Niagara based systems could be much more profitable, and it does seem that something like Niagara will be a long term feature of Sun's low-end servers - there has already been mention of second generation designs. A 65nm shrink would also allow for more aggressive pricing, helping to expand the market, and a second generation design could improve efficiency still further.

I think the current Niagara design represents just a small part of what is possible with TLP optimized designs. In that light, let's take a super-speculative look into what next-generation Niagara-based processors could be like.

Niagara in the Future

The likely plan for next generation processors based on the current 90nm Niagara design is "more of the same", rather than a radical re-design. So it may stay as a design with no SMP interface, and the "system on a chip" network-optimized style will likely continue too. Sun's multi-core Rock design looks to fit above Niagara's target market. So will Sun have two parallel UltraSPARC design teams off into the future? It seems possible due to the relatively lower development cost, though maybe future Rock-based CPU cores will be used in a Niagara-like system. In the meantime, perhaps Sun is willing to develop multiple TLP optimized core designs in parallel to speed overall development of the concept.

Compared to the initial Niagara systems, there are some opportunities for broadening the target market. Perhaps around HPC as the current Niagara design seems to have no regard for floating-point intensive workloads. Given the (likely) huge system bandwidth of Niagara systems, this could be a winner. There's even been some round-about hints of this in this EE Times article.

> *Sun has been more public about its plans to use the asynchronous technology that's been under development for several years in its labs. "We are trying to push in the direction of as asynchronous a machine as possible-we think that's where systems design will go," Gustafson said. "That lets us run everything as fast as possible while saving on power consumption."*
>
> *The resulting architecture presents the user with a single system image even though it employs multiple processors, each with about the same level of parallelism as Sun's upcoming Niagara processor. Niagara will sport eight cores, each executing up to four threads. Nevertheless, Gustafson said the underlying processor architecture is relatively unimportant in the overall design.*

A very simple CPU core design with an efficient FPU is similar to IBM's Cell/BlueGene plans. One major problem with this though is the power consumption - floating point units are typically the hottest and most power hungry parts of CPUs.

Making a mostly or fully asynchronous processor is damn hard, so doing an asynchronous version of a very simple synchronous CPU core (like Niagara) would be far more practical. Asynchronous designs are something for the long term future though.

Maybe Sun's petaflop HPC system will use an n^{th} generation Niagara design. For the sake of this discussion, it doesn't matter either way - this is just an example of how Niagara's focus could be expanded a bit. Let's look at some possible features that could be used to improve efficiency of general server performance.

More Advanced CPU Cores vs More CPU Cores

As noted in part 1, TLP optimized designs can improve performance by adding more CPU cores, or by improving the CPU core designs, as well as by tweaking the cache and memory system. Sun could simply double the number of CPU cores, and the size of the L2 cache, and roughly double performance (though the I/O and main memory system would also need improvements). This would double the size of the chip, so it would only be possible at 65nm and below. The basic point however, is that the die size increase is similar to the performance increase. So there is not a lot of point in making a more advanced CPU core unless the relative performance increase exceeds the die size increase.

This means the CPU designers must be very focused on efficiency. So how can a basic CPU core design like Niagara's be improved efficiently, without requiring a complete re-design? In this section, I'll discuss some possible ideas to achieve just this, which I thought of while working on my TLP articles. I've no idea if they're any good, but they seem quite practical and simple to me.

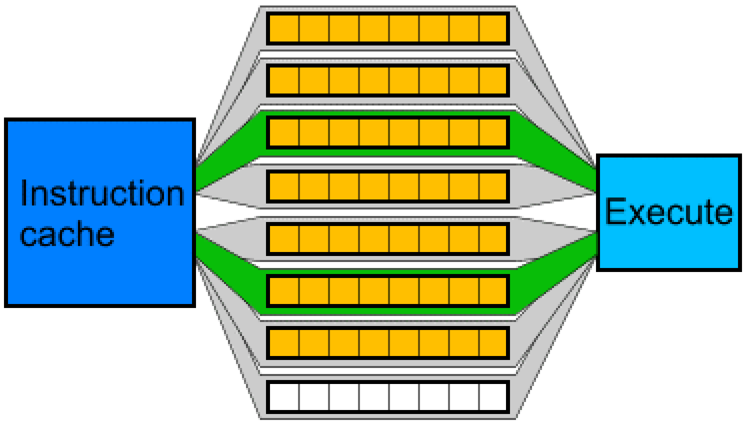
Super-Scalar, But Not as You Know It

Super-scalar processor design is an old idea. The basic idea is simple - a single CPU has multiple function units, so why not use them in parallel? E.g., do an "add" in the same cycle as a "load" or some other instruction. In its most basic form, 2-way in-order super-scalar, this requires four main changes to work. Firstly, the whole pipeline must be able to process 2 instructions per cycle. Secondly, the CPU registers must be able to handle reads and writes from 2 instructions per cycle. Thirdly, some logic is needed to determine what functional units are available for parallel processing. Finally, the instruction issue part of the pipeline must be able to extract ILP from the instruction stream so that issuing two instructions in parallel does not cause processing to change.

However, extracting much more ILP than this from the instruction stream is very inefficient, which is why most high-performance CPUs are 3-way or 4-way. But Niagara doesn't have to follow the same old pattern as Niagara is explicitly designed to process multiple threads. Instructions from different threads can always be issued in parallel, so long as they use different functional units. So a future Niagara based design could issue 2 instructions per cycle, from 2 different threads. In other words, the maximum IPC of a single thread will not exceed 1, but the IPC per CPU core will now be a maximum of 2.

This will require a double-width pipeline to sustain 2 instructions per cycle and some logic to determine what functional units are available for parallel processing. This might require adding a stage or two to the existing pipeline. However, it will not require logic to find available ILP or to increase the number of register ports. So not only is it easier to issue multiple instructions per cycle this way compared to a traditional super-scalar design, but less logic is required.

There is however, one new aspect, which requires a design trade-off. With 4 threads and up to 2 instructions per cycle, the instruction issue logic could either look for 2 instructions to issue from any 2 of the 8 threads, or 2 specific threads. The former will help maximize IPC, while the later is simpler. At a guess, I think there would not be much performance difference between the two, so the simpler solution would be best.



A future Niagara based design?

The easiest way to implement this would be to simply duplicate the front-end of the pipeline, with some minor changes at instruction fetch and the logic actually issuing the instructions to the execution units. The current Niagara design has one active thread at a time, and switches between them on stalls. This possible dual-pipeline Niagara design would still have 1 active thread per pipeline, and switch in the same way. So the instruction issue logic would look at the next instruction from the active thread from each of the two pipelines and issue both if there are no resource conflicts. If each pipeline has 4 threads, that gives 8 threads per core, which would also help the average IPC, though also puts more pressure on the cache system.

Taking this Idea Further

I don't think it would be practical to double this again - 4 pipelines of 4 threads, with a maximum IPC of 1 per thread and 4 per core. I think 4 threads per pipeline would be required to minimize a single pipeline stalling, but 16 threads per CPU core would probably be too much for the cache hit-rate. There might well also be issues just having that many active threads. It would also probably be more efficient to maximize 8 threads per core.

It would certainly be interesting to be able to issue 4 instructions per cycle from 4 different threads though. With 8 active threads, it would be reasonably likely that 4 of the next instructions from each would not have resource conflicts and could be issued in parallel. In practice of course, 8 threads would rarely be active at once due to stalls. The extra logic to handle this would also likely require another pipeline stage or two. So branches get worse again. The goal is to improve the average IPC per core though, and with this extended idea, averaging more than 2.0 might well be possible.

One downside is the increasingly complex and duplicated pipeline - instruction fetch, decoding and handling of branches and similar would have to be duplicated. The extra "fat" also adds to design time (though much would be duplicated), die size and power consumption.

There doesn't seem to be any way of entirely preventing this, but maybe the problem can be reduced by what I have started thinking of as "pipeline caching". The idea is similar to the trace cache on the Pentium 4. Instead of caching the raw data from main memory (or L2 cache) in the instruction cache, put some instruction decoding logic in front of it, caching the processed versions in the instruction cache. The decoding doesn't have to be parallel now because it is re-used. This also shortens the active part of the pipeline.

The obvious downside is the increase in the instruction cache size. On the Pentium 4, this is quite painful, and takes up about 4x more cache than the raw data. However, the RISC instructions on UltraSPARC are simpler and do not need to be decomposed into multiple micro-ops. The overall effect on die-size may be neutral, with the reduction in logic being offset by the increase in the instruction cache size. However, logic consumes more power than SRAM and is more likely to cause chips to fail. The shorter pipeline would also help improve performance.

It will be interesting to see how quickly advances in efficiency combine with Moore's Law to improve performance. However, each increase in the performance per CPU requires more memory bandwidth. This could put a severe strain on the bandwidth requirements for future Niagara based designs, as well as heavily TLP optimized designs in general.

Wrap Up and Look-Ahead

Without having much detail on the architecture, it is hard to make good predictions on Niagara without having to make lots of assumptions. So the main goal of this article has been to show how Niagara might work and how performance might turn out due to the design decisions. Fortunately Niagara's simplicity leaves less to assume. But, as always, real world performance, competitiveness and customer acceptance will have to wait for the real release.

In the meantime, hopefully this analysis will have helped give some insight into how TLP optimized processors could be designed, and the different sorts of problems they have to deal with. Looking into the future, there does seem to be a lot of potential to improve efficiency further, with some possibilities given on this page. The final article in this series will expand on this significantly, and cover a lot of aspects of processor and system design.