Scali's OpenBlog™
Programming, graphics, hardware, maths, and that sort of thing

Multi-core and multi-threading performance (the multi-core myth?)

Posted on June 1, 2012

Today I read a new article on Anandtech, discussing the Bulldozer architecture in detail, focusing on where it performs well, and more importantly, where it does not, and why. There weren't too many surprises in the article, as I mentioned most problematic areas (decoding, integer ALU configuration, shared FPU, cache configuration...) years ago already. But that is not really what I want to talk about today. What I want to focus on is multi-core and multi-threading performance in general.

The thing is, time and time again I see people recommending a CPU with more cores for applications that require more threads. As Johan de Gelas points out in the aforementioned article, it is not that simple. Although the Bulldozer-based Opterons have 16 cores, they often have trouble keeping up with the older 12 core Magny Cours-based Opterons.

Now, it seems there is a common misconception. Perhaps it is because the younger generations have grown up with multi-core processors only. At any rate, let me point out the following:

> *In order to use multiple cores at the same time, multiple threads are required.* ***The inverse is not true!***

That is: a single core is not limited to running a single thread.

First things first

Let me explain the basics of threading first. A *thread* is essentially a single sequence of instructions. A *process* (a running instance of a program) consists of one or more threads. A processor *core* is a unit capable of processing a sequence of instructions. So there is a direct relation between threads and cores. For the OS, a thread is a unit of workload which can be scheduled to execute on a single core.

This scheduling appears to get overlooked by many people. Obviously threads and multitasking have been around far longer than multi-core systems (and for the sake of this article, we can place systems with multiple single-core CPUs and CPUs with multiple cores in the same category). Since very early on, sharing the resources of the CPU between multiple users, multiple programs, or multiple parts of a program (threads), has been a very important feature.

In order to make a single core able to run multiple threads, a form of time-division multiplexing was used. To simplify things a bit: the OS sets up a timer which interrupts the system at a fixed interval. A single interval is known as a *time slice*. Everytime this interrupt occurs, the OS runs the scheduling routine, which picks the next thread that is due to be executed. The *context* of the core is then switched from the currently running thread to the new thread, and execution continues.

Since these timeslices are usually very short (in the order of 10-20 ms, depending on the exact OS and configuration), as a user you generally don't even notice the switches. For example, if you play an mp3 file, the

CPU has to decode the audio in small blocks, and send them to the sound card. The sound card will signal when it is done playing, and this will trigger the mp3 player to load new blocks from the mp3 file, decode them, and send them to the sound card. However, even a single-core CPU has no problem playing an mp3 in the background while you continue work in other applications. Your music will not skip, and your applications will run about as well as when no music is playing.

On a modern system that is no surprise, as playing an mp3 takes < 1% CPU time, so its impact is negligible. However, if we go further back in time, we can see just how well this scheme really works. For example, playing an mp3 in the background worked well even in the early days of Windows 95 and Pentium processors. An mp3 would easily take 20-30% CPU time to decode. But since the OS scheduler did its job well enough, nearly all of the remaining 70-80% were available to the rest of the system. So most applications still worked fine. Things like web browsers or word processors don't need all that much CPU time. They just need a bit of CPU time at the right moment, so that they are responsive to keystrokes, mouseclicks and such. And if the OS scheduler does a good enough job, then the response time is only one or two timeslices, so in the range of 20-40 ms. This is fast enough for people not to notice a delay.

Or, let's go back even further... The Commodore Amiga was the first home/personal computer with a multitasking OS, back in 1985. It only had a 7 MHz Motorola 68000 processor. But look at how well multitasking worked, even on such an old and slow machine (from about 3:52 on, and again at 7:07):

As you can see, even such a modest system can handle multiple heavy applications at the same time. Even though computers have multiple cores these days, there are usually many more threads than there are cores, so thread switching (multiplexing) is still required.

Multitasking vs multi-threading

The terms multitasking and multi-threading are used somewhat interchangeably. While they are slightly different in concept, at the lower technical level (OS scheduling and CPU cores), the difference is very minor.

Multitasking means performing multiple tasks at the same time. The term itself is more widespread than computers alone, but within the domain of computers, a *task* generally refers to a single application/process. So multitasking means you are using multiple applications at the same time. Which you always do, these days. You may have an IM or mail client open in the background, or a browser, or just a malware scanner, or whatnot. And the OS itself also has various background processes running.

Multi-threading means running multiple threads at the same time. Generally this term is used when talking about a single process which uses more than one thread.

The 'at the same time' is as seen from the user's perspective. As explained earlier, the threads/processes are multiplexed, running for a timeslice at a time. So at the level of a CPU core, only one thread is running at a time, but at the OS level, multiple threads/processes can be in a 'running' state, meaning that they will be periodically scheduled to run on the CPU. When we refer to a running thread or process, we generally refer to this running state, and not whether it is actually running on the CPU at the time (since the time slices are so short, there can be thread switches dozens of times per second, so it is too fast to observe this in realtime, and as such, it is generally meaningless to look at threading at this level).

A process is a container for threads, as far as the OS scheduler is concerned. Each process has at least one thread. When there are multiple threads inside a single process, there may be extra rules on which threads get scheduled when (different thread priorities and such). Other than that, running multiple processes and running multiple threads are mostly the same thing: after each timeslice, the OS scheduler determines the next thread to run for each CPU core, and switches the context to that thread.

There are threads, and then there are threads

All threads are not created equal. This seems to be another point of confusion for many people. In this age of multitasking and multi-threading, it is quite common for a single process to use multiple threads. In some cases, the programmer may not even be aware of it. The OS may start some threads in the background for some of the functions he calls, and some of the objects he uses, even though he only uses a single thread himself. In other cases, the OS is designed in a way that it demands that the programmer uses a thread for some things, so that the thread can wait for a certain event to occur, without freezing up the rest of the application.

And that is exactly the point: Even though there may be many threads in a process, they are not necessarily in a 'running' state. When a thread is waiting for an event, it is no longer being scheduled by the OS. Therefore it does not take any CPU time. The actual waiting is done by the OS itself. It simply removes the thread from the list of running threads, and puts it in a waiting list instead. If the event occurs, the OS will put the thread back in the running list again, so the event can be processed. Usually the thread is also scheduled right away, so that it can respond to the event as quickly as possible.

Applications that use these types of threads are still seen as 'single-threaded' by most people, because the work is still mostly done by one thread, while any other threads are mostly waiting for an event to occur, waking up only to process the event, and then going back to sleep again. As a result, such an application will appear to only use a single core. The additional threads may be processed on other cores, but their processing needs are so minor that they probably don't even register in CPU usage stats. Even if you only had a single core, you probably would not notice the difference, since the threads could be scheduled efficiently on a single core (just like the example of playing an mp3 file earlier).

To really take advantage of a multi-core system, an application should split up the main processing into multiple threads as well. Its algorithms need to be *parallelized*. This is only possible to a point however, depending on the algorithm. To give a very simple example:

```
e = a + b + c + d
```

You could parallelize a part of that, like so:

```
t0 = a + b
t1 = c + d
e = t0 + t1
```

t0 and t1 can be calculated in parallel threads. However, to calculate e, you need the results of both threads. So part of the algorithm can be parallel, but another part is implicitly sequential. It depends on results from earlier calculations, so there is no way to run this calculation in parallel with other dependent calculations.

Amdahl's law deals with these limitations of parallel computing. In one sentence, it says this:

> *The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program*

The sequential parts result in situations where threads for one step in the algorithm have to wait for the threads of the previous step to signal that they're ready. The more sequential parts there are in a program, the less benefit it will have from multiple cores. And also, the more benefit it will have from the single-threaded performance of each core.

And that brings me back to the original point: people who think that the number of cores is the only factor in performance of multithreaded software.

The multi-core myth

This is a myth that bears a lot of resemblance to the Megahertz-myth that Apple so aptly pointed out back in 2001, and which was also used to defend the AMD Athlon's superior performance compared to Pentium 4s running at higher clockspeed.

The Megahertz-myth was a result of people being conditioned to see the clockspeed as an absolute measure of performance. It is an absolute measure of performance, as long as you are talking about the same microarchitecture. So yes, if you have two Pentium 4 processors, the one with the higher clockspeed is faster. Up to the Pentium 4, the architectures of Intel and competing x86 processors were always quite similar in performance characteristics, so as a result, the clockspeeds were also quite comparable. An Athlon and a Pentium II or III were not very far apart at the same clockspeed.

However, when the architectures are different, clockspeed becomes quite a meaningless measure of performance. For example, the first Pentiums were introduced at 66 MHz, the same clockspeed as the 486DX2-66 that went before it. However, since the Pentium had a superscalar pipeline, it could often perform 2 instructions per cycle, where the 486 did only one at most. The Pentium also had a massively improved FPU. So although both CPUs ran at 66 MHz, the Pentium was a great deal faster in most cases.

Likewise, since Apple used PowerPC processors, and AMD's Athlon was much more similar to the Pentium III than the Pentium 4 in architecture, clockspeed meant very little in performance comparisons.

Today we see the same regarding the core-count of a CPU. When comparing CPUs with the same microarchitecture, a CPU (at the same clockspeed) with more cores will generally do better in multithreaded workloads. However, since AMD and Intel have very different microarchitectures, the core-count becomes a

rather meaningless measure of performance.

As explained above, a single core can handle multiple threads via the OS scheduler. Now, roughly put, if a single core CPU is more than twice as fast as one core of another dualcore CPU, then this single core CPU can also run two threads faster than the dualcore CPU.

In fact, if we factor in Amdahl's law, we can see that in most cases, the single core does not even have to be twice as fast. Namely, not all threads will be running at all times. Some of the time they will be waiting to synchronize sequential parts. As explained above, this waiting does not take any actual CPU time, since it is handled by the OS scheduler (in various cases you will want to use more threads than your system has cores, so that the extra threads can fill up the the CPU time that would go to waste otherwise, while threads are waiting for some event to occur).

Another side-effect of the faster single core is that parts that are strictly sequential in nature (where only one thread is active) are processed faster. Amdahl's law essentially formulates a rather paradoxal phenomenon:

> *The more cores you add to a CPU, the faster the parallel parts of an application are processed, so the more the performance becomes dependent on the performance in the sequential parts*

In other words: the **single-threaded performance** becomes more important. And that is what makes the multi-core myth a myth!

What we see today is that Intel's single-threaded performance is a whole lot faster than AMD's. This not only gives them an advantage in single-threaded tasks, but also makes them perform very well in multi-threaded tasks. We see that Intel's CPUs with 4 cores can often outperform AMD's Bulldozer architecture with 6 or even 8 cores. Simply because Intel's 4 cores are that much faster. Generally, the extra cores come at the cost of lower clockspeed as well (in order to keep temperatures and power consumption within reasonable limits), so it is generally a trade-off with single-threaded performance anyway, even with CPUs using the same microarchitecture.

The above should also give you a better understanding of why Intel's HyperThreading (Simultaneous Multithreading) works so nicely. With HyperThreading, a physical core is split up into two logical cores. These two logical cores may not be as fast as two physical cores would be, but that is not always necessary. Threads are not running all the time. If the cores were faster, it would just mean some threads would be waiting longer.

The idea seems to fit Amdahl's law quite well: for sequential parts you will only use one logical core, which will have the physical core all to itself, so you get the excellent single-threaded performance that Intel's architecture has to offer. For the parallelized parts, all logical cores can be used. Now, they may not be as fast as the physical cores, but you have twice as many. And each logical core will still exceed half the speed of a physical core, so there is still performance gain.

One of the things that SMT is good at is reducing overall response time. Since you can run more threads in parallel, new workloads can be picked up more quickly, rather than having to wait for other workloads to be completed first. This is especially interesting for things like web or database servers. Sun (now Oracle) took this idea to the extreme with their Niagara architecture. Each physical core can run up to 8 threads through SMT. For an 8-core CPU that is a total of 64 threads. These threads may not be all that fast individually, but there are a lot of them active at the same time, which brings down response time for server tasks. Because of SMT, the total transistor count for a 64-thread chip is extremely low, as is the power consumption.

So, to conclude, the story of multithreading performance is not as simple as just looking at the number of cores. The single-threaded performance per core and technologies such as SMT also have a large impact on the overall performance. Single-threaded performance is always a good thing, both in single-threaded and multi-threaded scenarios. With multi-core, your mileage may vary, depending on the algorithms used by the application (to what extent are they parallelized, and to what extent do they have to remain sequential?), and on the performance of the individual cores. Just looking at the core-count of a CPU is about as meaningless a way to determine overall performance as just looking at the clockspeed.

We see that Intel continues to be dedicated to improve single-threaded performance. With Bulldozer, AMD decided to trade single-threaded performance for having more cores on die. This is a big part of the reason why Bulldozer is struggling to perform in so many applications, even heavily multithreaded ones.

This entry was posted in Hardware news, Software development and tagged architecture, computer, core processors, CPU, de gelas, enterprise-it, HyperThreading, multicore myth, multitasking, myth, performance, review, scheduling, Simultaneous multithreading, SMT, software, technology, threading. Bookmark the permalink.

26 Responses to *Multi-core and multi-threading performance (the multi-core myth?)*

Bob *says:*
June 1, 2012 at 4:08 pm

This is a great blog entry and I think very important as we start to see mobile devices based on Medfield hit the market. Many bloggers are lamenting it is a single-core CPU at 1.6 GHz but in many cases it is likely to deliver similar is not better performance as dual-core and quad-core ARM SoCs especially in the 2 GHz version.

Reply

Klimax *says:*
June 1, 2012 at 10:11 pm

IIRC it might be single core only, but it still has Hyperthreading (two threads).

Reply

Scali *says:*
June 2, 2012 at 10:24 am

Yes, HT and SIMD are two things I pointed out as strong points of Atom vs ARM in my coverage of Medfield: http://scalibq.wordpress.com/2012/01/23/intel-medfield-vs-arm/

MacOS9 *says:*
June 2, 2012 at 9:26 pm

A damn refreshing entry on multitasking and multiprocessing if I ever read one; no longer will I be jealous of 4-, 6-, and more-core processors with my modest Core2Duos.

(I remember there being comparisons on the internet years ago in the 90s between a PowerPC [the 604 by IBM] running at 120MHz, and being as fast as the popular Pentium Pro of the mid-1990s running at 166MHz – although that may have been a comparison between RISC and CISC and had nothing to do with multitasking.)

By the way, since I'm already on the subject, what are Scali's views on RISC vs. CISC processors? I remember Apple's

controversial switch from PowerPC (RISC) to CISC-based Intel around 2005 – but perhaps newer Intel processors (those supporting 64-bit) have also gone by now in the direction of RISC?

Reply

MacOS9 *says:*
June 2, 2012 at 11:47 pm

From what I've gathered by reading this entry , my question may be irrelevant, but I look forward to Scali's comments nonetheless.

Reply

MacOS9 *says:*
June 2, 2012 at 11:49 pm

The article in question is called "Cisc or Risc? Or Both?" (my apologies for the several posts, since I wasn't able to paste in the URL).

Reply

MacOS9 *says:*
June 3, 2012 at 12:18 am

Well let me continue embarrassing myself by offering one more post: have now stumbled onto your "RISC/CISC" article of late Feb. that has answered most of my questions: in other words, the complexity (or lack thereof) of instruction sets is largely irrelevant in the "post-RISC" world I take it?

Reply

Scali *says:*
June 3, 2012 at 9:13 pm

The Pentium Pro was actually the first Intel x86 CPU that used a RISC backend. It couldn't quite keep up with the PowerPC yet, but a few hundred extra MHz could compensate for it. As CPUs became larger and more complex, the extra cost of the CISC-legacy became smaller. Today, Intel even has a competitive x86 CPU for smartphones and tablets, the Atom codenamed Medfield: http://scalibq.wordpress.com/2012/01/23/intel-medfield-vs-arm/
So yes, it is largely irrelevant. In theory a RISC CPU may still be slightly more efficient, but in practice, Intel can compensate by using superior manufacturing technology (they are at 22 nm, where the competition is at 28, 32 or 40 nm), and by other technologies such as HyperThreading.
The irony is that this time it's the ARM CPUs that aren't very impressive in terms of performance per GHz.

Reply

anthonyvenable110 *says:*
June 19, 2012 at 7:02 am

Reblogged this on anthonyvenable110.

Reply

Pingback: *Multi-core and multi-threading performance (the multi-core myth?) | Scali's OpenBlog™ | Itsaat*

Pingback: *The Multi-Core Myth | Shwuzzle*

chris *says:*

October 11, 2012 at 4:43 pm

Very interesting read. So if clock speed (when analyzing different architectures) doesn't matter, what should I be looking at when I buy my next processor? How does one determine how "fast" a processor executes a thread.

Reply

Scali *says:*
October 11, 2012 at 4:48 pm

Well, the only way to get a decent performance estimate is to look at it on a per-application basis.
The only reliable way to know which CPU runs application X best, is to just benchmark them and compare the results.
Of course, application Y could be a completely different story again.
There is no simple answer.
Which is why there are so many myths... The MHz myth and the multi-core myth approach the problem of performance from opposite angles, and they both fail equally hard at indicating overall performance.

Reply

chris *says:*
October 12, 2012 at 5:40 pm

For a user like myself who mainly runs things like a browser and netbeans (to a lesser extent LAMP stack locally) how is that feasible? Even for a small business determining which hardware to purchase to run a database server, how is that really feasible...

Scali *says:*
October 12, 2012 at 6:11 pm

Funny how some people can't accept that not all answers are 'feasible'. If you want to have a computer for generic use, then your performance needs will be some weighted average of the applications you will be using, where the weights depend on your wants/needs/priorities.
You think that asking again will change my answer? No it won't. I'm a realist, reality is not always 'feasible'. Performance of a CPU architecture cannot be caught in a single measurement.

Having said that, I can give you a simple tip: Filter your application needs on based on how performance-critical they are.
Browsers are not performance-critical whatsoever. Depending on the size of your projects and how you use it, Netbeans is generally not performance-critical either.
A local LAMP stack (presumably for development purposes) is usually not performance-critical either (you're the only user, where the technology can easily scale up to hundreds or thousands of users at a time).

Also, if you want to purchase a database server, you've already answered yourself how feasible it is: 'database server', so you only have to look at database performance, and more specifically, only at the specific database software you intend to run on it.

Pingback: *Thought this was cool: Multi-core and multi-threading performance (the multi-core myth?) | Scali's OpenBlog™ « CWY Alpha*

Pingback: *Of GCs and Multi-cores « Ping!*

Sreejith *says:*
October 18, 2012 at 6:56 pm

So wat about the dual core and quad core processors in mobile phones, increasing the number of cores in it doest really count…

Reply

Scali *says:*
October 18, 2012 at 7:31 pm

Yes, the same goes there as well, as you can see here for example:
http://www.anandtech.com/show/5563/qualcomms-snapdragon-s4-krait-vs-nvidias-tegra-3
In many benchmarks, even multithreaded ones, the dualcore Krait is faster than nVidia's quadcore Tegra 3. Simply because the Krait has much better singlethreaded performance.

Reply

Hambster *says:*
November 13, 2012 at 4:04 am

Hi,
It's a good post. May I refer your post on my blog?

Reply

Scali *says:*
November 13, 2012 at 10:11 am

Sure, go ahead.

Reply

elpipo *says:*
December 14, 2012 at 10:29 am

Good article, it has brung me back to school.

As I'm getting old and little by little far from this, could you also discuss the case of vitualisation. Are cores really allocated to virtual machine or is this just an abstraction ? Is it stupid to say "I want one core for each virtual machine" ? How hypervisor handle real time allocation ? …

Reply

Scali *says:*
December 16, 2012 at 12:07 am

I think those details are implementation-specific. I don't use virtualization much myself, but I do use VirtualBox or VMWare from time to time for some cross-platform testing/development, and I believe they just map the virtual cores to threads on the host OS, without specifically pinning them down on cores (as you could do with SetThreadAffinityMask() in Windows).
Which would make sense, since this would still allow the host to dynamically allocate cores, and do proper load balancing. I suppose you could still use the process affinity mask to have each instance run on a subset of the cores, but I think it would be a bad idea in general.

Reply

Christos *says:*
December 22, 2012 at 10:51 pm

The Megahertz myth would have you believe that higher clock means faster.
Well, it's true, for given cpu architecture.
The "multi core myth" will have you believe that more cores equals more speed.
Well, again it's true, only this time there are two parameters to consider instead of one.
Architecture and application.
Intel fueled the MHz myth because it was what they had to offer relative to the opposition, now AMD will advertise more cores for the same reason.
You could then argue about the "64bit" myth that AMD started with the Athlon 64 that offered nothing in terms of performance while offering a lot in terms of marketing.
But still, it was the way forward and someone had to make the first step.
More cores equals more performance generally speaking and anyone who doesn't understand the pitfalls involved will just end up paying a bit more for a bit less, not a big deal.
At the end of the day it's exactly the same as in every aspect of life, the more knowledgeable make more knowledgeable decisions.
No point in getting too crazy about it.
At the end of the day I doubt those who know little about computers will read any of this any way.
The just buy a PC with "Intel inside" cause it's all they know.
So the world is good place to be right now it seems because the name they all know makes the best cpus at the moment, how lucky is that.
Come to think of it the MHz myth must have done people's pockets much greater injustice then.

Reply

Scali *says:*
December 22, 2012 at 11:10 pm

Not sure what point you're trying to make. Everything you say is already literally mentioned in the article, or is so obvious that it would not need to be mentioned.
Or well... some part is rather dubious. Namely, you say Intel fueled the MHz myth... You mean to imply that AMD did not? Which would be funny, since as I pointed out, Apple coined the term 'MHz myth' when they were competing with their PowerPC CPUs against the Intel Pentium III and the AMD Athlon. And the Pentium III and the Athlon were in a tight race towards the GHz mark at the time.

Reply

Pingback: *Multi-core and multi-threading performance (the multi-core myth?) | Scali's OpenBlog™ | @EconomicMayhem*