

# Re: How to develop a random number generation device

---

*Source:* <http://sci.tech-archive.net/Archive/sci.electronics.design/2007-09/msg03935.html>

---

- *From:* David Brown <david.brown@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>
  - *Date:* Fri, 21 Sep 2007 00:43:57 +0200
- 

John Larkin wrote:

On Thu, 20 Sep 2007 22:32:03 +0200, David Brown  
<david.brown@xxxxxxxxxxxxxxxxxxxxxxxxxxxx> wrote:

John Larkin wrote:

On Wed, 19 Sep 2007 23:06:17 +0200, David Brown  
<david.brown@xxxxxxxxxxxxxxxxxxxxxxxxxxxx> wrote:

John Larkin wrote:

On Tue, 18 Sep 2007  
18:15:07 +0200, David  
Brown  
<david.brown@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>  
wrote:

John Larkin  
wrote:

On  
Mon,  
17  
Sep  
2007  
23:04:03  
+0200,  
David  
Brown  
<david.brown@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>  
wrote:

<snip>

Re: How to develop a random number generation device

This  
is  
not  
about  
performance;  
hardly  
anybody  
needs  
gigaflops.  
It's  
all  
about  
reliability.

Until you  
can come  
up with  
some sort of  
justification,  
however  
vague, as to  
why you  
think one  
cpu per  
process is  
more  
reliable than  
context  
switches,  
this whole  
discussion  
is useless.

You define yourself by the  
ideas you refuse to consider.  
So I suppose  
you'll still be running  
Windows 20 years from  
now.

I run windows (on desktops) and Linux (on a  
desktop, a laptop, and a bunch of servers,  
and on a fairly high-reliability automation  
system I am working on), and I'd use  
something else if I needed an OS in my  
embedded systems. If something better came  
along, I'd use that – whatever is the right  
tool for the job.

The relevant saying is "keep an open mind,

Re: How to develop a random number generation device

but not so open that your brains fall out". I'm happy to accept that doing things in hardware is often more reliable than doing things in software (I work with small embedded systems – I know when reliability is important, and I know about achieving it in practical systems). But what I am not willing to accept is claims that you alone understand the way to make all computers reliable,

I have made no such claims.

You have repeatedly said that current OS's (software OS's running on one or a few cores) is inherently unreliable, while your idea of a massively multi-core cpu running a task per core would be totally reliable. As far as I can see, you are the only person who believes this. If I've misunderstood (either about your claims, or if you can show that others share the idea), please correct me.

using a hardware design that is obviously (to me, anyway)

impractical,

Can't help what's obvious to you

and you offer no justification beyond repeating claims that

"hardware is always more reliable than software",

Isn't it?

No it isn't. At best, you can compare apples and oranges and note that a ram chip is more reliable than windows, despite the former having more transistors than the later has lines of source code.

We agree that typical hardware design processes are more geared to producing reliable and well-tested designs than common software design processes. But that does not translate into a generalisation that a given task can be performed more reliably in hardware than software.

and therefore you can

Re: How to develop a random number generation device

practically guarantee that the future of computing will be dominated by single task per core processors.

I can't guarantee it. My ideas are necessarily simplistic, and would

Perhaps "guarantee" was a bit strong – but you stated confidently that your 1024-core one-core-per-task devices were "gonna happen".

That's probably true. Sun will soon be shipping 8-core, multithread processors. Looks like the number of cores per chip is at least doubling every year, now that clock speeds are no longer the holy grail.

So, in 5 years, with  $8 * 2^5 = 256$  cores, running maybe 1K threads, why context switch?

Cores don't scale like that – Sun have done pretty well with their 8 core 64 thread cpu. I don't imagine we'll see so very many more cores on a device, because the interconnections and cache scaling get too difficult, and the whole device is too limited in memory bandwidth. It could get more, but why would anyone bother when there is a better way? The sort of application that works well with these devices is multi-process (or multi-thread) server software, such as web, email and database serving. These also scale well in clusters – there is little point in trying to run one OS on a 64 core machine when you can just as easily run 8 OS's on 8 8 core machines and get the same performance without anything more sophisticated than standard network connections.

If you are looking for a high performance server today, you can buy rack units with 4 separate PC's, each with 1 or 2 sockets for 4 or 8 core SMP. Load them all up with Linux in a cluster, and you have the same processing power as a 32 core system at a fraction of the cost.

And unlike in a single massively multi-core chip, such clusters can have redundancy built in – thus greatly increasing their reliability.

As a future prediction, I would expect to see motherboards with multiple independent PC's on the one board, designed specifically for this sort of cluster. I also expect to see virtual Ethernet links on these boards.

And as for your context switch obsession, you do realise that in an SMP server system, context switches waste only a fraction of a percent of the processing power? On a web server with 64 virtual cores, you'd expect something like a few thousand processes to be alive at a time, but most of them will be sleeping – the main worker threads will occupy the cores with very little context switching, while the sleeping threads can run as needed.

In other words, the organisation that exists today works perfectly well. There are plenty of things that can be improved in the software and hardware, but nothing is fundamentally broken (except perhaps the software development methods of many companies).

Re: How to develop a random number generation device

## Re: How to develop a random number generation device

On the desktop side, there will be a gradual shift towards more multi-threading software for processor intensive applications like games and media converters.

get more complex in a real system. Like, for example, my multicore chip would probably have a core+GPU or three optimized for graphics, and maybe some crypto or compression/decompression gadgets. There's no point sacrificing performance to intellectual purity.

This is beginning to sound a lot more like a practical system – devices exist today with several specialised cores, particularly in the embedded market. Arguably graphics cards fall into this category, as do high-end network cards with offload engines. But that's a far cry from your cpu-per-thread idea, and it is done for performance reasons – *\*not\** reliability.

Well, the world is ready for OS reliability.

There *\*are\** reliable OS's available today – they just don't begin with the letter "W". There is plenty of unreliable software that runs on these OS's, but *\*nix* systems designed for servers are solid (as are VMS, Netware, and many embedded OS's).

What you are trying to say, I think, is that the world is ready for reliable desktop software – that's a very different matter, and one I'd agree with.

But the trend towards multiple cores, running multiple threads each, is a steamroller. So far, it's been along the Microsoft "big OS" model, but when we get to scores of processors running hundreds of threads, wouldn't a different OS design start to make sense? The IBM Cell is certainly another direction.

Forget windows – it's a bad example of an OS, and it's an extreme example of unreliable software. There is no "Microsoft big OS" model – they just have a bad implementation of a normal monolithic kernel OS.

## Re: How to develop a random number generation device

There are uses for computers based on running large numbers of threads in parallel – the Sun Niagara processors can handle 64 threads in hardware (running on 8 cores). But these do not use a core (or even a virtual core) per thread – the cores have context switches as threads and processes come and go, or sleep and resume. Clearly you will get better \*performance\* when you can minimise context switching – but no one would plan for a system where context switching did not happen. There is nothing to suggest that the system could be made more reliable by avoiding context switches, except in the sense of reliably being able to complete tasks at the required speed – it's a performance issue.

I believe I have been open minded – I've tried to point out the problems with your ideas, and why I think it is impractical to design such chips,

Sorry, I missed that part. Why is it, or more significantly, why \*will it\* be impractical to design a chip that will contain, or act like it contains, a couple hundred CPU cores, all interfaced to a central cache?

Perhaps I didn't explain it well, or perhaps you didn't read these posts – it's hard to follow everything on s.e.d.

The problem with so many cores accessing a shared cache is that you have huge contention for the cache resources. RAM cells get bigger, slower and more complex the more ports they have – it's rare to get more than dual-ported RAM blocks. So if you have 1000 cores all trying to access the same cache, you're going to have huge latencies. You also need complex multiplexing hierarchies for your cross-switches – as each cpu needs to access the cache, you basically require a 1000:1 multiplexer. Assuming your cache has multiple banks and access to some IO or other buses, you'd need something like a 1000:10 cross-switch. That would be really horrible to implement – you'd need to find a compromise between vast switching circuits and multiple levels introducing delays and bottlenecks.

Here's a brief view of the Niagara II – your device would face similar challenges, but greatly multiplied:  
<http://www.theinquirer.net/?article=42256>

If each core has an L1 cache to relieve some of the pressure (without it, the system would crawl), you then have a very nasty problem of tracking cache coherency. Current cache coherency strategies do not scale well – they are a big problem on multicore systems.

## Re: How to develop a random number generation device

With existing multiprocessor systems, it is the cache and memory interconnection systems that are the big problem. If you look at high-end motherboards with 8 or 16 sockets, the cross-bar switches that keep memory coherent and provide fast access for all the cores cost more than the processors themselves. Building it all in one device does not make it significantly easier (although it saves on some buffers).

There are alternative ways to connect up large numbers of cores – a NUMA arrangement with cores passing memory requests between each other would almost certainly be easier. But you would have very significant latencies and bottlenecks, a very large number of inter-core buses, and you'd still have trouble with the L1 cache coherence.

With a new OS, and certain significant restraints on the software, you could perhaps avoid many of the L1 cache coherence problems. In particular, being even more restrictive about memory segments would allow you to assume that L1 data is private, and thus always coherent. For example, if all memory came from either a read-only source for code, or was private to the task using it, then you'd have coherency. You'd need a system for read and write locks for memory areas, with a central controller responsible for dishing out these locks and broadcasting cache invalidations when these changed, but it might work.

However, you've lost out on a range of requirements here. First off, your cores are now far from simple, and the glue logic is immense. Thus you have lost all hope of making the device cheap and reliable. Secondly, you've still got significant latencies for all memory access, slowing down the throughput of any given core, crippling your maximum thread speed. The bottlenecks don't matter so much in the grand view of the device – the total bandwidth to the cpus should still be more than if it were a normal multi-core device.

Exactly! Except there is no context switch overhead.

So what? In all practical measurements, there is no context switch overhead in SMP systems today – it all drowns out in comparison to delays in I/O. If you are running cpu-intensive work on a desktop with one core and fast pre-emptive switching, then the switch overhead can be noticeable – but not on a server.

Thirdly, you've lost

compatibility with all existing software – it won't run most programs, as they rely on being able to have shared data access.

Exactly! We can't run .NET forever.

## Re: How to develop a random number generation device

Who would want to run .NET at all, especially on a server?

and why they would be impractical for general purpose computing even if they were made.

Why? Because Windows, and other "big" OS's like Linux, don't support it?

Yes, that's about it. To be more precise, it will be impractical for general purpose computing because it won't run common general purpose programs.

Circular reasoning. Why aren't we still running 1401 code?

I'm talking about compatibility at the source code level and above (i.e., the design of the software, and the way it works), not the object code. Many essential building blocks of the internet are based on code 15 years old or more – the \*nix architecture is 30 years old or so. Any hardware that can't run this \*kind\* of software (even after modifications) can't run common general purpose software.

Even with the required major changes to the software and compilation tools, and without the cache restrictions mentioned earlier, it would run common programs painfully slowly.

If the cache throughput is the limit, you get the same amount of computing no matter how many CPUs are running. CPUs can also have a little bit of local instruction cache, since code does not have to be kept globally coherent.

Code does have to be kept globally coherent (though it is easier to do so than for data), and cores can't keep running without data.

But you are right about the bandwidth limitations being a similar problem for having a few cores or many. It will be less of an issue for the few core device, since you'd have fewer latencies in switching all the data around the device. And if your 256 cores cannot do more real work than 4 cores could – what is the point in having them? Please don't just repeat that it avoids context switches – the tiny advantage that might give does not outweigh the costs of the rest of the device.

## Re: How to develop a random number generation device

I've repeatedly asked for justification for your claims, and received none of relevance. I am more than willing to discuss these ideas more if you can justify them – but until then, I'll continue to view massively multi-core chips as useful for some specialised tasks but inappropriate for general purpose (and desktop in particular) computing.

It's generally accepted that a microkernel-based OS will be more reliable than a monolithic system, because of its simplicity, but the microkernel needs too many context switches to be efficient.

A microkernel *may* be more reliable because of its modular design – each part is relatively simple and communicates through limited, controlled ports. That's far from saying it always *will* be more reliable. Much of the theoretical reliability gains of a microkernel do not actually help in practice. For example, the ability of low-level services to be restarted if they crash is useless when the service in question is essential to the system. Thus there are no reliability benefits from putting your memory management, task management, virtual file system, or interrupt system outside the true kernel – if one of these services dies, you're buggered whether it kills the kernel or not. A similar situation is found in Linux – because X is separate from the kernel, it can die and restart independently of the OS itself. But to the desktop user, their system has died – they don't know or care if the OS itself survived.

Most of the benefits of a microkernel can actually be achieved in a monolithic kernel – you keep your services carefully modularised, developed and tested as separate units with clear and clean interfaces. It's a good development paradigm – it does not matter in practice if the key services are directly linked with the kernel or not, since they are all essential to the working of the OS. About the only way a microkernel improves reliability is by enforcing this model – you are not able to cheat.

What *does* make sense is keeping as many device drivers as possible out of the kernel itself. Non-essential services should not be in the kernel.

<http://en.wikipedia.org/wiki/Microkernel>

So, let's get rid of the context switches by running each process in its own real or virtual (ie, multithreaded) CPU. Then nobody can crash the kernel. A little hardware protection for DMA operations

Re: How to develop a random number generation device

makes even  
device drivers safe.

You underestimate the power of software bugs – you'll \*always\* be able to crash the kernel!

No. Not if it's small and correct, and it's absolutely protected by the hardware, and it runs on a CPU that runs nothing else. I've written RTOS's that never crashed.

You can make systems small and correct when they are doing a limited and well-understood job – that's why we can make embedded systems that are reliable. It is also possible to make big systems that are correct and reliable, if you do it well enough (look at mainframes). But dividing a complex system into parts does not by itself make it more reliable – it only makes it easier for the developer to use solid development and test methodologies.

I'm not saying that smaller kernels and better protection through more advanced hardware are not helpful – merely that they are not a magic bullet (who cares if your RTOS never crashes if the application running on it dies? It's the whole system's reliability that is important), nor are they essential.

There's nothing wrong with dreaming, quite the opposite. But you have to be able to see when it is nothing but a dream.

Do I have to give all that money back? Roughly \$200 million so far.

Only if you've sold them nothing more than dreams!

John