# That lunch wasn't free after all

**Ronald Landheer-Cieslak**[1*]

***For correspondence:**
rlc@vlinder.ca (Vlinder Software)

[1]Vlinder Software — Founding Analyst

**Abstract** The Spectre and Meltdown bugs have shown that the free lunch of ever-increasing software performance thanks to ever-increasing hardware performance was indeed over a decade ago. We should therefore stop attempting to exploit instruction-level parallelism with ever more complex caches and ever more complex pipelines and branch predictors, and start exploiting the inherent parallelism of hardware. In order to do that, we need to change the way we think about software from our current imperative way of thinking to a more declarative way of thinking. At the same time, we need to change the way our computers think about software to allow them to exploit this more declarative style to use their inherent parallelism and free up die space currently used for caches and ILP.

## Introduction

In ***Sutter*** (***2005***), Herb Sutter proclaimed that "the free lunch is over": processors would stop getting faster and faster on a single core and programmers would have to learn new constructs to exploit the parallelism that comes with having multiple cores in a single CPU. Computers would still become faster, but mostly because "chip designers are under so much pressure to deliver ever-faster CPUs that they'll risk changing the meaning of your program, and possibly break it, in order to make it run faster".

***Lipp et al.*** (***2018***) and ***Kocher et al.*** (***2019***) showed this to be clairvoyant: optimizations implemented in CPUs were visible to user-space code and could be exploited, causing security issues in various applications. ***Chisnall*** (***2018***) argues that this is, at least in part, caused by "the quest for high ILP" (instruction-level parallelism), which has added significant complexity to processor design and is (he argues) largely an effort to turn modern CPUs into effective PDP-11 emulators[1].

Mitigating the Meltdown and Spectre bugs incurs a significant performance hit in all applications, regardless of which language they were written and and regardless of whether the software exploits the CPU's multiple cores using explicit parallelism or concurrency. This effectively rolls back a large part of the advances in processor speed since Sutter's article was published, effectively making us pay, collectively, for the last decade's free lunch. We now have to determine how we're going to pay for dinner, and what that dinner should look (and taste) like.

## Why we should change the way we think about software

Outside of the realm of scientific computing, we tend to think of software and software design in fairly linear terms, as instructions being executed in a particular order to perform a particular task. Imperative paradigms — procedural, object-oriented — work this way and allow functions, procedures and methods to exhibit side-effects: they assume that any line of code can change the state of any variable anywhere in the program, provided they have some way to access (reference) that variable. This way of thinking is inhibiting: it makes it more difficult to parallelize processes

[1]This also concurs with Sutter's contention, as cited in the previous paragraph: "chip designers are under so much pressure to deliver ever-faster CPUs that they'll risk changing the meaning of your program and possibly break it in order to make it run faster". The problem, of course, is thay they not only broke it, but they introduced security issues that software cannot easily work around.

because parallel processes, in this way of thinking, tend to mutate shared data. This leads to using synchronization mechanisms such as mutexes which, in turn, leads to problems such as deadlocks.

“Alternative” ways of thinking about software have traditionally tended towards event-driven designs, in which still-explicit threading mechanisms communicate with each other through events (ranging from single-bit signals to structured messages). Such events can be sent using simple functions like the Windows API’s `SetEvent` function, in the case of a single-bit signal, or can be sent through message queues, sockets, and the like in the case of structured messages. These designs have become ubiquitous in both application-level software and in operating systems, but when implemented using imperative approaches/languages, still assume universal read/write access to shared resources — which means access to those resources still has to be synchronized. In software, this means locks; in hardware, this means complex hardware to ensure cache coherency etc., as pointed out by ***Chisnall*** (***2018***).

With declarative, functional languages such as Erlang and Haskell, the developer’s “mental model” of the software they’re writing is radically different than it is for any software written in an imperative language. This mental model is important because it guides the way we reason about the software, and thus tends to determine where the bugs are. The differences are both important for the way the code is translated and executed by the implementation (compiler suite, CPU), and for the human mind reasoning about that code and that execution. Further, and importantly, they affect the way things can be parallelized by the implementation.

Because pure functional languages do not allow for side-effects for functions, code written in these languages does not assume universal read/write access to all memory, but can only access the resources passed to them as parameters. That means three things: firstly, functional, declarative code can sometimes be implicitly parallelized, as shown by ***Barwell*** (***2018***) in his PhD thesis:

> Automatic approaches to parallelization seek to simplify parallelization for the programmer by removing the programmer from the equation. (...)

This does not mean, however, that which parallelization is trivial:

> Despite being generally desirable, automatic approaches to parallelisation are limited in a number of ways. (...) Moreover, extending these analyses can be difficult; any program transformations must be correct for all cases, for example. (...) [L]imitations [to current approaches] can reduce the amount of parallelism that can be introduced to a program, and so potentially reduce any performance gains that can be achieved. (***Barwell, 2018***)

Barwell goes on to introduce two novel pattern discovery mechanisms for automatic parallelization, both of which are very promising, but both of which require a declarative approach to programming.

Secondly, as explained by ***Chisnall*** (***2018***), hardware designed for declarative, functional languages that do not assume universal read/write access to a flat memory can, in theory, be simplified w.r.t. current hardware that is designed to run software written in imperative languages.

Finally, it has long been argued that a more declarative style with stricter checks by the compiler leads to fewer run-time bugs and, thus, to greater productivity on the part of the developers. With this in mind, one might point out that it is also possible to use such a declarative style in an otherwise-imperative multi-paradigm language such as C++, which has built-in support for a functional style of programming and the community around which has, over the last decade or so, embraced a more functional style.

## Addressing the pitfalls of instruction-level parallelism

As pointed out by ***Chisnall*** (***2018***) and made evident by the Meltdown (***Lipp et al., 2018***) and Spectre (***Kocher et al., 2019***) bugs, chip manufacturers and designers have tended, over the last decade or so, to increase instruction-level parallelism in order to increase the performance of their CPUs.

They would not have done this if such parallelism was not an important part of improving that performance, but the Meltdown and Spectre bugs also show that this is far from trivial to implement.

Branch prediction and predictive execution are only necessary because the CPU would otherwise “waste cycles” loading values from, and storing values to, memory. This is also why caches, which are orders of magnitude faster than main memory, but still slower than CPU registers, exist. Keeping those caches coherent is also very complex, as shown by the large number of patents that exist in this domain[2].

However, hardware is inherently parallel: a CPU with more cores but less optimizations for linear, “access to everything for everyone” execution would under-perform on today’s software, but would perform much better than current CPUs with software written for those CPUs. We can see this directly when we look at software uses that the GPU not for graphics processing, but for other domain-specific applications in which massively parallel calculations are performed without requiring side-effects. The same is true for applications that use FPGAs or dedicated hardware for optimization. Whenever this is done, however, the developer needs to provide the data for the computation to be performed and let that computation take place in parallel — which requires another change in the way we think about software.

Explicit mechanisms for this type of thinking about software have emerged, over the last decade or so, in multi-paradigm languages such as C++, as well as in imperative high-level languages such as C# and Java: it has become more-or-less common practice to explicitly make certain computations and certain tasks parallel to the “main” program using “futures” and “promises” together with either explicit threads or thread pools. In C++, they have been part of the standard since C++11. These explicit approaches, like the side-effects found in imperative programs, are inhibiting: they force the developer to actively think about the parts of their program that can be parallelized, and to “make it so”.

## The cost of converging towards the computer

The advent and success of modeling techniques and UML, since the late 1990s, has shown that humans tend to reason well in terms of abstraction: the body of anecdotal evidence to this effect is practically infinite and the consensus among software engineering practitioners and researchers is practically complete. Some questions remain on what, exactly, the impact of modeling and abstraction is on software quality and maintenance cost and some studies[3] on these questions exist. None of these studies show any hint of a doubt, however, that modeling helps in reasoning about the modeled software.

While it is clear that humans think well visually and at a certain level of abstraction, it is also clear that computers do not: humans tend to get “bogged down” in details while computers excel at exactly the most detailed level. In the vast majority of cases, the human could care less about what the value of any given bit may be, while the computer cares for nothing else. The question then becomes how the two can efficiently and effectively come together: as we have collectively gotten better at working with computers and telling them what to do, and as computers have collectively gotten better at doing what we tell them to, we have tended, over the last few decades, to change our way of thinking about what we want computers to do to put that thinking in terms that a computer can better understand, at the cost of our own understanding and our own ability to effectively and efficiently reason about the instructions we give to a computer[4]. At the same

[2]A quick search on Google Patents showed several dozens of patents on this area filed and granted over the last few years, indicating significant effort being expended in innovation in this field.

[3]For example, in an effort to quantify the impact of modeling and to create a modeling tool to aid in certain tasks, ***Ohmann et al.*** (***2016***) showed that with adequate tools, developed for this purpose, students were both quicker and better at understanding and debugging tasks; ***Eloranta et al.*** (***2015***) showed that formal UML works better than informal “free-form” drawings; and ***Amrit and Tax*** (***2014***) examined different aspects of UML modeling to determine what makes an UML model more or less understandable.

[4]To put it differently: software engineers tend to spend disproportionate amounts of time on the tiniest of details, making sure the computer does exactly what is intended, and losing sight of the “big picture”.

time, the computer hides what it *actually* does and pretends to be a PDP-11 (as pointed out by ***Chisnall*** (***2018***)) at the cost of its own efficiency. These two trends are converging, and have perhaps already converged, to the point where both humans are least efficient, spending too much time and effort on details, and computers are also least efficient, pretending to implement an obsolete architecture based on a popular, but obsolete, abstract machine. The question of efficiently and effectively “coming together” is whether we can take our own strengths, working at an abstract level without having to worry about side-effects (which we’re not good at) and allowing us to visualise our logic (which helps us to reason about it), and combine those strengths with the strengths of a computer, working at the minutest of detailed levels with many parallel processes; and how far from that goal we have strayed in the last few decades.

## Changing the way computers think about software

In 1989, Philip Koopman wrote a book called *Stack computers: the new wave* (***Koopman, 1989***) in which he argued for a computer based on a multi-stack zero-operand machine (ML0)[5]. In this book, which presents a taxonomy of stack machines, a number of stack machine implementations and their relative benefits and drawbacks, and a general description of stack machines, he argues that stack computers can be fast while also taking less space on the die than RISC or CISC computers do, and being simpler to implement.

That stack machines with an instruction set based on the Forth language are relatively straightforward to implement is doubtlessly true[6], but it is also clear that stack machines have not had the commercial success that RISC and CISC-based architectures have. One of the reasons for this is the level of investment that already exists in RISC and CISC architectures: the largest three PC processor manufacturers, Intel, AMD and VIA, are heavily invested in CISC architectures based on x86. ARMH and manufacturers of ARM-architecture-based CPUs are heavily invested in RISC architectures. The two “camps” oppose each other, leaving very little space in the minds of investors and technologists for other types of architectures. Forth-based and WISC architectures are therefore not represented in the collective psyche of the group of people that have the money and/or the know-how.

The economic tide that has carried the RISC and CISC-based architectures does not negate, however, that these architectures continue to become more and more complex in order to execute software that makes certain assumptions about the machine and its memory model, and continue doing so at faster rates. Simpler machines that do not cater to those assumptions — that memory is a flat space that is universally accessible, that software is not written to be parallelized and therefore needs to be augmented using instruction-level parallelism, etc. — will continue to have an uphill battle, even if their advantages are becoming more and more obvious: a simple architecture with a small silicon footprint onto which a high-level functional, declarative language can be mapped as an assembly language into which other languages (such as Haskell) can be translated allows scalability through parallelization that cannot be matched with PDP-11 emulators. Changing the way the computer “thinks” about software and the assumptions it makes, by design, about the assumptions we make can help push software design towards a model, discussed above, that makes it easier to reason about software, while also making it more efficient to run such software.

## Conclusion

The requisite technologies already exist, and have existed (in relative obscurity) for decades. The necessary programming models also already exist and are becoming more popular with veteran programmers and language developers. Side-effects are more and more universally seen as “evil”

[5]**M** for multiple stacks, **L** for large stacks, **0** for a zero-operand instruction set — in this case based on the Forth programming language.

[6]In order to test this hypothesis, I sketched out an implementation of such a stack machine including an ALU, the stack, a instruction interpreter/control component and the general architecture for the internal bus (to access memory, I/Os and the like) in VHDL. While I stopped short of implementing a complete multi-core processor, I went far enough in the proof-of-concept implementation to show that such a processor was certainly feasible and relatively straightforward to design.

— and rightly so. This is, therefore, the price of dinner: we should change the way we think about software, change the way the computer thinks about software, make the two converge on a more declarative, functional paradigm that is easier to reason about for us humans and easier to parallelize for computers, and start exploiting the inherent parallelism of hardware. These changes, which will necessarily be costly at first, will change the flavor of programming but may also open the door to massively parallel processing in our pockets — just like the technological revolution of the last few decades has put the Internet in our pockets.

## References

Amrit, C. and Tax, N. (2014). Towards understanding the understandability of uml models. pages 49–54.

Barwell, A. D. (2018). *Pattern discovery for parallelism in functional languages*. PhD thesis, University of St Andrews.

Chisnall, D. (2018). C is not a low-level language. *Queue*, 16(2):10.

Eloranta, V.-P., Isohanni, E., Lahtinen, S., and Sievi-Korte, O. (2015). To uml or not to uml?: Empirical study on the approachability of software architecture diagrams. pages 101–105.

Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution.

Koopman, P. (1989). *Stack computers: the new wave*.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space.

Ohmann, T., Stanley, R., Beschastnikh, I., and Brun, Y. (2016). Visually reasoning about system and resource behavior. pages 601–604.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210.