Can Architecture Be Liberated from the von Neumann Style?

by Shade

As brilliant computer scientist John Backus noted in 1977, common programming languages are only becoming "fatter", but not stronger. Each of them inherits it common ancestor - von Neumann computer, with all its features and weaknesses. The problem has only become wider with the appearance of languages like C++, with an immense amount of mess which has accumulated over the years of the existence of such languages. For many of them it is even difficult or impossible to develop full-fledged compilers.

The answer to this is - well, but in the same time we have had elegant, powerful and reliable languages like Haskell or Erlang, right?

All of this is true, but what about architecture? Just remember how all commonly used architectures work. x86? ARM? RISC-V? All of them are basically arranged almost indistinguishably - a program counter, sequential execution of instructions, transfer of control by conditional jumps. **All of them are von Neumann machines** - the difference is only in the encoding of instructions, the number of registers and other minor details. Moreover, in fact, we have **almost no general-purpose architectures other than von Neumann**.

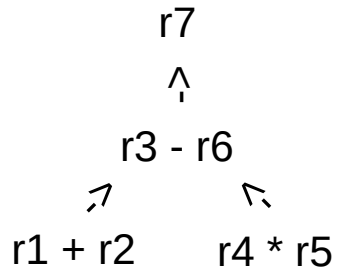
Common responses that can be heard - so what? After all, isn't von Neumann's style fundamentally the only possible approach? After all, all these lambda calculus, they argue, is just "abstract nonsense on paper", a **real** machine works with an external state, and should it not be otherwise than imperative?

In this article, I will show that this is not the case. Moreover, **real** machines are, ironically, **quite far** from the strict step-by-step imperative style!

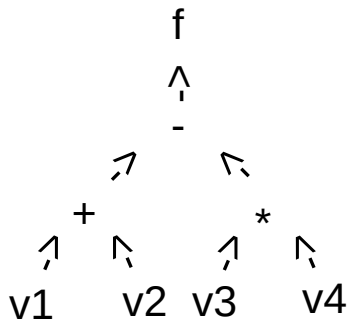
Firstly, sequential execution of instructions is not an intrinsic property of algorithms, but rather an artifact of how we think about them. Imagine such a simple algorithm - values are extracted from registers 1 and 2 and added, the result is stored in register 3. Then the values of registers 4 and 5 are multiplied and stored in register 6. Finally, the value in register 6 is subtracted from the value from register 3, the result is written to register 7. Written in pseudocode:

```
r1 + r2 -> r3;
r4 * r5 -> r6;
r3 - r6 -> r7.
```

Here you can notice that the first two instructions can be executed in any order and even in parallel - they are absolutely independent of each other. The natural representation of the algorithm is not a linear sequence of instructions, but rather a graph-like expression:



Or, without intermediate values:



where **v1-v4** is the input values,
and **f** is the output value.

Furthermore, if the CPUs of the 70s really executed instructions sequentially, in the 90s superscalar CPUs appeared with related features - instructions are loaded in large blocks, they can be dynamically reordered or executed in parallel, registers can be renamed, etc. In fact, modern CPU really "sees" a part of an algorithm more like a graph rather than a line! However, the "scope" of the CPU is limited by the size of the ROB. Even large CPU manufacturers, such as Intel, have practically not improved instruction parallelism for more than 10 years, the main increase in speed is due to the miniaturization of transistors. Indeed, if we try to increase ROB, then clock frequency will need to decrease, thus, the global optimum for von Neumann CPUs is achieved.

Moreover, a program written in sequential form is completely unsuitable for the needs of modern CPUs. Let's take a bird's view. There is a program with its **natural parallelism** and there is a CPU running **in parallel**. Between them is a **sequential** system of instructions. On the software side, the compiler is trying to squeeze a parallel program through the needle eye of a sequential instruction system; on the hardware side, the superscalar CPU is trying to restore the intended parallelism. But does the compiler know how the CPU actually works? Indeed, the work of today's compilers is not a strict science, but rather a dance with a shaman's tambourine.

But modern CPUs have another difference from typical CPUs of the 70s - the vast majority of them are **multi-core**, i.e. contain several von Neumann machines in one device. But how often do we encounter a situation where we need several linear threads of execution? Almost never! (Not to mention that the presence of multithreading leads to the need to cope with race conditions, deadlocks, livelocks and other unpleasant things.) Instead of thinking of multithreading as a method of **parallelizing a sequence** of instructions, it would be more correct to call it the **sequencing** of several branches of a **parallel** program graph!

It would probably be better if **all the parallelism** could be completely transferred to the CPU **without any restrictions**. But to satisfy this condition, the program must be explicitly written as a graph. This style can be called "declarative assembly" or something like that.

But that's not all. Recall that almost all general purpose CPUs are **clocked**. Nothing surprising if we remember that such an approach fits perfectly with the step-by-step semantics of algorithm execution. However, even in the case of simple CPUs of the 70s, this approach quickly reveals significant drawbacks. First of all, the speed of a clocked CPU is **limited by its slowest component** - the rest have to wait and, even more absurdly, consume enormous amount of energy! In modern CPUs, the problem is even more acute - the **speed of light** begins to impose its limitations. Thus, many CPUs are already have to be divided into several subsystems for further acceleration, each with has its own clock generator! (Recall also that it takes time to transfer data from RAM to the CPU and back, as a result, the bus frequency is 20-40 times less than the CPU frequency).

But what about to completely abandon the clock?

So, what should that Architecture of the Future be like?

First of all, as noted above, the program must be **explicitly written as a graph**. Forget the words "subroutine", "call", "return" and "stack"! Instead, think of the pointer to the **branch** as a true **reference** to the **expression** to be **evaluated** and **replaced** with the **resulting value**. No artificial "transfers of control" because the control flow is built into the very structure of the graph!

Furthermore, graph can be evaluated in **lazy mode** by default - it's not even need to pay attention to each individual branch of the graph! No ugly squeezing of laziness through von Neumann's Procrustean bed by "thunks" is needed.[1]

Also, such an architecture will require embedding computational elements **directly into RAM**. Instead of one or a few large cores, we have many (up to tens of thousands) execution units, making the device **look like a GPU or NUMA**. Wait, that means we don't even need a separation to CPU and GPU! In a way, the notion of a *central* processor is missing - such a computation is more like a swarm of termites building a termite mound together, with each element performing the task most accessible to it.

As a consequence, there are **no threads of execution**, since any execution unit can start reducing part of the graph. There is even **no program counter**, since the current state of the memory area where the program is stored uniquely sets the state of the program itself![2] Also, there is **no context switching**, since the "program context" like external registers is missing!

And since the execution units run on demand, there is **no clock** - our architecture is **asynchronous**. Among other things, this allows to eliminate performance bottlenecks and drastically reduce power consumption - **idle units rest** and just don't consume energy. (It also eliminates the need for various complex power management systems - no need to artificially change power settings in response to user actions!).

- But isn't this approach too radical? - you might ask. But remember that everything I just listed is already a property of modern computers to one degree or another! Superscalar and multi-core reduce the importance of instruction order and allow the use of parallelism, there is no single clock frequency for the entire CPU for a long time, and the CPU and RAM are slowly but steadily moving towards merging (and in many data processing architectures they are already merged!). In light of this, the proposed architecture looks not so much a radical departure from modern approaches, but rather a finishing touch, synthesizing what already exists!

- But such an architecture requires a functional programming language, isn't?

Yeah, it is!

- But I don't understand functional programming at all, it's so damn hard! Probably, only PhDs understand this, even not all of them, and you propose to write everything on this? This is madness!

This is usually the end of any talk about functional programming. In fact, the rejection of functional style does not seem to be simply a matter of taste. Apparently, it's something deeply wired in the gray matter (or are people just afraid of the imaginary "academicity" of FP?) As James Iry subtly points out in his "A Brief, Incomplete, and Mostly Incorrect History of Programming Languages":

> *1936 - Alan Turing invents every programming language that will ever be but is shanghaied by British Intelligence to be 007 before he can patent them.*
>
> *1936 - Alonzo Church also invents every language that will ever be but **does it better**. His lambda calculus is ignored **because it is insufficiently C-like**. This criticism occurs in spite of the fact that **C has not yet been invented**.*

However, there are rare attempts to think in a style other than imperative - a striking example is the EDGE/TRIPS architecture. According to the creators of the architecture, they intended to achieve a trillion instructions per second on a single chip (hence the name - TRillion Instructions Per Second) by 2005-2010, but moving away from imperative architecture, in the field of programming languages, all efforts were thrown at adapting a completely imperative language C. Needless to say, the goal of a trillion operations per second wasn't achieved? Apparently, there is some kind of barrier in the head, saying, let computers *may* not be von Neumann-ish - but they *must* be, period.

Even GHC developer Simon Peyton Jones ironically states that functional languages are "useless" (the problem is that over the years of systems written in Erlang, there are fewer known bugs than fingers on one hand!)

[1]Here I, in fact, lied a little - the algorithm in the form of a graph is not exactly without thunks, rather it is literally a colossal thunk of subthunks of subsubthunks... to primitive values at the end, however, the cost of laziness drops sharply due to the reduction in the size of the references by optimization of the "declarative assembler" itself.

[2]While there is no need for a program counter, a table of **head pointers** per each process, for example, might be needed. (For the operating system itself head pointer is the zero address).

And is it a coincidence that the most highly reliable software, for example, for pacemakers, spacecraft, or control systems for nuclear power plants, is created primarily in functional languages? And isn't it great when everything, right down to computer games, is as reliable as a pacemaker or a power plant?

- Well, but since there is no state in functional languages, how are you going to implement a functional architecture?

Another misunderstanding of the FP. Functional languages **have state**! What they don't have is an assignment operator. Indeed, if the assignment operator is banned in imperative languages, the program will be forced to become much more "functional". If in imperative programming there is a mutable state, then in functional programming any value or structure of values cannot be changed - you can only get a new state from the old one by applying a function. Thus, any value either has a reference to it (and is immutable), or has no reference and is garbage.

- But I/O!

Okay, just to remember how input and output are handled in imperative machines. Recall that a Turing-complete von Neumann machine requires only a few instructions: a set of logical operations (for example, Not, And, Or) and a conditional jump operator. However, such a machine is unable to communicate with external devices. To do this, it needs **system calls**. If we call operations only with RAM "pure" and operations affecting external devices "impure", then, as we can see, the instruction set itself, as it were, isolates "pure" instructions from "impure" ones! Why is the same impossible in a functional architecture? The only difference is that the first is linear, the second is represented as a graph. "Impure" instructions can be implemented in a graph - when such an instruction is reduced, some information is received or sent. If a linear code can send and receive information, why can't a graph?

- So what about OOP? I was taught design patterns, code reuse. Encapsulation, inheritance, polymorphism...

The thing is, functional languages are so expressive that **you don't need design patterns** at all! It's nothing more than a "crutch", due to the inexpressiveness of the language. Think about how often in C you typed something like this:

```
for (int i = 0; i < [limit]; i++) {
    [body]
}
```

Modern IDEs are able to fill in these templates automatically. Indeed, has it occurred to you that this clearly demonstrates **how poor and inexpressive** the common style of programming is?

Let's take Haskell for comparison. For example, if you need to derive from one function another with less arity, how would you do it in imperative languages? Of course, patterns, in this case "Adapter". In Haskell, you will never need this! Because all functions are curried, just apply only part of the arguments and the new function is ready!

As for encapsulation and polymorphism, these two things are not tied to any language or paradigm at all. **The real problem is inheritance**. It's because of it, as Joe Armstrong put it, that you want a banana, but you also get a monkey and all the jungle to boot.

When it comes to code reuse, functional languages simply have no equal! You will **never** have to write the same code again.

Here I present a brief table of the advantages and disadvantages of each programming approach:

Properties \ Paradigm	Imperative (C, C++, C#, Java, Go etc.	Functional (Haskell, Erlang, Idris etc.
Automatic parallelization	Very hard or impossible	All programs are ready for parallelization
Reliability	Usually low	Incredibly high
Testing/Debugging	Hard	Very easy
Formal verification	Extremely hard or impossible	Always possible and easy

Automatic optimization	Hard or impossible	Easy
Lazy evaluation	Possible, but appears rarely	By default
Expressiveness	Low or medium	Very high
Code reusability	Medium	Very high
Performance on imperative machine	Higher	Lower
Performance on functional machine	Lower	Higher

And another table, with architectures:

Properties / *Architecture*	von Neumann (x86, ARM, RISC-V)	Functional (???)
Parallelism	Absent or unclear	Present and clear
Lazy evaluation	Ugly implementation with thunks	By default
Energy efficiency	Lower	Higher
Asynchrony	No or rarely	Yes
Separation to CPU, GPU and RAM	Yes	No
Prevalence at present	Absolute	Exists only in project

In conclusion, here are the properties that the Architecture of the Future should have:

Explicit parallelism;
Absence of clock;
Simple lazy evaluation;
High performance and energy efficiency;
No separation to CPU, GPU and RAM;
~~*Backward compatibility with von Neumann architectures.*~~

Finally we face the last question: is it worth it at all? However, remember that today physical limitations are slowly but surely raising their heads since 2005. In 2000, there were forecasts to increase the frequency to 30 GHz in the 20s, but already in 2005 the maximum on-air (without liquid nitrogen) frequency was about 5 GHz and today it's not growing at all! Even the world record for the frequency of real general-purpose processors (under liquid nitrogen) has not reached 8.5 GHz! Single-thread performance has been growing for some time at the expense of ILP, but recently growth has stopped on x86 architectures and is close to stopping on RISC architectures. There remains a reduction in the size of transistors, but it slows down due to approaching the size of the atoms themselves. Experimental single-atom transistors have been around for a long time since 2004. Even energy efficiency is limited by Landauer's law, which prohibits dissipating less than 0.0175 eV per bit at room temperature. Despite the fact that modern computers consume millions of times more energy, in a few decades the limit will be reached. The notorious Moore's law is also slowly but surely approaching its end. Von Neumann architectures simply cannot meet all these challenges. In this regard, the proposed architecture meets all the needs of tomorrow.

References:

1. John Backus, 1977. Can Programming Be Liberated from the von Neumann Style?
2. James Iry, 2009. A Brief, Incomplete, and Mostly Wrong History of Programming Languages
3. The TRIPS architecture (cs.utexas.edu/~trips)
4. Simon Peyton Jones, 2011. Haskell is useless (youtube.com/watch?v=iSmkqocn0oQ)