In this article we'll explain what the von Neumann machine is, and why it is the source of some assumptions that are not only unhelpful in our programming, but they also turn out to be largely false.

The “stored program” computer architecture – see figure 1 – has served us well since John von Neumann described it in 1945. It is the basis for the modern computer: The processor reads the program instructions from memory and executes them, giving rise to the manipulation of data in memory. The memory and processor are separate entities, connected by a bus.

Figure 1 Von Neumann “stored program” hardware architecture

Now we're going to demonstrate some performance characteristics of your machine that don't quite fit the von Neumann picture. A singly linked list is a sequence of nodes where each item contains a reference to the next item in the list. Many languages provide linked lists as

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/blackheath/](http://www.manning.com/blackheath/)
part of their standard library. We are going to give our example in C because it's relatively close to the underlying machine instructions.

We traverse a linked list of a million items 1000 times (listing 1). It will shuffle the nodes into a random order before linking them, but if we give the --no-shuffle option on the command line, it won't do that. We generate the random numbers for the shuffle even when not shuffling, so we can be sure the random number generation doesn't account for the performance difference. You can find this code under sodium/book/von-neumann/ in the Sodium project.

Listing 1 How long does it take to traverse a linked list?

```c
#include <stdlib.h> #include <assert.h>

typedef struct Node {
    struct Node* next;
    unsigned value;
} Node;

void shuffle(Node** nodes, unsigned n, int doit) {
    unsigned i;
    for (i = 0; i < n; i++) {
        unsigned j = (unsigned)(((long long)random() * n) / ((long long)RAND_MAX + 1));
        if (i != j && doit) {
            Node* node = nodes[i];
            nodes[i] = nodes[j];
            nodes[j] = node;
        }
    }
}
```

so let's link the nodes in the order they were allocated in and see how long it takes:

```
time ./linked-list --no-shuffle
user 0m3.390s
```

Now what if we link the nodes in random order?

```
time ./linked-list
user 1m19.563s
```

23 times as long. Why?

#1 Generate random numbers always

#2 Swap only if shuffle is enabled

1.1.1.1.1 5.1.1 Why so slow? The cache
Actually the second run wasn't slow: the first one was such an amazing feat of engineering that it just made the second one look slow by comparison. Today's machines are actually based on an architecture called NUMA, standing for \textit{non-uniform memory access}.

The code doesn't quite do what the von Neumann picture would suggest. What's going on here? Between the main memory and processor in a modern machine, there's an artifice known as a \textit{cache} – see figure 5.2. A cache is a bank of memory that keeps a local copy of the most recently accessed parts of the main memory.

![Figure 5.2 Today's non-uniform memory access (NUMA) architecture](image)

When you access data that is already in the cache it's called a \textit{cache hit} and when you require a costly read from slower memory, it's a \textit{cache miss}. When it misses, the cache doesn't just fetch the requested data. It fetches a small chunk of data, typically 128 bytes or so, and it stores that into the cache. This is done because an assumption of \textit{locality} often holds true in practice: Any memory we access is likely to be \textit{nearby} something accessed recently.

That's why in our example, shuffling the nodes killed the performance. It just happens to be likely in my operating system that each allocated memory block is adjacent to the previous. The assumption of locality holds true and when we come to read data, it has often been prefetched. But when we shuffle the nodes, locality is destroyed so each loop is almost guaranteed to be a cache miss.
A cache miss exposes us to the **latency** of a fetch to main memory. Latency itself is not bad if we can give the CPU other work to do while it is waiting. The linked list structure means each loop depends on information from the previous loop. Because of this our program can't supply the CPU with any work, and the CPU must block. Our program falls off a “performance cliff”.

### 1.1.1.1.1 MULTI-PROCESSOR MACHINES

Caches get a lot more complicated with more than one processor. If one processor writes to memory, it must clear the caches of all the other processors. If several write at once, any conflicts must be resolved. Arbitration is the term for all the negotiation that takes place.

Having processors fight over the same memory causes a lot of arbitration and is known as **cache contention**.

Ultimately the NUMA architecture is really a set of processors with local memory with an elaborate illusion of shared memory between them. Can it scale to 1000 processors? I don't know.

#### 1.1.1.2 5.1.2 The madness of bus optimization

Often people will optimize their code for cache and bus performance. The general rule is that memory accessed temporally nearby should be physically nearby, and each processor should have its own local memory pool. But there are many more rules.

The C programming language gives you almost direct access to a contiguous block of memory. The ability of the compiler to optimize automatically for cache and bus performance is limited. For example, when we have a pointer in C, the compiler is prevented by the design of the language from transparently relocating the allocated block somewhere that might fit the temporal memory access patterns better.

**Q: Why are we in this strange situation?**

**A: Because modern machines are forced by current languages to pretend to be a machine that hasn't existed since the 1970's.**

To get the performance we expect today out of existing software, caches have become extremely complicated. For an application programmer to optimize their code for cache efficiency is generally not a good idea, yet people do exactly this. These are our reasons for saying so:

- Hardware architectures have been made complicated so they can run existing software fast.
• This complicated architecture means that optimization is largely beyond the ability of a programmer to optimize for cache and bus performance by hand. Programming is difficult enough already.
• This optimization should be the job of the language compiler, but most of our languages are not well designed for this.
• Optimizing an application by hand locks it into today's architecture, but it won't be optimized for tomorrow's. This entrenches the approach, making innovation in hardware more difficult.
• End result: A situation where software and hardware mutually complicate each other.

1.1.1.2.1 Getting the best bus performance out of your code

The Intel® 64 and IA-32 Architectures Optimization Reference Manual is 800 pages long and contains advice like this (section 3.6.12):

"If there is a blend of reads and writes on the bus, changing the code to separate these bus transactions into read phases and write phases can help performance."

"Note, however, that the order of read and write operations on the bus is not the same as it appears in the program."

"Bus latency for fetching a cache line of data can vary as a function of the access stride of data references. In general, bus latency will increase in response to increasing values of the stride of successive cache misses. Independently, bus latency will also increase as a function of increasing bus queue depths (the number of outstanding bus requests of a given transaction type).”

Did you get that?

The processor is working with sequential instructions that mutate state in place. When it blocks on a memory read, it must analyze the dependencies in the code to find anything it can execute that doesn't depend on the outstanding data. In chapter 1 we talked about how the programmer's job is typically largely concerned with translating dependencies into a sequence.

When we do things this way, the compiler doesn't have the original dependencies to generate better code. Now the processor has to extract whatever dependency information it can from the sequential instructions to get any performance. Its ability to do this is quite limited.

Large-scale parallelism on single processor is not really possible with this design.

WHAT ARE WE EVEN DOING THIS FOR?
We've built our practices on some shaky assumptions.

The von Neumann machine is a hardware architecture designed around state mutation that was common in the 1970's. Our programming languages were designed to mutate state on a von Neumann machine and they haven't changed much. State mutation is assumed to be efficient but the reality is more complex.

There are mathematical reasons behind the “complexity wall” experienced in commercial software projects: State mutation creates a maze of possible data dependencies such that unraveling them is an intractable problem. This makes programming harder and complicates parallelism and optimization. Object-oriented programming brings order to state mutation, but this just entrenches an approach that doesn't help anyone.

The von Neumann machine has a design bottleneck that limits its speed, but our languages tie us to it. In order to run existing software fast, modern machines go to great lengths to pretend to be von Neumann machines.

1.1.1.2.2  Complex? Now add more processors

<table>
<thead>
<tr>
<th>When your code is based on mutating program state in place, and you want to parallelize it to run on multiple processors, you have to protect the state with locks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This style of programming is prone to non-deterministic bugs, meaning you can get race conditions or deadlocks that occur only one time out of a million runs at random.</td>
</tr>
<tr>
<td>This style does not scale with program size. The reason: Coarse-grained locks are safe but defeat parallelism. Fine-grained locks require policies for acquiring them in the right order that increase in complexity with program size until they become intractable.</td>
</tr>
<tr>
<td>Result: Non-deterministic, difficult to reproduce bugs increasing with ballooning complexity. If you are an experienced programmer, then this ought to give you the wehi – the fear.</td>
</tr>
</tbody>
</table>

In summary, we are **programming in a bad way**, because it's **compatible with a nonexistent, inefficient hardware architecture**, forcing us to make **complicated hardware emulations** of it, so that it's **complicated to optimize** our software, further **entrenching** the hardware emulation. See figure 5.3.
BUT WE CAN BREAK THE CYCLE

To think NUMA – today’s optimization of the von Neumann machine – is the only possible architecture is to think in a limited way. There are many ways to build a computer. For example,

- single processors with local memory connected by fast Ethernet,
- massively parallel array processors, such as Graphics Processing Units (GPUs),
- Field Programmable Gate Arrays (FPGAs),
- optical computers,
- quantum computers?

Perhaps future computers will be seamless hybrids of multiple architectures, where each bit of code runs on the hardware that suits the underlying problem best.

The fundamental mistake we're making is programming away from the problem, and towards the machine. In the process, we make our job harder than it needs to be and we limit the options of the compiler and the processor maker.

**HOW DOES THIS RELATE TO FRP?**
To future-proof our code and free us up for hardware innovation, we just need to do one simple thing: Program in a way that fits the problem, and gives dependency information to the compiler – and in our specific case, the FRP system – so it can write the best code for the machine.

Functional programming in general does this by eliminating state mutation. FRP does this in a more specific way for one problem space. Of the architectures listed above, the one that fits FRP the best is the FPGA, although the fit is not perfect. This relationship would interesting to research.

When a program runs in parallel, it can achieve the same throughput with considerably less power consumption. This basic fact is the true reason why parallelism is here to stay. Parallelism is the pachyderm in the parlor that will ultimately force us to adopt ways of programming that are focused on the problem, not on the machine.

**NOTE** FRP is still in its early days, and we are a long way from saying parallelism is a direct selling point of FRP. Current FRP implementations don't do much for parallelism yet. And in general, parallelism is not an easy problem. But FRP is inherently parallelizable in a way that traditional programming isn't.