An Analysis of

Hardware Performance against Expectations

Josh Spieth

CS 393: Operating Systems

Dylan McNamee

May 2023

Introduction

In the past fifty years, hardware in commercial computers has increased its performance and speed dramatically. In the 70s, the first personal computer, which was incredibly rare to have at the time, had 256 bytes of RAM, an 8 bit 8008 intel processor, and an LED panel. It took extra work and money to even have a display or keyboard attached.[1] This computer, the Altair, cost the equivalent of 1800 dollars today. Now, we can purchase gigabytes of ram and terabytes of storage, all for significantly cheaper prices, and expect much better performance from our hardware. However, the increase in software speed and performance has not kept up with this rapid advancement in hardware. Various parts of computers, from file systems to DRAM and processors, have all advanced their speed and efficiency at varied rates, and maintaining optimization of each part while others may not be advancing at an equal rate has been a challenge for decades. There have been multiple theories on why this discrepancy has occurred, from holdovers from C and unix being outdated, to issues with how often we write to disc. In this paper, I will look at the history of advancements in hardware compared to software, how software has taken advantage of some advancements in hardware while failing to fully utilize others, and aim to understand what constricts software speed and performance from being what it could be.

[1] https://www.lighterra.com/articles/historyofcomputers/1970s.html

Historical Background

The discrepancy between software and hardware is not a new problem, and has been studied for decades. In 1990, John K Ousterhoust, at the University of California in Berkeley, ran a series of experiments on various common operating systems along with different hardware specifications to see what was the major bottleneck in system performance. His general findings were that CPU speeds were scaling faster than memory speeds, and thus creating a bottleneck there, and disc writes, which are incredibly time consuming, were fairly common and thus the CPU would have to be idle between disc writes. To get these findings, he ran a series of tests, including kernel entry and exit, context switching, select, block copy, read from file cache, open-close, create-delete, and then one larger scale metric, a modified version of the Andrew benchmark, which works by copying a directory hierarchy with source code for a program, stat-ing every file, reading contents, and then compiling the code.

Through these tests, Ousterhoust came to his conclusions about the discrepancy between processor and DRAM speeds causing issues, as well as the slow write time to disc being a major slowdown. On the whole, Ousterhoust noted that benchmark speeds were much slower than what he would have expected given the processor speeds. More specifically, for the discrepancy between processor speeds and memory speed, he noted a three to four times difference in the ratio between CPU speed relative to memory bandwidth between old CISC architectures and newer RISC architectures. He noted that unless memory bandwidth improved rapidly, this would continue to be an issue as the gap between processors and memory could grow, and thus mostly waste our improvements to processors.

Although the difference between DRAM and processor speeds is important to address, there is not much software engineers can do about it. However, Ousterhoust also noted issues with operating systems. Operating systems derived from UNIX (which is a lot of them), in order to run operations that modify files, which is very common, the machine requires synchronous disk I/O. Since disk writes are so much slower than any operation with the cpu, the cpu will often end up just waiting for the disk. The issue with waiting to do disk writes is that if the computer dies or has some issue, all the data that was going to be written will be lost, so you don't want to wait too long before writing to disk. Ousterhoust mentions a few possible solutions, including smarter and better structured file systems, or possibly having an intermediary stage between CPU and disk, where the CPU writes to non volatile memory which is then later can be moved to disk. In the case of a crash, you will still keep your non volatile memory, and you only need to disk write things that are going to be kept around for a while. This solution involves more overhead, but can decrease the number of disk writes needed, and overall greatly speed up performance.[2]

For more context on the history of memory and file systems, and their increase in power, we can look at a presentation given at the 2015 Symposium on Operating Systems Principles, by professor Mahadev Satyanarayanan.[3] In his presentation, he detailed the four main aspects that have driven progress in computers: scale, speed, transparency and robustness. For our objective of understanding performance issues, transparency and robustness aren't too critical, but scale and speed are the two most important factors in analyzing performance. As he presented, the best way to analyze scale is to look at the cost of certain metrics of memory or computing power. The price of memory has essentially had thirteen different stages of magnitude in the last 60 years,

[2] https://web.stanford.edu/~ouster/cgi-bin/papers/osfaster.pdf
[3] https://sigops.org/s/conferences/sosp/2015/history/04-satya-slides.pdf

each stage being approximately 10x cheaper than the previous one. Scale has increased drastically, and it is easy to see evidence of this in what kind of machine is readily available today versus in the 50s.

Looking at performance is more interesting to our question though. From 1980 to 2010, DRAM speed doubled approximately every ten years. However, processor speed was doubling approximately every 18 months. This obviously leads to a quickly growing discrepancy in the speeds of these two different components of computers. When we connect this to the fact that Ousterhoust, in the 1970s, was already concerned that discrepancy between processor speed and DRAM speed was causing issues, we have to keep in mind that it might not just be software inefficiently using hardware that is causing slowdowns, it is possible that components of hardware itself are a primary contributor to issues.[4]

Transparency and robustness will come up somewhat in our discussion of optimizing our modern computers. Transparency is ensuring that programs act as people think they should, and making user experience as good as it possibly can be, and robustness is the handling of human and system errors. For our question, we will see how some facets of programming lack transparency and robustness because they aren't attuned to the capabilities of modern machines, and our attempts to utilize the greater processing power of our computers today with these old systems can lead to difficult coding process, and risks of dangerous memory leaks, among other issues.

[4] https://sigops.org/s/conferences/sosp/2015/history/04-satya-slides.pdf

Unix and C

To understand the main issues facing our computers today, we have to understand UNIX and C, and how their part in computer systems since their inception has led to what we are dealing with today. There are many computer scientists who point at the use of Unix, C, and Posix as a major drawback to performance today. Since these languages and tools were invented, they have been in heavy use, but they were designed with the machines of the time in mind, which were generally much slower single core computers, and without the incredibly powerful GPUs and large amounts of RAM that computers commonly have today.

George V. Neville-Neil, also known as Kode Vicious, is a popular computer science blogger who wrote about his perceived issues with Unix and Posix. He claims that since the advent and rise in popularity of Posix (and Unix), any program that was going to be used had to be made usable within the framework of Posix, and this leads to some serious throttling of progress, or as he puts it, “an elephant that has been sitting on our collective backs for nearly 40 years: Posix”[5] Neil goes on to specify the ways in which Posix is holding programmers back, primarily about how it isn’t designed for the powerful machines we have today. For example, pthreads API, which is the multithreading part of Posix, is so famously bad that programmers are warned never to use it. This was not much of an issue when Posix was invented, as very few machines had multiple cores to run on, but today even a relatively weak commercial computer has multiple cores, and should take advantage of multithreading. He also addresses that most software design classes still work on a paradigm of single core, limited memory, large storage computers. This is not at all accurate to any computers in use today. Neil claims that in order to

[5] https://dl.acm.org/doi/fullHtml/10.1145/3570921

optimize our computers and programs, we need to buck the paradigms that Posix focused coding requires, and take advantage of our modern memory and processors.

The other old programming tool that some computer scientists claim hold us back is the language of C. David Chisnall, a computer science researcher at University of Cambridge, wrote an article in 2018 entitled “C is Not a Low-level Language”, in which he details the issues with C and how we use it, especially as it works on modern machines. As he writes, when C was invented it was a good low level language for the PDP-11, a popular computer in the 70s. These computers only had one core and limited one dimensional memory. C worked well to utilize a computer with these restraints. However, our attempts to use C on modern computers have made it not an effective low level language, and generally just not useful in optimizing our computers and programs. The main benefit programmers generally use low level languages for is their speed. However, the only reason C code is fast on our modern machines are incredibly complicated and well designed compilers that adapt C code into instructions that take advantage of modern machines. Even with these intelligent compilers, doing things like creating multithreaded programs is difficult and inefficient, and in the modern day, with commonplace multi core computers, multithreaded coding should be easy and simple to do. At the end of the article, after detailing the various ways in which C is not actually a low level language, and working with C is unnecessarily confusing and inefficient, Chisnall addresses the idea that multithreaded coding is hard. He claims that this is false, and that we have plenty of modern examples of well done simple multithreaded coding. He says, “It's more accurate to say that parallel programming in a language with a C-like abstract machine is difficult”[6], and given how much parallel hardware we have in modern machines, he states that really what he is saying is C is just not good with modern machines.

[6] https://queue.acm.org/detail.cfm?id=3212479

This brings us back to the issue of transparency and robustness, brought up by Satyanarayanan in his presentation. For programmers today, it is very difficult to implement multithreaded programs in C, compliant with posix limitations. This is failing the robustness criteria that Satyanarayanan claimed drives development of computers and programming. Ideally, with machines that are optimized for multithreaded programs, coding such programs should be simple and efficient, and not much harder than programming single threaded programs. Programming in C today fails to be robust. Programmers struggle to write multithreaded programs, which are the ideal way to optimize modern machines.

Modern Languages and OS

In recent years, there have been developments of operating systems that do not rely on Unix/Posix, along with new languages that provide the advantages of being low level, similar to what we expect C to give, while taking better advantage of what our modern machines provide.

The main language used today that is an attempt to replace C as a low level language is Rust. There have been many tests done on whether Rust is actually faster than C, and although in simple applications it generally hasn't been, there are still a lot of advantages to it due to how it uses modern machine capabilities in more sensible ways. In 2021, Codilime, a software engineering company, ran a test on Rust vs C in low level network programming, looking for both safety and performance.[7] They had noted that Rust had become increasingly popular in the previous five years, with Facebook joining the Rust foundation, and Linux kernel developers proposing including Rust in the Linux kernel, which was originally just C. The article starts by noting the innate advantages of Rust over C, regardless of the performance tradeoffs. C

[7] https://codilime.com/blog/rust-vs-c-safety-and-performance-in-low-level-network-programming/

compilers don't worry about safety. If a programmer is careless, they could easily violate memory and attempt unsafe programs. However, the Rust compiler catches all of these issues, and ensures your code is safe, which makes it much easier and more accessible to do low level programming, which has the inherent risk to do unsafe things to memory, since it works on that scale.

Their tests ran a variety of packet sending programs in both Rust and C and measured the speed of execution. They found that, generally, Rust was a little bit slower than C, but not by much, and noted ways in which Rust libraries could be improved to lessen the gap.[8] C has been around much longer, and has been optimized, whereas Rust still has room to grow. Despite the slightly slower speed, they still were impressed by Rust, particularly because of the ways in which it improved on C, primarily on its memory management and multithreading, which are both much safer and easier to implement in Rust than it was in C. This highlights how Rust is optimized for modern machines more so than C, with multithreading in particular, which has been a major focus of optimizing modern computers. Rust is significantly more robust than C, and not much slower in any metric, which makes it an appealing choice looking forward to maximizing our programs and computers.

Modern OS's have had less progress in moving away from their legacy than languages. There have been a few attempts to create OS's not reliant on posix and linux, but not many have caught on as popular. Most popular OS's today are at least linux-like, and thus constricted by the same problems that posix has. One attempt at an OS totally separate from the constraints of Unix is BareMetal OS. This is a very simplistic operating system, written fully in assembly language,

[8] https://codilime.com/blog/rust-vs-c-safety-and-performance-in-low-level-network-programming/

with no dependencies. It has minimalistic operations, and not much testing has been done on its performance compared to legacy operating systems.[9]

Conclusion

There is no definitive answer on why our computers and programs aren't running as fast as we would expect given the advancements in hardware over the last five decades. From mismatched performance upgrades in various parts of hardware, to issues with the languages we code in and the operating systems we run on, there are a multitude of reasons we aren't optimizing our modern machines. As we begin to modernize our methods of programming, and move away from systems that relied on lower capacity machines, we will see easier methods to multithread our programs, and operating systems that don't rely on outdated assumptions about our computers. Rust is a good first step in the path away from C and Unix, but our operating systems are still hamstrung by the limitations of Unix/Posix, and performance of Rust is still not quite as good as C at the very low level, since it hasn't been optimized for years in the same way that C has. However, progress is being made, Rust has been growing in popularity, and we are slowly moving away from the dependencies that have been hamstringing our performance for decades.

[9] https://en.wikipedia.org/wiki/BareMetal

Citations

https://www.lighterra.com/articles/historyofcomputers/1970s.html

https://web.stanford.edu/~ouster/cgi-bin/papers/osfaster.pdf

https://sigops.org/s/conferences/sosp/2015/history/04-satya-slides.pdf

https://dl.acm.org/doi/fullHtml/10.1145/3570921

https://queue.acm.org/detail.cfm?id=3212479

https://codilime.com/blog/rust-vs-c-safety-and-performance-in-low-level-network-programming/

https://en.wikipedia.org/wiki/BareMetal