Code Vault

Gerard J. Holzmann

I didn't write my first program until about five years after that famous first NATO conference on Software Engineering in 1968, and it took me a couple of years after that to realize that there was something very special about this field. What is special about it is the apparent inevitability of bugs, and the unreasonable difficulty of intercepting them. The same feeling that this is not how things ought to be in a proper engineering discipline has inspired many others. For instance, Maurice Wilkes expressed it as follows in his memoirs [1]:

> *"By June 1949 people had begun to realize that it was not so easy to get programs right as at one time appeared. I well remember when this realization first came on me with full force. […] It was on one of my journeys between the EDSAC room and the punching equipment that […] the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs."*

The effectiveness with which I could hide subtle bugs in my programs was not hampered by the fact that I had oceans of time to reflect on the quality of my code in between runs, forced by the inefficiency of the then standard batch processing of jobs on large mainframes. After each failed run I was tempted to conclude that this time surely the machine was at fault. But it never was.

So what changed since then? Depending on your point of view everything has changed, or nothing much has changed at all. The part that didn't change much is that we still struggle with writing code that is robust enough to trust. The part that did change dramatically is the performance of the hardware that runs our code.

The Trash-80

I bought my first PC, a TRS80 Model II, sometimes lovingly referred to as the Trash-80, in 1981. I paid about $3,800 for it at the time, which is the equivalent of about $12,000 today. The Trash-80 ran at 4 MHz and came with just 64 KB of RAM. It had no hard disk. All data was stored on 8" floppy disks that could hold about 500 KB each. For the same price today you can buy not one but four quad-core PCs, running at 4 GHz and with 32 GB of RAM each, or together about 16,000 times the processing capability and 500,000 times the amount of RAM of that old Trash-80, and with a hard-disk about a million times larger than those old floppy disks. The difference is staggering.

Of course we've found great ways to use up all that extra power, to make sure you don't notice it too much when you use your PC today, but the power is truly there. To see this more clearly I revived an old image processing program called *Pico* that I wrote in 1984 [2]. The program defines new images by evaluating a user-defined expression once for every pixel in the new image. *Pico* expressions can for instance refer to values, Cartesian or polar coordinates, trigonometric functions, and to parts of other images. That processing can be compute-intensive, so it makes a nice test case. The original code used an on-the-fly compiler written by Ken Thompson to turn the *Pico* expressions into executable code for a

VAX-750 computer. I separately also wrote an interpreter for the language that was more portable, but that was of course much slower. Even though the VAX-750 ran at only 6 MHz, the on-the-fly compiler made it possible to see the result of most image transformations in a few seconds, though still only for relatively small images. It was impressive enough that in 1989 this early digital darkroom tool landed me a spot on CNN for a few fleeting minutes of fame. (You can find the video on my homepage.)

With a few small tweaks I ported the original code to work on my current desktop and, predictably, the interpreted version of the code is now magnificently faster than even the compiled code had been in the 1980s. The main tweak I had to make was to switch out old code for displaying images on large and costly framebuffer hardware and replace it with a simple X11 display routine.

Table I lists the speed of transformations for which I recorded the performance in those long past days on the VAX-750 [2], and compares them with the speed of those same transformations on my desktop today. In the 23 seconds it took to evaluate the simplest possible expression, assigning the value middle gray to every pixel in a 512x512 grayscale image, the same code can now process over 23,000 of those same images. Or, it can process a 4096x4096 full color image with an alpha channel (256x more information per image) over 40 times faster than the small grayscale image. Compared to the on-the-fly compiler the difference is still very impressive, even for the more complex types of expressions.

For good measure, I've illustrated the result of the last transformation from Table I when it is applied to the color portrait you see in the masthead for this column in Figure 1. Figure 2 gives the more intriguing result of generating an image from scratch using Cartesian and polar coordinates, using a modulo and an exclusive-or operator. It can be endlessly fascinating to play with these types image operations, as I rediscovered with this newly ported version of this tool that predated Adobe Photoshop® by a good stretch.

The possibility remains to reintroduce an on-the-fly compiler, or more to boost performance further by using multiple cores or GPUs, both things that were not on the horizon when I first wrote this editor.

Figure 1, The result of transforming the photo at the top of this column with the last transformation from

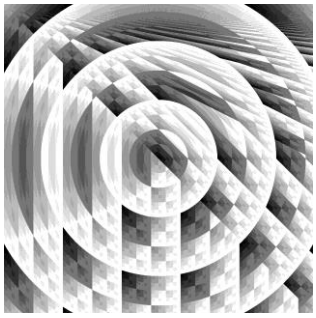

Figure 2, The result of evaluating *Pico* expression (x%y)^r, where x and y are the usual Cartesian coordinates and r is the radius in Polar coordinates.

Table 1, Relative Performance of image processing code, 1984 vs 2018

Transformation	1984		2018	
	Interpreter	On-the-fly Compiler	Interpreter	
	512x512 bw	512x512 bw	512x512 bw	4096x4096 rgba
new=128	23.6 s	5.3 s	0.00102 s	0.58 s
new=$1	43.3 s	5.9 s	0.00113 s	0.59 s
new=Z-old	59.9 s	6.5 s	0.00177 s	0.74 s
new=($1+$2)/2	105.9 s	9.3 s	0.00468 s	1.48 s
new=(x<X/2)?$1:$2	107.7 s	7.2 s	0.00531 s	0.98 s
new=($1<128)?Z-$1:$1[X-x,y]	304.5 s	10.8 s	0.00593 s	2.04 s

A more dangerous temptation however is to start packing more and features into this still very simple code, until it all becomes so bulky that it is unbearably slow again. If you wonder why your desktop system still feels so sluggish today, despite a few decades worth of massive speedups, it is precisely for that reason. Getting fast code sometimes requires us to step back and reconsider what is really the essential part of the problem we are trying to solve, and to push the rest aside.

Foundations

So other than the impressive gains in processor speed and memory sizes, did anything else change fundamentally in our field? A recent blog post [3] concluded that the traditional requirements for a career in software engineering, a solid foundation in algorithms and data structures and a proper understanding of computational complexity and logic, have slowly been replaced with a new focus on more social skills.

The blog's author reasons that the software engineer today is part of a much larger community of developers that share code on popular sites like *GitHub*. A developer must be able to collaborate within that larger community, which means an increasing emphasis on the ability to communicate effectively. In the view of the author, the ability to design and write new code from scratch is now less important than the ability to navigate existing code repositories and stitch together parts of solutions found there.

What I find curious about this view is that my experience is quite the opposite. Given that software dominates just about every aspect of our world today, and that even the tiniest glitch can have significant consequences, it is more important than ever that software engineers are well trained not just in the traditional topics of algorithm design, data structures, complexity, and logic, but also in newer issues like privacy and security. They should also have a good understanding of the new capabilities of formal methods for software design and verification. The types of code analysis that were infeasible on the machines from a few decades ago can now be performed efficiently and accurately. Rolling the dice on safety critical code is no longer an option for any professional software engineer.

There is indeed a large amount of open source code available today, but like everything else, not all of it is equally well written. This means that it is that much more important today to be able to evaluate the quality and security of code that you did not develop yourself, to check the choice of algorithms, and evaluate the code for any unnecessary execution bottlenecks.

Communication

Social skills are useful in any discipline, although the members of our community are generally not considered to be among the standard bearers here. I actually believe that the ability to interact and communicate effectively was considerably more important in those dark days before there was an internet to browse for answers to whatever type of problem you run into. In those early days you had to be able to find the actual person who likely knew the answers you needed, and talk to that guru face to face. The community of software developers is now large enough that there is always someone else who had exactly the same problem as you do now, and who put the answer on a website. We can now safely stay in our caves and type away, without needing to interact face to face with any other human beings, unless we want to of course. The good thing is that when we want to now, we can talk about other things than just code. Now that's progress!

References

[1] *Memoirs of a Computer Pioneer*, Maurice Wilkes, MIT Press, 1985, p. 145.

[2] *Pico - a picture editor*, G.J. Holzmann, AT&T Technical Journal, Vol. 66, No. 2, pp. 2-13, Apr. 1987. See also: http://spinroot.com/pico/

[3] *The Era of Hackers is Over*, Yegor Bugayenko, blog post April 23, 2018, https://cacm.acm.org/blogs/blog-cacm/227154-the-era-of-hackers-is-over/fulltext