Editor: **Gerard J. Holzmann**
Nimble Research
gholzmann@acm.org

Code Vault

Gerard J. Holzmann

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

> *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 [Electronic Delay Storage Automatic Calculator] 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.*[1]

The effectiveness with which I could hide subtle bugs in my programs wasn't hampered by the fact that I had oceans of time to reflect on the quality of my code 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 has changed since then? Depending on your point of view, nothing much has changed, or everything has changed. The part that didn't change much is that we still struggle with writing code that's robust enough to trust. The part that has changed dramatically is the performance of the hardware that runs our code.

The Trash-80

I bought my first PC, a TRS-80 Model II, sometimes lovingly called the Trash-80, in 1981. I paid about $3,800 for it, which was the equivalent of about $12,000 today. The Trash-80 ran at 4 MHz and came with just 64 Kbytes of RAM. It had no hard disk. All data was stored on 8″ floppy disks that could hold about 500 Kbytes each.

For the same price today, you can buy not one but four quad-core PCs, running at 4 GHz and with 32 Kbytes of RAM each. Together, that's about 16,000 times the processing capability and 500,000 times the amount of RAM of that old Trash-80. These PCs' hard drives are also 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 be sure, you don't notice that power too much when you use your PC today, but it's truly there. To see this more clearly, I revived an image-processing program called Pico that I wrote in 1984.[2] Pico defines new images by evaluating a user-defined expression once for every pixel in the original image. Pico expressions can, for instance, refer to values, Cartesian or polar coordinates, trigonometric functions, or parts of other images. That processing can be computationally 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-11/750 computer. I separately wrote an interpreter for the language that was more portable but was, of course, much slower. Even though the VAX-11/750 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, although 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 at http://spinroot.com/gerard/cnn.mpg.)

With a few small tweaks I ported the original code to work on my current desktop. Predictably, the interpreted version of the code is now magnificently faster than even the

0740-7459/18/$33.00 © 2018 IEEE

Table 1. The relative performance (s) of image-processing code, 1984 vs. 2018.

	1984		2018	
	Interpreter	On-the-fly compiler	Interpreter	
Transformation	512 × 512 B/W	512 × 512 B/W	512 × 512 B/W	4,096 × 4,096 RGBA
new=128	23.6	5.3	0.00102	0.58
new=$1	43.3	5.9	0.00113	0.59
new=Z-old	59.9	6.5	0.00177	0.74
new=($1+$2)/2	105.9	9.3	0.00468	1.48
new=(x<X/2)?$1:$2	107.7	7.2	0.00531	0.98
new=($1<128)?Z-$1:$1[X-x,y]	304.5	10.8	0.00593	2.04

FIGURE 1. The result of performing the last transformation from Table 1 on the photo at the beginning of this article.

compiled code had been in the 1980s. The main tweak was to replace old code for displaying images on large, costly framebuffer hardware with a simple X11 display routine.

Table 1 compares the speed of transformations for which I recorded the performance in those long-past days on the VAX-11/750[2] with the speed of those transformations on my current desktop. In the 23 seconds it once took to evaluate the simplest-possible expression, assigning the value 128 (for middle gray) to every pixel in a 512 × 512 grayscale image, the same code can now process more than 23,000 of those images. Or, it can process a 4,096 × 4,096 full-color image with an alpha channel (256× more information per image) more than 40 times faster than the small grayscale image. Compared to the on-the-fly compiler, the difference is still impressive, even for the more complex expressions.

For good measure, Figure 1 illustrates the result of applying the last transformation from Table 1 to my color portrait at the beginning of this article. Figure 2 gives the more intriguing result of generating an image from scratch using Cartesian and polar coordinates and using a modulo and an exclusive-or operator. It can be endlessly fascinating to play with these types of image operations, as I rediscovered with this newly ported version of this tool, which predates Adobe Photoshop by a good stretch.

The possibility remains of reintroducing an on-the-fly compiler or further boosting performance by using multiple cores or GPUs, both things that weren't on the horizon when I first wrote this image editor.

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's unbearably slow again. If you wonder why your desktop system still feels so sluggish, despite a few decades worth of massive speedups, it's precisely for that reason. Getting fast code sometimes requires us to step back, reconsider what's really the essential part of the problem we're trying to solve, and push the rest aside.

Foundations

So, other than the impressive gains in processor speed and memory size, has anything else in our field changed fundamentally? In a recent blog post, Yegor Bugayenko concluded that the traditional requirements for a software engineering career—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.[3]

ABOUT THE AUTHOR

GERARD J. HOLZMANN works on developing stronger methods for the design and analysis of safety-critical software as a consultant and researcher at Nimble Research. Contact him at gholzmann@acm.org.

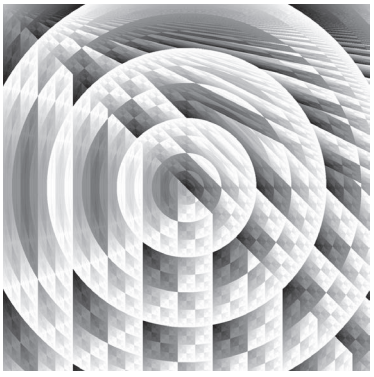

FIGURE 2. The result of evaluating the Pico expression **(x%y)^r**, where **x** and **y** are the usual Cartesian coordinates and **r** is the radius in polar coordinates.

Bugayenko reasons that the software engineer today is part of a much larger community of developers who share code on popular sites such as GitHub. A developer must be able to collaborate within that larger community, which means an increasing emphasis on the ability to communicate effectively. In Bugayenko's view, 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.

My experience has been quite the opposite. Software today dominates just about every aspect of our world, and even the tiniest glitch can have significant consequences. So, it's more important than ever that software engineers are well trained, in not just the traditional topics of algorithm design, data structures, complexity, and logic but also newer issues such as 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 of 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.

Much open source code is indeed available today, but like everything else, not all of it is well written. This means that it's that much more important to be able to evaluate the quality and security of code you didn't develop yourself, 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 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 knew the answers you needed, and talk to that guru face to face.

The community of software developers is now large enough that there's always someone else who has had the problem you're having and has put the solution 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, we can talk about things other than just code. Now that's progress!

References

1. M. Wilkes, *Memoirs of a Computer Pioneer*, MIT Press, 1985, p. 145.
2. G.J. Holzmann, "Pico—a Picture Editor," *AT&T Technical J.*, vol. 66, no. 2, 1987, pp. 2–13; http://spinroot.com/pico/atttj.pdf.
3. Y. Bugayenko, "The Era of Hackers Is Over," blog, 23 Apr. 2018; https://cacm.acm.org/blogs/blog-cacm/227154-the-era-of-hackers-is-over/fulltext.