Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

The Tools We Use

Diomidis Spinellis

It is impossible to sharpen a pencil with a blunt ax. It is equally vain to try to do it with ten blunt axes instead. —Edsger W. Dijkstra

What's the state of the art in the tools we use to build software? To answer this question, I let a powerful server build from source code about 7,000 open source packages over a period of a month. The packages I built form a subset of the FreeBSD operating system ports collection, comprising a wide spectrum of application domains: from desktop utilities and biology applications to databases and development tools. The collection is representative of modern software because, unlike say a random sample of SourceForge.net projects, FreeBSD developers have found these programs useful enough to port to FreeBSD.

The build process involves fetching each application's source code bundle from the Internet, patching it for FreeBSD, and compiling the source code into executable programs or libraries. Over the one-month period, I also set up the operating system to write an accounting record for each command it executed. I then tallied the CPU times of the 144 million records corresponding to the work to get a picture of how our software builds exploit the power of modern gigahertz processors.

Figure 1 shows the time breakup of the commands that took more than one percent of the 18 days of accumulated CPU time. The picture isn't pretty. First of all, variety in our tools ecosystem appears to be extremely limited. A full 94 percent of the CPU time is taken by only 10 commands, of which six drive the unpackaging process and are not directly part of each package's compilation. This wouldn't be so bad if the remaining commands, which apparently represent the state of the art in software-building tools, were based on shiny modern ideas. But this isn't the case. Three of the tools have their roots in the 1970s and 1980s: C (1978), make (1979), and C++ (1983). As you can see in the figure, the compilation of C and C++ code takes up the lion's share of the building effort.

Is that so?

I hear you arguing that studying the tools used at build time is disingenuous because most improvements have happened in the environment where the software is written, not in the tools that compile the software. And that nowadays, many developers code using advanced IDEs (integrated development environments) that integrate design, coding, debugging, performance analysis, and testing—a sure sign of progress.

I beg to differ. Using a shiny IDE on top of 1970s technologies is equivalent to wearing an iPod while ox-plowing: the work becomes less burdensome, but we're unlikely to reap substantial productivity improvements from such a change.

The most important (perhaps only important) artifact of software development is the source code. This is where we store all knowledge acquired during a system's design, development, and subsequent evolution. Specifications and design documents (when they exist) quickly become out of date, knowledgeable developers switch jobs or retire, and many teams don't document or enforce development pro-

 0740-7459/07/$25.00 © 2007 IEEE

cesses. This is why organizations often stumble when they try to replace a legacy system. All they have is code, and legacy code is (as we saw in the May/June column) often a mess. Therefore, by looking at the tools we use to convert source code into executable format, we get an accurate picture of the abstraction level that programmers will face during construction and maintenance (where the largest chunk of software development effort takes place). We'll see order-of-magnitude productivity improvements only when we raise our code's level of abstraction.

Some of you might also argue that—because programs written in Java, C#, and scripting languages don't require the operating system-specific compilation step that I measured—I haven't taken into account the large amount of software written in these languages. This is true, but Java and C# still use the same data types and flow-control constructs as C++. They're also still niche players in some important markets: system software, desktop applications, and embedded systems. Where scripting languages are used (think of Ruby on Rails for Web site building), they offer big productivity gains by raising the level of abstraction and offering domain-specific functionality. However, it's not yet clear how we can apply these gains to other fields.

Ox-plowing revisited

So what would make me happy? For a start, I'd like to see a tractor replace the ox plow. I'd like to see large chunks of a build's CPU time going to compilers for languages with at least an order of magnitude more expressive power than C and C++. Some candidates that can offer us higher levels of abstraction are domain-specific languages, general-purpose declarative languages such as Haskell, and executable UML (Unified Modeling Language). These will let our computers work harder to understand our higher-level programs, thus trading CPU power for human intellect.

For instance, the data I collected reflects the higher level of abstraction of C++ over C. Each invocation of the C++ compiler consumes on average 1.6 s of CPU time—many times more than the 0.17 s of each C compiler run. Accumulated over more than a million executions, this is a lot of processor time. However, nowadays CPU power is a resource that (a) we can afford to use and (b) we can't afford not to use.

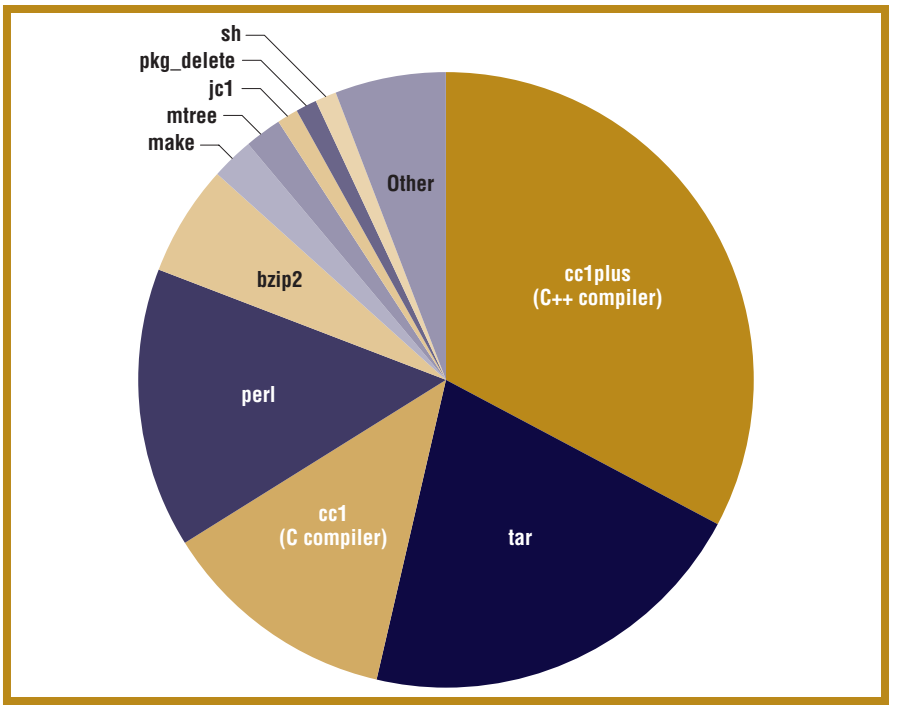

Figure 1. Composition of the FreeBSD ports build effort.

I'd also like to see the use of higher-yield grains, fertilizer, and high-tech irrigation; in our case, ancillary tools that help us build reliable, secure, efficient, usable, and maintainable software. These could ensure, for example, that the locks in my software are correctly paired or that the implementation satisfies a formally described specification. I'd also like to see in the top places of the build-effort breakup a single testing framework, a style checker, and a bug finder. Although you could argue that all these tools will only be used during development, I think that a clean bill of health from them during each build would help us focus on reliability and readability. After all, we don't disable the compiler's type-checking functionality when performing a release build. In an ideal world, one or two tools dominating each category would let developers learn one of them and apply it in all their work.

Two success stories of the 1970s that raised the level of abstraction for a specific domain are Stephen Johnson's parser generator YACC (yet another compiler compiler) and Michael Lesk's lexical analyzer generator Lex. These two tools and the theory behind them transformed the task of writing a compiler from wizardry into a standard rite of passage for computer science undergraduates. It's been a long time since they appeared, and it's high time to come up with similarly revolutionary new tools. So, the next time you design a system's architecture, think which tools can give you the highest expressive power. Look around, ask your tool vendors, experiment, and invent. Don't just settle for the bland comfort of repolished 1970s technologies.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.

Post your comments online by visiting the column's blog: **www.spinellis.gr/tools**