Interview with Donald Knuth

By Donald E. Knuth,Andrew Binstock

Date: Apr 25, 2008

Return to the article

Andrew Binstock and Donald Knuth converse on the success of open source, the problem with multicore architecture, the disappointing lack of interest in literate programming, the menace of reusable code, and that urban legend about winning a programming contest with a single compilation.

Andrew Binstock: You are one of the fathers of the open-source revolution, even if you aren't widely heralded as such. You previously have stated that you released TeX as open source because of the problem of proprietary implementations at the time, and to invite corrections to the code—both of which are key drivers for open-source projects today. Have you been surprised by the success of open source since that time?

Donald Knuth: The success of open source code is perhaps the only thing in the computer field that *hasn't* surprised me during the past several decades. But it still hasn't reached its full potential; I believe that open-source programs will begin to be completely dominant as the economy moves more and more from products towards services, and as more and more volunteers arise to improve the code.

For example, open-source code can produce thousands of binaries, tuned perfectly to the configurations of individual users, whereas commercial software usually will exist in only a few versions. A generic binary executable file must include things like inefficient "sync" instructions that are totally inappropriate for many installations; such wastage goes away when the source code is highly configurable. This should be a huge win for open source.

Yet I think that a few programs, such as Adobe Photoshop, will always be superior to competitors like the Gimp—for some reason, I really don't know why! I'm quite willing to pay good money for really good software, if I believe that it has been produced by the best programmers.

Remember, though, that my opinion on economic questions is highly suspect, since I'm just an educator and scientist. I understand almost nothing about the marketplace.

Andrew: A story states that you once entered a programming contest at Stanford (I believe) and you submitted the winning entry, which worked correctly after a *single* compilation. Is this story true? In that vein, today's developers frequently build programs writing small code increments followed by immediate compilation and the creation and running of unit tests. What are your thoughts on this approach to software development?

Donald: The story you heard is typical of legends that are based on only a small kernel of truth. Here's what actually happened: John McCarthy decided in 1971 to have a Memorial Day Programming Race. All of the contestants except me worked at his AI Lab up in the hills above Stanford, using the WAITS time-sharing system; I was down on the main campus, where the only computer available to me was a mainframe for which I had to punch cards and submit them for processing in batch mode. I used Wirth's ALGOL W system (the predecessor of Pascal). My program *didn't* work the first time, but fortunately I could use Ed Satterthwaite's excellent offline debugging system for ALGOL W, so I needed only two runs. Meanwhile, the folks using WAITS couldn't get enough machine cycles because their machine was so overloaded. (I think that the second-place finisher, using that "modern" approach, came in about an hour after I had submitted the winning entry with old-fangled methods.) It wasn't a fair contest.

As to your real question, the idea of immediate compilation and "unit tests" appeals to me only rarely, when I'm feeling my way in a totally unknown environment and need feedback about what works and what doesn't. Otherwise, lots of time is wasted on activities that I simply never need to perform or even think about. Nothing needs to be "mocked up."

Andrew: One of the emerging problems for developers, especially client-side developers, is changing their thinking to write programs in terms of threads. This concern, driven by the advent of inexpensive multicore PCs, surely will require that many algorithms be recast for multithreading, or at least to be thread-safe. So far, much of the work you've published for Volume 4 of The Art of Computer Programming (*TAOCP*) doesn't seem to touch on this dimension. Do you expect to enter into problems of concurrency and parallel programming in upcoming work, especially since it would seem to be a natural fit with the combinatorial topics you're currently working on?

Donald: The field of combinatorial algorithms is so vast that I'll be lucky to pack its *sequential* aspects into three or four physical volumes, and I don't think the sequential methods are ever going to be unimportant. Conversely, the half-life of parallel techniques is

very short, because hardware changes rapidly and each new machine needs a somewhat different approach. So I decided long ago to stick to what I know best. Other people understand parallel machines much better than I do; programmers should listen to them, not me, for guidance on how to deal with simultaneity.

Andrew: Vendors of multicore processors have expressed frustration at the difficulty of moving developers to this model. As a former professor, what thoughts do you have on this transition and how to make it happen? Is it a question of proper tools, such as better native support for concurrency in languages, or of execution frameworks? Or are there other solutions?

Donald: I don't want to duck your question entirely. I might as well flame a bit about my personal unhappiness with the current trend toward multicore architecture. To me, it looks more or less like the hardware designers have run out of ideas, and that they're trying to pass the blame for the future demise of Moore's Law to the software writers by giving us machines that work faster only on a few key benchmarks! I won't be surprised at all if the whole multithreading idea turns out to be a flop, worse than the "Itanium" approach that was supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write.

Let me put it this way: During the past 50 years, I've written well over a thousand programs, many of which have substantial size. I can't think of even *five* of those programs that would have been enhanced noticeably by parallelism or multithreading. Surely, for example, multiple processors are no help to TeX.[1]

How many programmers do you know who are enthusiastic about these promised machines of the future? I hear almost nothing but grief from software people, although the hardware folks in our department assure me that I'm wrong.

I know that important applications for parallelism exist—rendering graphics, breaking codes, scanning images, simulating physical and biological processes, etc. But all these applications require dedicated code and special-purpose techniques, which will need to be changed substantially every few years.

Even if I knew enough about such methods to write about them in *TAOCP*, my time would be largely wasted, because soon there would be little reason for anybody to read those parts. (Similarly, when I prepare the third edition of Volume 3 I plan to rip out much of the material about how to sort on magnetic tapes. That stuff was once one of the hottest topics in the whole software field, but now it largely wastes paper when the book is printed.)

The machine I use today has dual processors. I get to use them both only when I'm running two independent jobs at the same time; that's nice, but it happens only a few minutes every week. If I had four processors, or eight, or more, I still wouldn't be any better off, considering the kind of work I do—even though I'm using my computer almost every day during most of the day. So why should I be so happy about the future that hardware vendors promise? They think a magic bullet will come along to make multicores speed up my kind of work; I think it's a pipe dream. (No—that's the wrong metaphor! "Pipelines" actually work for me, but threads don't. Maybe the word I want is "bubble.")

From the opposite point of view, I do grant that web browsing probably will get better with multicores. I've been talking about my technical work, however, not recreation. I also admit that I haven't got many bright ideas about what I wish hardware designers would provide instead of multicores, now that they've begun to hit a wall with respect to sequential computation. (But my MMIX design contains several ideas that would substantially improve the current performance of the kinds of programs that concern me most—at the cost of incompatibility with legacy x86 programs.)

Andrew: One of the few projects of yours that hasn't been embraced by a widespread community is literate programming. What are your thoughts about why literate programming didn't catch on? And is there anything you'd have done differently in retrospect regarding literate programming?

Donald: Literate programming is a very personal thing. I think it's terrific, but that might well be because I'm a very strange person. It has tens of thousands of fans, but not millions.

In my experience, software created with literate programming has turned out to be significantly better than software developed in more traditional ways. Yet ordinary software is usually okay—I'd give it a grade of C (or maybe C++), but not F; hence, the traditional methods stay with us. Since they're understood by a vast community of programmers, most people have no big incentive to change, just as I'm not motivated to learn Esperanto even though it might be preferable to English and German and French and Russian (if everybody switched).

Jon Bentley probably hit the nail on the head when he once was asked why literate programming hasn't taken the whole world by storm. He observed that a small percentage of the world's population is good at programming, and a small percentage is good at writing; apparently I am asking everybody to be in both subsets.

Yet to me, literate programming is certainly the most important thing that came out of the TeX project. Not only has it enabled me to write and maintain programs faster and more reliably than ever before, and been one of my greatest sources of joy since the 1980s—it has actually been *indispensable* at times. Some of my major programs, such as the MMIX meta-simulator, could not have been written

with any other methodology that I've ever heard of. The complexity was simply too daunting for my limited brain to handle; without literate programming, the whole enterprise would have flopped miserably.

If people do discover nice ways to use the newfangled multithreaded machines, I would expect the discovery to come from people who routinely use literate programming. Literate programming is what you need to rise above the ordinary level of achievement. But I don't believe in forcing ideas on anybody. If literate programming isn't your style, please forget it and do what you like. If nobody likes it but me, let it die.

On a positive note, I've been pleased to discover that the conventions of CWEB are already standard equipment within preinstalled software such as Makefiles, when I get off-the-shelf Linux these days.

Andrew: In Fascicle 1 of Volume 1, you reintroduced the MMIX computer, which is the 64-bit upgrade to the venerable MIX machine comp-sci students have come to know over many years. You previously described MMIX in great detail in MMIXware. I've read portions of both books, but can't tell whether the Fascicle updates or changes anything that appeared in MMIXware, or whether it's a pure synopsis. Could you clarify?

Donald: Volume 1 Fascicle 1 is a programmer's introduction, which includes instructive exercises and such things. The MMIXware book is a detailed reference manual, somewhat terse and dry, plus a bunch of literate programs that describe prototype software for people to build upon. Both books define the same computer (once the errata to MMIXware are incorporated from my website). For most readers of *TAOCP*, the first fascicle contains everything about MMIX that they'll ever need or want to know.

I should point out, however, that MMIX isn't a single machine; it's an architecture with almost unlimited varieties of implementations, depending on different choices of functional units, different pipeline configurations, different approaches to multiple-instruction-issue, different ways to do branch prediction, different cache sizes, different strategies for cache replacement, different bus speeds, etc. Some instructions and/or registers can be emulated with software on "cheaper" versions of the hardware. And so on. It's a test bed, all simulatable with my meta-simulator, even though advanced versions would be impossible to build effectively until another five years go by (and then we could ask for even further advances just by advancing the meta-simulator specs another notch).

Suppose you want to know if five separate multiplier units and/or three-way instruction issuing would speed up a given MMIX program. Or maybe the instruction and/or data cache could be made larger or smaller or more associative. Just fire up the meta-simulator and see what happens.

Andrew: As I suspect you don't use unit testing with MMIXAL, could you step me through how you go about making sure that your code works correctly under a wide variety of conditions and inputs? If you have a specific work routine around verification, could you describe it?

Donald: Most examples of machine language code in *TAOCP* appear in Volumes 1-3; by the time we get to Volume 4, such low-level detail is largely unnecessary and we can work safely at a higher level of abstraction. Thus, I've needed to write only a dozen or so MMIX programs while preparing the opening parts of Volume 4, and they're all pretty much toy programs—nothing substantial. For little things like that, I just use informal verification methods, based on the theory that I've written up for the book, together with the MMIXAL assembler and MMIX simulator that are readily available on the Net (and described in full detail in the MMIXware book).

That simulator includes debugging features like the ones I found so useful in Ed Satterthwaite's system for ALGOL W, mentioned earlier. I always feel quite confident after checking a program with those tools.

Andrew: Despite its formulation many years ago, TeX is still thriving, primarily as the foundation for LaTeX. While TeX has been effectively frozen at your request, are there features that you would want to change or add to it, if you had the time and bandwidth? If so, what are the major items you add/change?

Donald: I believe changes to TeX would cause much more harm than good. Other people who want other features are creating their own systems, and I've always encouraged further development—except that nobody should give their program the same name as mine. I want to take permanent responsibility for TeX and Metafont, and for all the nitty-gritty things that affect existing documents that rely on my work, such as the precise dimensions of characters in the Computer Modern fonts.

Andrew: One of the little-discussed aspects of software development is how to do design work on software in a completely new domain. You were faced with this issue when you undertook TeX: No prior art was available to you as source code, and it was a domain in which you weren't an expert. How did you approach the design, and how long did it take before you were comfortable entering into the coding portion?

Donald: That's another good question! I've discussed the answer in great detail in Chapter 10 of my book Literate Programming, together with Chapters 1 and 2 of my book Digital Typography. I think that anybody who is really interested in this topic will enjoy reading those chapters. (See also *Digital Typography* Chapters 24 and 25 for the complete first and second drafts of my initial design of TeX in 1977.)

Andrew: The books on TeX and the program itself show a clear concern for limiting memory usage—an important problem for systems of that era. Today, the concern for memory usage in programs has more to do with cache sizes. As someone who has designed a processor in software, the issues of cache-aware and cache-oblivious algorithms surely must have crossed your radar screen. Is the role of processor caches on algorithm design something that you expect to cover, even if indirectly, in your upcoming work?

Donald: I mentioned earlier that MMIX provides a test bed for many varieties of cache. And it's a software-implemented machine, so we can perform experiments that will be repeatable even a hundred years from now. Certainly the next editions of Volumes 1-3 will discuss the behavior of various basic algorithms with respect to different cache parameters.

In Volume 4 so far, I count about a dozen references to cache memory and cache-friendly approaches (not to mention a "memo cache," which is a different but related idea in software).

Andrew: What set of tools do you use today for writing *TAOCP*? Do you use TeX? LaTeX? CWEB? Word processor? And what do you use for the coding?

Donald: My general working style is to write everything first with pencil and paper, sitting beside a big wastebasket. Then I use Emacs to enter the text into my machine, using the conventions of TeX. I use tex, dvips, and gv to see the results, which appear on my screen almost instantaneously these days. I check my math with Mathematica.

I program every algorithm that's discussed (so that I can thoroughly understand it) using CWEB, which works splendidly with the GDB debugger. I make the illustrations with MetaPost (or, in rare cases, on a Mac with Adobe Photoshop or Illustrator). I have some homemade tools, like my own spell-checker for TeX and CWEB within Emacs. I designed my own bitmap font for use with Emacs, because I hate the way the ASCII apostrophe and the left open quote have morphed into independent symbols that no longer match each other visually. I have special Emacs modes to help me classify all the tens of thousands of papers and notes in my files, and special Emacs keyboard shortcuts that make bookwriting a little bit like playing an organ. I prefer rxvt to xterm for terminal input. Since last December, I've been using a file backup system called backupfs, which meets my need beautifully to archive the daily state of every file.

According to the current directories on my machine, I've written 68 different CWEB programs so far this year. There were about 100 in 2007, 90 in 2006, 100 in 2005, 90 in 2004, etc. Furthermore, CWEB has an extremely convenient "change file" mechanism, with which I can rapidly create multiple versions and variations on a theme; so far in 2008 I've made 73 variations on those 68 themes. (Some of the variations are quite short, only a few bytes; others are 5KB or more. Some of the CWEB programs are quite substantial, like the 55-page BDD package that I completed in January.) Thus, you can see how important literate programming is in my life.

I currently use Ubuntu Linux, on a standalone laptop—it has no Internet connection. I occasionally carry flash memory drives between this machine and the Macs that I use for network surfing and graphics; but I trust my family jewels only to Linux. Incidentally, with Linux I much prefer the keyboard focus that I can get with classic FVWM to the GNOME and KDE environments that other people seem to like better. To each their own.

Andrew: You state in the preface of Fascicle 0 of Volume 4 of *TAOCP* that Volume 4 surely will comprise three volumes and possibly more. It's clear from the text that you're really enjoying writing on this topic. Given that, what is your confidence in the note posted on the *TAOCP* website that Volume 5 will see light of day by 2015?

Donald: If you check the Wayback Machine for previous incarnations of that web page, you will see that the number 2015 has not been constant.

You're certainly correct that I'm having a ball writing up this material, because I keep running into fascinating facts that simply can't be left out—even though more than half of my notes don't make the final cut.

Precise time estimates are impossible, because I can't tell until getting deep into each section how much of the stuff in my files is going to be really fundamental and how much of it is going to be irrelevant to my book or too advanced. A lot of the recent literature is academic one-upmanship of limited interest to me; authors these days often introduce arcane methods that outperform the simpler techniques only when the problem size exceeds the number of protons in the universe. Such algorithms could never be important in a real computer application. I read hundreds of such papers to see if they might contain nuggets for programmers, but most of them wind up getting short shrift.

From a scheduling standpoint, all I know at present is that I must someday digest a huge amount of material that I've been collecting and filing for 45 years. I gain important time by working in batch mode: I don't read a paper in depth until I can deal with dozens of others on the same topic during the same week. When I finally am ready to read what has been collected about a topic, I might find out that I can zoom ahead because most of it is eminently forgettable for my purposes. On the other hand, I might discover that it's fundamental and deserves weeks of study; then I'd have to edit my website and push that number 2015 closer to infinity.

Andrew: In late 2006, you were diagnosed with prostate cancer. How is your health today?

Donald: Naturally, the cancer will be a serious concern. I have superb doctors. At the moment I feel as healthy as ever, modulo being 70 years old. Words flow freely as I write *TAOCP* and as I write the literate programs that precede drafts of *TAOCP*. I wake up in the morning with ideas that please me, and some of those ideas actually please me also later in the day when I've entered them into my computer.

On the other hand, I willingly put myself in God's hands with respect to how much more I'll be able to do before cancer or heart disease or senility or whatever strikes. If I should unexpectedly die tomorrow, I'll have no reason to complain, because my life has been incredibly blessed. Conversely, as long as I'm able to write about computer science, I intend to do my best to organize and expound upon the tens of thousands of technical papers that I've collected and made notes on since 1962.

Andrew: On your website, you mention that the Peoples Archive recently made a series of videos in which you reflect on your past life. In segment 93, "Advice to Young People," you advise that people shouldn't do something simply because it's trendy. As we know all too well, software development is as subject to fads as any other discipline. Can you give some examples that are currently in vogue, which developers shouldn't adopt simply because they're currently popular or because that's the way they're currently done? Would you care to identify important examples of this outside of software development?

Donald: Hmm. That question is almost contradictory, because I'm basically advising young people to listen to themselves rather than to others, and I'm one of the others. Almost every biography of every person whom you would like to emulate will say that he or she did many things against the "conventional wisdom" of the day.

Still, I hate to duck your questions even though I also hate to offend other people's sensibilities—given that software methodology has always been akin to religion. With the caveat that there's no reason anybody should care about the opinions of a computer scientist/mathematician like me regarding software development, let me just say that almost everything I've ever heard associated with the term "extreme programming" sounds like exactly the wrong way to go...with one exception. The exception is the idea of working in teams and reading each other's code. That idea is crucial, and it might even mask out all the terrible aspects of extreme programming that alarm me.

I also must confess to a strong bias against the fashion for reusable code. To me, "re-editable code" is much, much better than an untouchable black box or toolkit. I could go on and on about this. If you're totally convinced that reusable code is wonderful, I probably won't be able to sway you anyway, but you'll never convince me that reusable code isn't mostly a menace.

Here's a question that you may well have meant to ask: Why is the new book called Volume 4 Fascicle 0, instead of Volume 4 Fascicle 1? The answer is that computer programmers will understand that I wasn't ready to begin writing Volume 4 of *TAOCP* at its true beginning point, because we know that the initialization of a program can't be written until the program itself takes shape. So I started in 2005 with Volume 4 Fascicle 2, after which came Fascicles 3 and 4. (Think of *Star Wars*, which began with Episode 4.)

Finally I was psyched up to write the early parts, but I soon realized that the introductory sections needed to include much more stuff than would fit into a single fascicle. Therefore, remembering Dijkstra's dictum that counting should begin at 0, I decided to launch Volume 4 with Fascicle 0. Look for Volume 4 Fascicle 1 later this year.

Footnote

[1] My colleague Kunle Olukotun points out that, if the usage of TeX became a major bottleneck so that people had a dozen processors and really needed to speed up their typesetting terrifically, a super-parallel version of TeX could be developed that uses "speculation" to typeset a dozen chapters at once: Each chapter could be typeset under the assumption that the previous chapters don't do anything strange to mess up the default logic. If that assumption fails, we can fall back on the normal method of doing a chapter at a time; but in the majority of cases, when only normal typesetting was being invoked, the processing would indeed go 12 times faster. Users who cared about speed could adapt their behavior and use TeX in a disciplined way.

Andrew Binstock is the principal analyst at Pacific Data Works. He is a columnist for SD Times and senior contributing editor for InfoWorld magazine. His blog can be found at: http://binstock.blogspot.com.

© 2008 Pearson Education, Inc. Informit. All rights reserved.
800 East 96th Street Indianapolis, Indiana 46240