Interview with Timothy Roscoe

RIK FARROW

Timothy Roscoe (aka Mothy) is a full Professor in the Systems Group of the Computer Science Department at ETH Zurich. He received a PhD from the Computer Laboratory of the University of Cambridge, where he was a principal designer and builder of the Nemesis operating system, as well as working on the Wanda microkernel and Pandora multimedia system. After three years working on Web-based collaboration systems at a startup company in North Carolina, Mothy joined Sprint's Advanced Technology Lab in Burlingame, California, working on cloud computing and network monitoring. He then joined Intel Research at the University of California, Berkeley in April 2002, as a principal architect of PlanetLab, an open, shared platform for developing and deploying planetary-scale services. In September 2006, he spent four months as a visiting researcher in the Embedded and Real-Time Operating Systems group at National ICT Australia in Sydney before joining ETH Zurich in January 2007. His current research interests include network architecture and the Barrelfish multicore research operating system. He was recently elected Fellow of the ACM for contributions to operating systems and networking research.

Rik is the editor of *;login:*.
rik@usenix.org

I interviewed Mothy Roscoe over six years ago, after I became aware of the Barrelfish operating system [1]. Barrelfish was designed specifically for the new generation of many-core CPU chips, and Mothy was one of the people on the team who built the OS [2].

Since that time, Intel released the Single-chip Cloud Computer [3], and the Barrelfish team created a port for this unusual chip. The SCC was designed as a research microprocessor with 48 cores and a message passing system. Today, Intel has moved toward the Xeon Phi, a chip that sits on the PCI express bus, exposes many processing cores, and provides a Linux-based stack for communicating with the Phi. The Phi has 68 cores and is designed for HPC.

A number of sophisticated 64-bit ARM multicore server chips are appearing that look like a great fit for Barrelfish. Mothy told me that Barrelfish does support some of these, but actually getting documentation from vendors can prove difficult in some cases.

I'd had the opportunity to chat with Mothy during many systems conference luncheons, and I wanted to follow up on some of those conversations and share them.

Mothy is co-chair of OSDI '16, along with Kim Keeton of HP Labs and HP Enterprise. The discussion of systems conference program committees that I mention in "Musings" (page 2) occurred after we had conducted this interview.

Rik: Please update us on what has been happening with Barrelfish since we last talked (April 2010).

Mothy: Quite a lot—it's been an interesting journey! To some extent the focus of the project has changed, but it's remarkable how many of our original goals, and our conjectures about the challenges of future hardware, turned out to be on the mark.

When we started in 2007, we thought that the hard problems in OS design were scaling, heterogeneity, and diversity. Those challenges led us to the multikernel model: by structuring the OS as a distributed set of cores communicating with messages, we handle heterogeneity, lack of cache coherence and/or shared memory, dynamic cores, etc. in a single elegant framework.

The one challenge the multikernel model does not solve directly is scalability. However, we find it a lot easier to think about scaling in the context of message passing rather than shared memory—and you also see this trend in other areas like HPC and large-scale "Big Data" applications as well.

We've learned a lot about modern hardware so far in building Barrelfish and a lot about how to write a modern OS for that hardware. There is a huge difference between building a real OS from scratch with a different design versus tweaking existing Linux or Windows kernels. We were lucky to have the opportunity to pull off a sufficiently large engineering project over a long period of time. For us, the big payoff happened after six years on when companies started to get very interested in Barrelfish—that's a long time by the standards of most university research projects.

Along the way, we've published a lot of papers and graduated a bunch of PhD students. Academically, it's been a huge success so far. A common criticism of big projects like this is that they don't generate enough papers in the modern academic climate—several colleagues (not at ETH) have suggested that it's better to focus on smaller projects and higher paper counts so that students find it easier to get academic jobs.

I want the students to publish, but I feel strongly that paper mills are not the only way to run a university research group, and paper-driven research isn't a good way to have long-term impact (it's also not ETH's mission). Simon Peter, the PhD student who wrote the first line of code of Barrelfish, is now a professor at University of Texas, Austin. He also won Best Paper at OSDI 2014 with Arrakis [5], a variant of Barrelfish, so we seem to be doing OK nine years on.

In building Barrelfish, we also made a bunch of decisions that were somewhat arbitrary at the time—we really felt they were a good way to build an OS, but we didn't view them as essential to the research. In retrospect, however, many of these choices were highly fortuitous.

For example, we adopted a capability model (which we extended from seL4) for managing all memory resources, and this turned out to have profound consequences much later. It's only now that we're finding this is a great match for very large main memories, like HP's "The Machine" project.

The big change came in about 2012. Barrelfish had become a useful vehicle for OS research and teaching at ETH, despite the sometimes uphill struggle to convince people that "it's not Linux" was not a showstopper for a realistic OS, even in the research community.

At this point, we started being contacted by companies (hardware vendors and others) who had become interested in Barrelfish. They were building hardware that didn't look like a 1980s VAX, and it was dawning on them that Linux wasn't a good fit for these new hardware designs.

It was this interest that convinced us to keep going with the project rather than move on to new things. It was great to feel that the ideas that Paul Barham, Rebecca Isaacs, Simon Peter, Andrew Baumann, myself, and others had started with back in 2007 had finally been vindicated.

We've now got a great core development group at ETH (including one full-time software engineer and, hopefully, more in the future), we accept external contributions (both patches and pull requests on GitHub), and we're looking to support more hardware as we work with more vendors.

We've also got a ton of ideas, thoughts, war stories, etc. that we'd really like to talk about, but which are a bit of a challenge to fit into traditional computer science conference publishing or system documentation—we're starting a blog of these to see what interest there is out there.

Rik: What's the main research direction of the project now?

Mothy: One of the biggest driving challenges right now is hardware complexity. Modern computer hardware is incredibly complex in terms of peripheral devices, interconnects, memory hierarchies, etc.

To take one example: most people think they know what a physical address is; every memory cell or device register sits at a unique physical address, which is what you can use to access that location by having the MMU put out that address on the memory interconnect. Unfortunately, this just isn't true any more.

Instead, within a single machine (even a single SoC), different cores will see the same memory location appear at different physical addresses. Each core will only be able to address a subset of all the memory locations (storage cells or device registers), and these subsets are different, but not necessarily disjoint: they intersect in interesting ways. The access latency and bandwidth to a given location also varies, of course. For any pair of cores in the machine, the same location might be at the same address, or different addresses, or only addressable from one core, and might or might not be coherent in cache.

This happens on modern PCs, phone SoCs, and pretty much any other piece of "mainstream" hardware. We also see analogous complexity and diversity across systems in DMA engines, interrupt routing, network interfaces, and so on.

So what do we do? Nobody has a really good, crisp, formal description of what, say, a physical address is these days. The closest you find in traditional OS designs is a device tree (http://www.devicetree.org/), but device trees are really a file format—it doesn't capture semantics in a way you can make strong statements about.

We'd like to be able to put the hardware/software interface that OSes must use on a much more sound formal footing. What we're doing is writing semi-formal descriptions of all the memory systems, interrupt routing models, interconnect topologies that we can find, and then devising representations of these in subsets of first-order logic.

The short-term benefit of this is that Barrelfish can easily adapt to new hardware online—we've always programmed PCIe buses using Constraint Logic Programming in Barrelfish at boot time, and we're doing a lot more in this line. It's just a much easier way to engineer a more portable, general purpose OS for modern hardware. And in the medium term, of course, we can use these models and representations to provide a basis for formal verification of system software.

The ultimate goal, however, is a "software-centric" description of hardware that allows us as OS designers to talk about what kinds of hardware designs are "tasteful" as far as the OS is concerned, and what hardware design patterns are not. OS people are pretty good at critiquing bad hardware designs (and there are many!), but hardware designers pretty much ignore OS folks, and one reason for this is that we're not nearly as good at saying up front what the rules and patterns are for good hardware interface design.

This sounds philosophical, but there's very little academic work on this, and it's amazing how much traction we've got for these long-term ideas from industry partners. It's fundamental work that also has direct short-term applicability in industry.

Rik: Any plans on porting Barrelfish to more ARM-based server SoCs? It sounds like some of them might be vaguely similar to the Intel cloud-on-a-chip.

Mothy: We're always interested in future hardware platforms, if we can find out anything about them :-). If you're writing an operating system for new hardware in an academic environment, a constant headache is getting documentation. Pretty much any hardware vendor assumes you want to run Linux on their chip, and that's it, so why would you want documentation?

We sign NDAs, and sometimes this really helps, and sometimes it doesn't. But even finding someone to talk to about getting documentation is often the challenge.

Sometimes it works out great—Intel actually approached us before the Single-chip Cloud Computer was announced some years back, and my student did a port to the SCC that was ready when they launched. ARM, Intel, and some vendors (such as TI) released extremely detailed documentation. AMD usually does as well, but we've found nothing about the Seattle [A1100] processors, for example, and haven't found anyone to ask about it either.

There is a flurry of really interesting ARMv8-based server chips appearing, however, and we're excited about supporting them. We currently support AppliedMicro's X-Gene 1 SoC and ARM's FAST models for emulated hardware, but we'll talk to anyone who is prepared to share documentation with us.

Rik: In conversations we've had during USENIX conferences, you mentioned that Barrelfish will only support certain programming languages. Can you explain the thinking behind that decision?

Mothy: Actually, we're happy if Barrelfish supports any programming language that people would like. As a small team, there's a limit to the number that we can support ourselves, but we're happy to accept contributions!

We've had various student projects porting language runtimes to Barrelfish, and it's usually not too much of a problem. The key challenge is generally that languages often implicitly assume a POSIX-like system, and Barrelfish deliberately isn't like POSIX. This makes a fast Java runtime, for example, more work than Rust, which was extremely easy to bring up.

What you're probably referring to is the group decision about which languages we could use in the OS itself. Remember this was back in 2008, so this is before Go, D, Rust, Swift, and Dart had traction. The discussion was remarkably short and pretty much unanimous: we decided on C, assembly, Haskell, and Python for tools. We also felt happy with OCaml and Ruby, but in the event we didn't use them. We unanimously banned Java, C#, C++, and Perl. Nothing has happened to make us regret this decision.

Rik: Functional programming languages are becoming increasing popular. We now have a unikernel OS, MirageOS, but it relies on OCaml programming. When I mentioned just how hard I found it to write functional programs, you suggested that it might take someone six months to switch over to a functional programming style. Could you elaborate?

Mothy: We don't have any studies to back this up, and it depends ultimately on the programmer. However, OS kernels have always employed a variety of programming paradigms expressed in assembly or C, and good OS kernel hackers are generally comfortable with a variety of different ways of expressing computations inside the OS. You see a lot of snippets of functional programming in the Linux kernel, for example.

Choice of languages are a different matter. It's useful to contrast unikernels from operating systems: they're very different, and ultimately complementary.

We like to use a "functional" (in a different sense) definition of an OS—it is "that which manages, multiplexes, and protects the hardware resources of the machine." In this sense, it could be a hypervisor like Xen, or a traditional OS like Linux or Windows, or a multikernel like Barrelfish.

A unikernel like MirageOS is essentially a runtime for a single application that executes in a resource container. Some people call this a LibraryOS—a term which goes back to exokernel systems like Aegis and Nemesis.

For an OS, a garbage-collected language is problematic because you are interested in providing performance isolation between competing, untrusted applications (such as containers). Hence, implementations using a language with more predictable performance like C, Rust, or even C++ make a lot of sense.

For a unikernel, you're not worried about scheduling, resource sharing, or crosstalk since you're already isolated in a container. What you really want here is something which works well from a

software development and correctness perspective, and languages like OCaml are great for this kind of environment.

As an aside, if you move to a world where all your applications are running in containers over unikernels, the question naturally arises as to what the "ideal" underlying OS is. It's not Linux or Windows, and it's not Barrelfish either yet, but we are evolving Barrelfish that way—the Arrakis work is a step in that direction, for example.

References

[1] R. Farrow, "The Barrelfish Multikernel: Interview with Timothy Roscoe." *;login:*, vol. 35, no. 2 (April 2010): bit.ly/29lczBY.

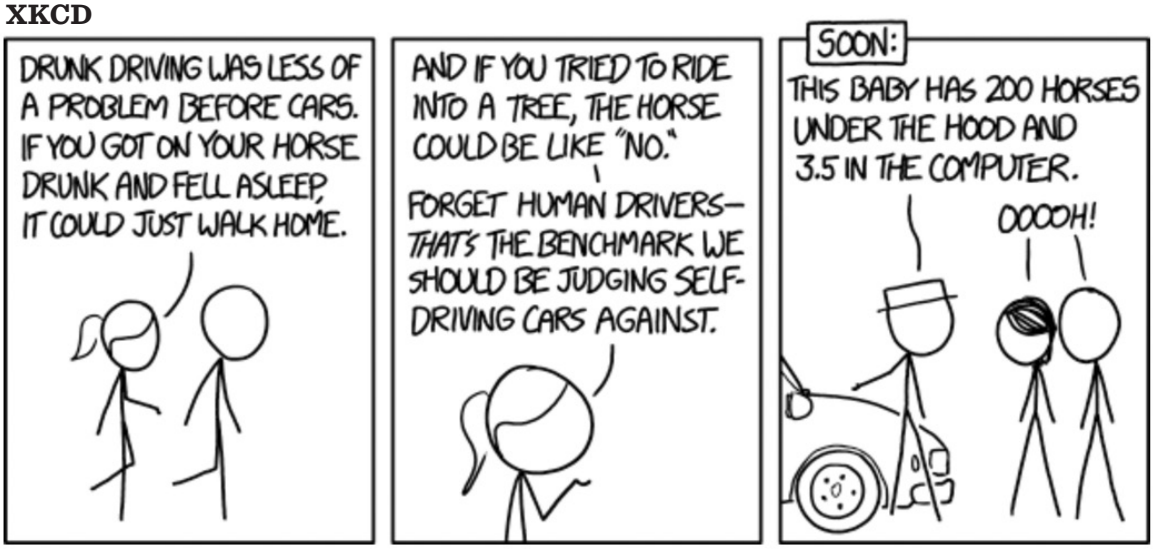
[2] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *Proceedings of the 22nd ACM Symposium on OS Principles,* October 2009: http://www.sigops.org/sosp/sosp09/papers/baumann-sosp09.pdf.

[3] "Intel Single-chip Cloud Computer," 2003: intel.ly/29oPSNx.

[4] Intel Xeon Phi: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[5] S. Peter and T. Anderson, "Arrakis: The Operating System as Control Plane." *;login:*, vol. 38, no. 4 (August 2013): bit.ly/29gu97f.

XKCD



xkcd.com