RIK FARROW

the Barrelfish multikernel: an interview with Timothy Roscoe

Timothy Roscoe is part of the ETH Zürich Computer Science Department's Systems Group. His main research areas are operating systems, distributed systems, and networking, with some critical theory on the side.

troscoe@inf.ethz.ch

Rik Farrow is the Editor of *;login:*.

rik@usenix.org

INCREASING CPU PERFORMANCE WITH faster clock speeds and ever more complex hardware for pipelining and memory access has hit the brick walls of power and bandwidth. Multicore CPUs provide the way forward but also present obstacles to using existing operating systems design as they scale upwards. Barrelfish represents an experimental operating system design where early versions run faster than Linux on the same hardware, with a design that should scale well to systems with many cores and even different CPU architectures.

Barrelfish explores the design of a multikernel operating system, one designed to run non-shared copies of key kernel data structures. Popular current operating systems, such as Windows and Linux, use a single, shared operating system image even when running on multiple-core CPUs as well as on motherboard designs with multiple CPUs. These monolithic kernels rely on cache coherency to protect shared data. Multikernels each have their own copy of key data structures and use message passing to maintain the correctness of each copy.

In their SOSP 2009 paper [1], Baumann et al. describe their experiences in building and benchmarking Barrelfish on a variety of Intel and AMD systems ranging from four to 32 cores. When these systems run Linux or Windows, they rely on cache coherency mechanisms to maintain a single image of the operating system. This is not the same thing as locking, which is used to protect changes to data elements which themselves consist of data structures, such as linked lists, that must be changed atomically. In monolithic kernels, a change to a data element must be visible to all CPUs, and this consistency gets triggered when a CPU attempts to read or write this data in its own cache. Cache consistency mechanisms prevent the completion of this read or write if the cache line is invalid, and also mean that execution may be paused until the operation is complete.

In a multikernel, each CPU core runs its own kernel and maintains its own data structures. When a kernel needs to make a change to a data structure (e.g., memory page tables) that must be coordinated with kernels running on other cores, it sends messages to the other kernels.

I asked Timothy Roscoe of the Systems Group at ETH Zurich if he could answer a few questions

about Barrelfish, working in a manner similar to Barrelfish, using asynchronous messaging. Before I begin the interview, Mothy wanted me to point out that the development of Barrelfish involves a very large team of people, and he is just one person among many working on this very complex project. You can learn more about this team by visiting the Barrelfish Web site, http://www.barrelfish.org/.

Farrow: Barrelfish maintains separate kernel state, and this seems to me to be one of the key differentiators from monolithic kernels.

Roscoe: Actually, this is not quite, but nearly, true: monolithic kernels started with a single shared copy of kernel state, and to a limited extent they have started to replicate or partition this state to reduce memory contention on multiprocessors. Solaris is probably the most advanced version of this. The model, however, remains one of a single image managing the whole machine, with the replication and/or partitioning of kernel state as an optimization.

In a multikernel, this is the other way around. No kernel state at all is shared between cores by default, and so consistency must be maintained by explicitly sending messages between cores, as in a distributed system. The model is one of replicated or partitioned data which is accessed the same way as one would access replicas in a distributed system. In particular, depending on the consistency requirements, changing some OS state may be a two-phase operation: a core requests a change and, at some point in the future, gets confirmation back that every other core has agreed to it, or, alternatively, that it conflicted with some other proposed change and so didn't happen.

In principle, we could share kernel data between cores in Barrelfish, and this might be a good idea when the cores are closely coupled, such as when they share an L2 or L3 cache or are actually threads on the same core. We also intend to do this at some point, but the key idea is that the model is of replicated data, with sharing as a transparent optimization. In traditional kernels it's the other way around.

Farrow: Barrelfish has a small CPU driver that runs with privilege, and a larger monitor process that handles many of the tasks found in a monolithic operating system. Barrelfish is not a microkernel, as microkernels share a single operating system image, like much larger monolithic kernels. Barrelfish does seem to share some characteristics of microkernels, such as running device drivers as services, right?

Roscoe: You're right that every core in Barrelfish runs its own CPU driver, which shares no memory with any other core. Also, every core has its own monitor process, which has authority (via capabilities) to perform a number of privileged operations. Most of the functionality you would expect to find in a UNIX kernel is either in driver processes or servers (as you would expect in a microkernel) or the distributed network of monitor processes.

Farrow: The SOSP paper talks about a system knowledge base (SKB) that gets built at boot time using probes of ACPI tables, the PCI bus, CPUID data, and measurement of message passing latency. Could you explain the importance of the SKB in Barrelfish?

Roscoe: The SKB does two things. First, it represents as much knowledge as possible about the hardware in a subset of first-order logic—it's a Constraint Logic Programming system at the moment. This, as you say, is populated using resource discovery and online measurements. Second, because it's a reasoning engine, the OS and applications can query it by issuing constrained optimization queries.

This is very different from Linux, Solaris, or Windows: traditional OSes often make some information about hardware (such as NUMA zones) available, but they often over-abstract them, the format of the information is ad hoc, and they provide no clean ways to reason about it (resulting in a lot of non-portable complex heuristic code). The SKB is not a magic bullet, but it drastically simplifies writing OS and application code that needs to understand the machine, and it means that clients can use whatever abstractions of the hardware are best for them, rather than what the OS designer thought useful.

We currently build on ARM, x86_64, x86_32, and Beehive processors. We're currently also porting to Intel's recently announced SCC (Single-chip Cloud Computer), which is a somewhat unconventional variant of x86_32.

One interesting feature of Barrelfish is that you don't really "port" the OS to a different architecture; rather, you add support for an additional CPU driver. Since CPU drivers and monitors only communicate via messages, Barrelfish will in principle happily boot on a machine with a mixture of different processors.

Farrow: While reading the paper, I found myself getting confused when you discussed how a thread or process gets scheduled. Could you explain how this occurs in Barrelfish?

Roscoe: Well, here's one way to explain this: Barrelfish has a somewhat different view of a "process" from a monolithic OS, inasmuch as it has a concept of a process at all. It's probably better to think of Barrelfish as dealing with "applications" and "dispatchers."

Since an application should, in general, be able to run on multiple cores, and Barrelfish views the machine as a distributed system, it follows that an application also, at some level, is structured as a distributed system of discrete components which run on different cores and communicate with each other via messages.

Each of these "components," the representative of the application on the core, so to speak, is called a "dispatcher." Unlike a UNIX process (or thread), dispatchers don't migrate—they are tied to cores. When they are descheduled by the CPU driver for the core, their context is saved (as in UNIX), but when they are rescheduled, this is done by upcalling the dispatcher rather than resuming the context. This is what Psyche and Scheduler Activations did, to first approximation (and K42, which is what we took the term "dispatcher" from, and Nemesis, and a few other such systems).

Farrow: So how do you support a traditional, multi-threaded, shared-memory application like OpenMP, for example?

Roscoe: Well, first of all, each dispatcher has, in principle, its own virtual address space, since each core has a different MMU. For a shared-memory application, clearly these address spaces should be synchronized across the dispatchers that form the application so that they all look the same, whereupon the cache coherence hardware will do the rest of the work for us. We can achieve this either by messages or by sharing page tables directly, but in both cases some synchronization between dispatchers is always required when mappings change.

As an application programmer, you don't need to see this; the dispatcher library handles it. Incidentally, the dispatcher library also handles the application's page faults—another idea we borrowed from Nemesis and Exokernel.

Application threads are also managed by the dispatchers. As long as a thread remains on a single core, it is scheduled and context-switched by the

dispatcher on that core (which, incidentally, is a much nicer way to implement a user-level threads package than using signals over UNIX). Note that the CPU driver doesn't know anything about threads, it just upcalls the dispatcher that handles these for the application, so lots of different thread models are possible.

To migrate threads between cores (and hence between dispatchers), one dispatcher has to hand off the thread to another. Since the memory holding the thread state is shared, this isn't too difficult. It's simply a question of making sure that at most one dispatcher thinks it owns the thread control block at a time. The dispatchers can either do this with spinlocks or by sending messages.

Farrow: Why should a multikernel work better than a monolithic kernel on manycore systems? In your paper, you do show better performance than a Linux kernel when running the same parallel tasks, but you also point out that the current Barrelfish implementation is much simpler/less functional than the current Linux kernel.

Roscoe: Our basic argument is to look at the trends in hardware and try to guess (and/or influence) where things are going to be in 10 years.

The main difference between a multikernel like Barrelfish and a monolithic OS like Linux, Windows, or Solaris is how it treats cache-coherent shared memory. In monolithic kernels, it's a basic foundation of how the system works: the kernel is a shared-memory multi-threaded program. A multikernel is designed to work without cache-coherence, or indeed without shared memory at all, by using explicit messages instead.

There are four reasons why this might be important:

First, cache-coherent shared memory can be slower than messages, even on machines today. Accessing and modifying a shared data structure involves moving cache lines around the machine, and this takes hundreds of machine cycles per line. Alternatively, you could encode your operation (what you want to be done to the data structure) in a compact form as a message, and send it to the core that has the data in cache. If the message is much smaller than the data you need to touch, and the message can be sent efficiently, this is going to be fast.

"Fast" might mean lower latency, but more important is that cores are generally stalled waiting for a cache line to arrive. If instead you send messages, you can do useful work while waiting for the reply to come back. As a result, the instruction throughput of the machine as a whole is much higher, and the load on the system interconnect is much lower—there's just less data flying around.

Ironically, in Barrelfish on today's hardware, we mostly use cache-coherent shared memory to implement our message passing. It's really the only mechanism you've got on an x86 multiprocessor, aside from inter-processor interrupts, which are *really* expensive. Even so, we can send a 64-byte message from one core to another with a cost of only two interconnect transactions (a cache invalidate and a cache fill), which is still much more efficient than modifying more than three or four cache lines of a shared data structure.

The second reason is that cache-coherent shared memory can be too hard to program. This sounds counterintuitive—it exists in theory to make things easier. It's not about shared-memory threads vs. messages per se either, which is an old debate that's still running. The real problem is that hardware is now changing too fast, faster than system software can keep up.

It's a bit tricky, but ultimately not too hard to write a correct parallel program for a shared-memory multiprocessor, and an OS is to a large extent a somewhat special case of this. What's much harder, as the scientific computing folks will tell you, is to get good performance and scaling out of it. The usual approach is to specialize and optimize the layout of data structures, etc., to suit what you know about the hardware. It's a skilled business, and particularly skilled for OS kernel developers.

The problem is that as hardware gets increasingly diverse, as is happening right now, you can't do this for general mass-market machines, as they're all too different in performance characteristics. Worse, new architectures with new performance tradeoffs are coming out all the time, and it's taking longer and longer for OS developers, whether in Microsoft or in the Linux community, to come out with optimizations like per-core locks or read-copy-update—there's simply too much OS refactoring involved every time.

With an OS built around inter-core message passing rather than shared data structures, you at least have a much better separation between the code responsible for OS correctness (the bit that initiates operations on the replicated data) and that responsible for making it fast (picking the right consistency algorithm, the per-core data layout, and the message passing implementation). We'd like to think this makes the OS code more agile as new hardware comes down the pipe.

The third reason is that cache-coherent shared memory doesn't always help, particularly when sharing data and code between very different processors. We're beginning to see machines with heterogeneous cores, and from the roadmaps this looks set to continue. You're going to want to optimize data structures for particular architectures or cache systems, and a one-size-fits-all shared format for the whole machine isn't going to be very efficient. The natural approach is to replicate the data where necessary, store it in a format appropriate to each core where a replica resides, and keep the replicas in sync using messages—essentially what we do in Barrelfish.

The fourth reason is that cache-coherent shared memory doesn't always exist. Even a high-end PC these days is an asymmetric, non-shared memory multiprocessor: GPUs, programmable NICs, etc., are largely ignored by modern operating systems and are hidden behind device interfaces, firmware blobs, or, at best, somewhat primitive access methods like CUDA.

We argue that it's the job of the OS to manage all the processors on a machine, and Barrelfish is an OS designed to be able to do that, regardless of how these programmable devices can communicate with each other or the so-called "main" processors.

It's not even clear that "main" CPUs will be cache-coherent in the future. Research chips like the Intel SCC are not coherent, although they do have interesting support for inter-core message passing. I'm not sure there's any consensus among the architects as to whether hardware cache-coherence is going to remain worth the transistor budget, but there's a good chance it won't, particularly if there is system software whose performance simply doesn't need it.

Barrelfish is first and foremost a feasibility study for this—knowing what we now do about how to build distributed systems, message passing, programming tools, knowledge representation and inference, etc., we can build an OS for today's and tomorrow's hardware which is at least competitive with current performance on a highly engineered traditional OS and which can scale out more effectively and more easily in the future.

If a handful of researchers can do that, it sounds like a result.

REFERENCE

[1] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," *Proceedings of the 22nd ACM Symposium on OS Principles*, Big Sky, MT, USA, October 2009: http://www.barrelfish.org/barrelfish_sosp09.pdf.

N.B.: The Barrelfish team also includes researchers Jan Rellermeyer, Richard Black, Orion Hodson, Ankush Gupta, Raffaele Sandrini, Dario Simone, Animesh Trivedi, Gustavo Alonso, and Tom Anderson.