Outlook on Operating Systems

Dejan Milojičić, Hewlett Packard Labs
Timothy Roscoe, ETH Zurich

Will OSs in 2025 still resemble the Unix-like consensus of today, or will a very different design achieve widespread adoption?

Seventeen years ago, six renowned technologists attempted to predict the future of OSs for an article in the January 1999 issue of *IEEE Concurrency*.[1] With the benefit of hindsight, the results were decidedly mixed. These six experts correctly predicted the emergence of scale-out architectures, nonuniform memory access (NUMA) performance issues, and the increasing importance of OS security. But they failed to predict the dominance of Linux and open source software, the decline of proprietary Unix variants, and the success of vertically integrated Mac OS X and iOS.

The reasons to believe that OS design won't change much going forward are well known and rehearsed: requirements for backward compatibility, the Unix model's historical resilience and adaptability,[2] and so on. "If it ain't broke, don't fix it."

However, we argue that OSs will change radically. Our motivation for this argument is two-fold. The first has been called the "Innovator's Dilemma," after the book of the same name:[3] a variety of interests, both commercial and open source, have invested substantially in current OS structures and view disruption with suspicion. We seek to counterbalance this view. The second is more technical: by following the argument that the OS will change, we can identify the most promising paths for OS research to follow—toward either a radically different model or the evolution of existing systems. In research, it's often better to overshoot (and then figure out what worked) than to undershoot.

Current trends in both computer hardware and application software strongly suggest that OSs will need to be designed differently in the future. Whether this means that Linux, Windows, and the like will be replaced by something else or simply evolve rapidly will be determined by a combination of various technical, business, and social factors beyond the control of OS technologists and researchers. Similarly, the change might come from incumbent vendors and open source communities, or from players in new markets with requirements that aren't satisfied by existing designs.

Ultimately, though, things are going to have to change.

HARDWARE TRENDS

Hardware is changing at the levels of individual devices, cores, boards, and complete computer systems, with deep implications for OS design.

Complexity

Hardware is becoming increasingly more complex. Today, the programming manual for a system-on-chip (SoC) part for a phone or server blade typically runs to more than 6,000 pages, usually not including documentation for the main application cores. Large numbers of peripheral devices are integrated onto a die, each with complex, varying programming models. More and more of these devices—networking adaptors, radios, graphics processors, power controllers, and so on—are now built as specialized processors that execute specialized firmware with little OS integration.

The OS communicates with these peripherals via proprietary messaging protocols specific to producers of peripheral devices. This marks the beginning of the trend toward *dark silicon*, a large collection of highly specialized processors, only a few of which can be used at a time.[4]

These devices' interconnects are also becoming more complex. A modern machine is a heterogeneous network of links, interconnects, buses, and addressing models. Not all devices are accessible from all general-purpose cores, and the same is true of memory: the cozy view of a computer as a single cache-coherent physical address space containing RAM and devices has been a myth for at least the last decade.

Energy

These systems aren't static, either. As a system resource to be managed by the OS, energy is now just as important as CPU cycles, bytes of RAM, or network bandwidth, whether in the form of cellphone battery life or datacenter power consumption. Not least among the many implications for new OS design is that most of the hardware—such as cores, devices, and memory—can be powered up and down at any point during execution.

Nonvolatile main memory

As technology advances, we expect large, nonvolatile main memories to become prevalent.[5] This doesn't only mean that most data will persist in nonvolatile memory (NVM) as opposed to DRAM or disk, but also that packaging, cost, and power efficiency will make it possible to architect and deploy far more nonvolatile RAM (NVRAM) than DRAM. Main memory will be mostly persistent and will have much greater capacity than today's disks and disk arrays.

With large numbers of heterogeneous processors, this memory will be highly distributed but perhaps not in the way we see today. Photonic interconnects and high-radix switches can expand the load-store domain (where cores can issue cache-line reads and writes) across multiple server units (within a rack or even a datacenter) and potentially flatten the switch hierarchy, resulting in more uniform memory access.[6]

This further enlarges the memory accessible from a single CPU beyond that supported by modern 64-bit processors, which implement no more than 52 bits of physical address space and as little as 42 bits for some CPUs. In the short term (it'll be a few years until processor vendors implement more address bits), this will lead to configurable apertures or windows into available memory. Typical time scales of general-purpose OSs span multiple decades, whereas real-time and embedded OS lifetimes are only a few years.

In the longer term, memory controllers are likely to become more intelligent and programmable at the OS—and perhaps application—level. They will be able to execute adaptive algorithms subject to memory access patterns. Memory controller functions will be executed closer to memory (outside CPUs), implementing optimizations such as encryption, compression, and quality-of-service functions.

Systems

Taking a step back, we see that the boundaries of today's machines are different from traditional scale-up and scale-out systems. The resources of a closely coupled cluster such as a rack-scale InfiniBand cluster must be managed at a timescale the OS is used to, rather than those used for traditional middleware.

We're so accustomed to thinking of Linux or Windows as OSs—because they started that way—that it's rare to consider a functional definition of an OS. Consider the following traditional definition of an OS: "an OS is a collection of system software that manages the complete hardware platform and securely multiplexes resources between tasks."

Linux, Windows, and Mac OS all fail this definition. A cellphone OS is a mishmash of proprietary device firmware, embedded real-time executives, and the Linux or iOS kernel and its daemons that run applications and manage a fraction of the hardware. The OS of a datacenter or rack-scale appliance is typically a collection of different Linux installations as well as custom and off-the-shelf middleware. If we want to examine the structure of a real-world, contemporary OS, we need to look elsewhere. A great deal of traditional OS functionality is now occurring outside of general-purpose OSs; it's moved to the top-of-rack management server, closer to memory (memory-side controllers), to various appliances (intrusion detection systems, and storage), or to specialized cores on the same die.

Diversity

Hardware is also becoming more diverse. Beyond the complexity of any single product line of machines, the designs of every processor, SoC, and complete system are different. SoCs were always heterogeneous, but they were used in special-purpose systems; now they're used in general-purpose systems. Engineering a general-purpose OS that can be used widely and evolve as hardware changes (and with it, the tradeoffs required for scalability, performance, and energy efficiency) is a formidable challenge.

In a remarkable change from 15 years ago, hardware adapts and diversifies much faster today than system software does—faster than the current OS engineering practice of monolithic kernels can keep up with. Short-term solutions include backward-compatible workarounds by hardware vendors (giving up on the full potential of new hardware features), hiding or mitigating the problem through vertical integration, or deploying huge internal software-engineering efforts for each new product.

APPLICATION CHANGES

The ways in which computers are used and the applications that run on them are also changing, calling into question many OS designs of the last 40 years. This is both a challenge and an opportunity. It's a challenge because current OSs don't match up well with the applications they're called on to support. It's an opportunity because the burden of history (backward compatibility, long-standardized APIs, and established programming models, for example) is light: applications are very different now, granting us the freedom to change the OS interface.

Rack-scale computing

One trend we're seeing in applications is rack-scale computing, which is sometimes deployed as software appliances (called *tin-wrapped software*). Many enterprise applications such as file servers, relational and nonrelational databases, and big data analytics now come prepackaged in a rack-scale software appliance (examples include Oracle's Exadata and Exalytics products or the SAP HANA database). Such appliances usually consist of a collection of server machines and optional custom hardware accelerators, connected by a high-bandwidth internal network such as InfiniBand. Customers plug in the power and network cables, configure it, and go.

Tin-wrapped software is attractive to software vendors for a number of reasons. Because the vendor controls the hardware platform that the OS runs on, support costs are greatly reduced—there's no need to validate the software on every conceivable PC server, network card, or version of Windows or Linux. Tin-wrapped software also allows vendors to introduce custom hardware that doesn't have to be compatible with every customer's existing systems. The market for such enterprise software appliances is huge.

Because the appliance only runs one software package, one might think OS issues are simplified, but this isn't true in practice. Enterprise software packages are highly complex services with many different tasks running at the same time and competing for resources. In addition to the traditional application resources an OS has to manage, there's also the appliance's backplane network. Allocating interconnect bandwidth becomes important in these machines: a skewed hash join in a large relational database can easily saturate 100 gigabits per second (Gbps) FDR InfiniBand links inside the appliance and significantly impact performance. Distributed coordination of CPU scheduling, combined with careful interconnect management, is essential.

In an effort to optimize performance, rack-scale applications often exploit the lowest-latency communication mechanisms available, such as *remote direct memory access* (RDMA) one-side operations. This means that memory management and protection in the OS becomes a distributed systems problem.

Large-scale persistent main memories are likely to be adopted first in the appliance space because most data processing in such systems is already done in main memory (either RAM or with a large cache of memory-mapped flash memory). Being able to support very large, persistent physical memory within a given power envelope could greatly increase the capacity of enterprise data-processing appliances.

Datacenter challenges

Beyond the scale of information appliances are enterprises and service providers who operate applications across entire datacenters.

Datacenter challenges include application deployment, upgrades, and maintenance. Unlike tin-wrapped software, datacenter applications can't be coupled to a controlled hardware platform. Instead, code must be upgraded across thousands of machines in a coordinated manner, often without suffering from downtime. Provisioning capacity for such applications is an ongoing problem: workloads change and new processing nodes must be acquired, installed, and added to the running system without adversely affecting computation.

Datacenters run many applications at a time—sharing single conventional OSs, packaged into containers, or running on virtual machines (and potentially over virtual networks). In all cases, performance isolation is a critical concern. Indeed, for many commercial applications, adequate predictable performance is a correctness issue, albeit in a very different manner from classical real-time systems.

Although applications require proper allocation of interconnect bandwidth in rack-scale systems, cloud services in datacenters often have much more complex requirements from the infrastructure, both in their use of dynamically instantiated virtual machines and with the increase in software-defined networks (SDNs) and network function virtualization.

These applications' security requirements have also radically changed—not only do the requirements apply to network traffic, but these applications can have hundreds of millions of users. In many cases, as we've seen recently, such applications' security is best expressed in terms of complex security policies governing information flow to prevent leakage of sensitive data, not as simple resource access control.

Finally, these applications increasingly span centralized datacenter services, individual users' mobile devices, and a growing number of embedded sensors and actuators in the Internet of Things. A look at recent media reports and research papers strongly suggests that no one has a good handle on how to manage the interconnected security, reliability, and software update issues posed by this new kind of environment.

In the early days of computers, an OS existed to manage a competing set of complete applications sharing a machine. The basic OS abstractions that evolved to represent applications (for example, processes) have long been inadequate: applications grew to include many processes in the same OS. Today, a single application spans potentially thousands of OS instances on hardware platforms ranging from sensors and smartphones to high-performance datacenter server machines. It shouldn't be surprising, then, that existing OSs, whose core abstractions have changed little over the last three decades, don't provide a clear match to this application structure.

WHAT'S BROKEN?

It's remarkable how many assumptions about OS design embodied in modern systems like Linux and Windows are either violated or irrelevant, in light of the modern trends in hardware and application models previously mentioned.[7] We're being deliberately provocative in this article, but we're also serious.

We're certainly not the first to point out the deficiencies of current OS designs. Also note that the following issues are not "bugs"—they arise from the fundamental structures on which an OS like Unix or Windows is based. If you fixed them, you'd have a different OS. This naturally begs the question: However it ends up being branded, what will such an OS look like?

Single monolithic kernel

Modern OS designs share the concept of a single, multithreaded, shared-memory program that runs in kernel mode and handles interrupts and system calls. This simply doesn't work on a machine with heterogeneous processors; different instruction sets; and memory that's not completely coherent, exists at different addresses as seen by different cores, and might not even be shared. All of these are features of modern machines that are likely to continue in the future.

A single large kernel raises other concerns, such as trust. Viewing a single monolithic kernel as the trusted computing base is one thing on a machine with a few gigabytes of memory and a handful of cores, but is very different on a machine with a petabyte of main memory and thousands of cores. With hardware at this scale, transient or persistent failures are common (as they are in clusters), and it's no longer reasonable for one core to depend on, let alone trust, code running in kernel mode on another core on the other side of the machine.

Recognizing this situation forces an OS designer into a much more distributed design. The interesting question moving forward is to what extent this scenario resembles a classical distributed system and to what extent it will be something new. At least two features distinguish large, modern computers from the distributed systems of yore: they have regions of partially shared memory as well as message channels, and the message latency between nodes (or cores) is close to the cost of a last-level cache miss on a single core.

Authorization and security

Unsurprisingly, an authorization and security model designed for a small workgroup of trusted interactive users (such as POSIX) or human members of a larger organization (such as Kerberos or Windows Active Directory) is inappropriate for online cloud services, application stores, virtual infrastructure platforms, single-user handheld devices, and distributed applications.

The security challenges we face today concern privacy, information flow, untrusted applications running with user privileges, and social networks with billions of users. At the OS layer, it might not make much sense to talk about traditional users—human users installing and running programs they don't understand from third parties are the wrong security principals. A fine-grained authorization mechanism like that afforded by capabilities, perhaps combined with a concept of distributed information flow control at scale, could be the way forward.

Scheduling

Modern OS schedulers manage processes or threads as a basic unit of CPU allocation. But for modern multicore hardware and typical applications, this is irrelevant. The recent increase in containers—and virtual machines before them—is a response to the fact that, astonishingly, no mainstream OS in the mid-1990s had an abstraction corresponding either to a complete application installation or to a running application instance. On a single machine, an application spans multiple processes and threads, and calls out to server processes it shares with other programs. Moreover, dynamic migration of threads to balance the load across cores (as in Linux, for example) makes no sense when cores are radically heterogeneous, and where the objective is to optimize energy usage subject to fixed performance goals, rather than raw interactive response time or bulk throughput. Effective spatial scheduling, rather than temporal scheduling, becomes the key challenge.

Worse, in rack-scale machines, an application is inherently distributed. In larger systems, it runs on behalf of millions of individual users. As with security, we need scheduling entities that make sense. Containers are a step in the right direction, but they don't yet make sense in a distributed machine. Building the right CPU scheduler for the future will be challenging: early experience with cluster-level schedulers suggests that a loosely coupled, distributed approach is essential. There's simply too much important, fine-grained, and time-sensitive scheduling information to be effectively aggregated in a single centralized scheduler.

[THE WAYS IN WHICH COMPUTERS AND APPLICATIONS ARE USED ARE CHANGING, CALLING INTO QUESTION MANY OS DESIGNS OF THE PAST 40 YEARS.]

Virtual memory

Virtual memory is another strong candidate for change.[8] Page-based address translation hardware was originally designed to allow applications to use more memory than was physically present in the machine via demand paging. The OS abstraction corresponding to this hardware was a virtual address space magically backed with (uniform) physical RAM.

Paging almost never happens today, and large machines in the future are likely to have much more memory than can be represented in the virtual address space. The translation hardware (memory managing unit; MMU) has proven to be incredibly useful for a variety of purposes: relocation of code and data, simplified memory allocation, copy-on-write of regions, detecting reads and writes to certain locations, and so on.

However, this has all been achieved in spite of, rather than using, the basic virtual address space abstraction, which hides physical memory characteristics. Moreover, the physical memory backing an application increasingly matters to the application. For example, RDMA reads and writes are stored in physical memory, data structures are allocated in NUMA-aware ways, and physical memory exists in different forms (DRAM, scratchpad, persistent, and so on). This rich functionality is accessed by a motley collection of functions that punch holes in the clean virtual address space abstraction.

Network stack

The network stack is also looking a bit tired. Network bandwidth to an adaptor is still increasing, but the speed of individual processing cores is decreasing. The solution is to demultiplex network flows between end-system cores in the network interface controller (NIC) hardware, using direct memory access (DMA) to write into a large number of different ring buffers (some modern NICs already

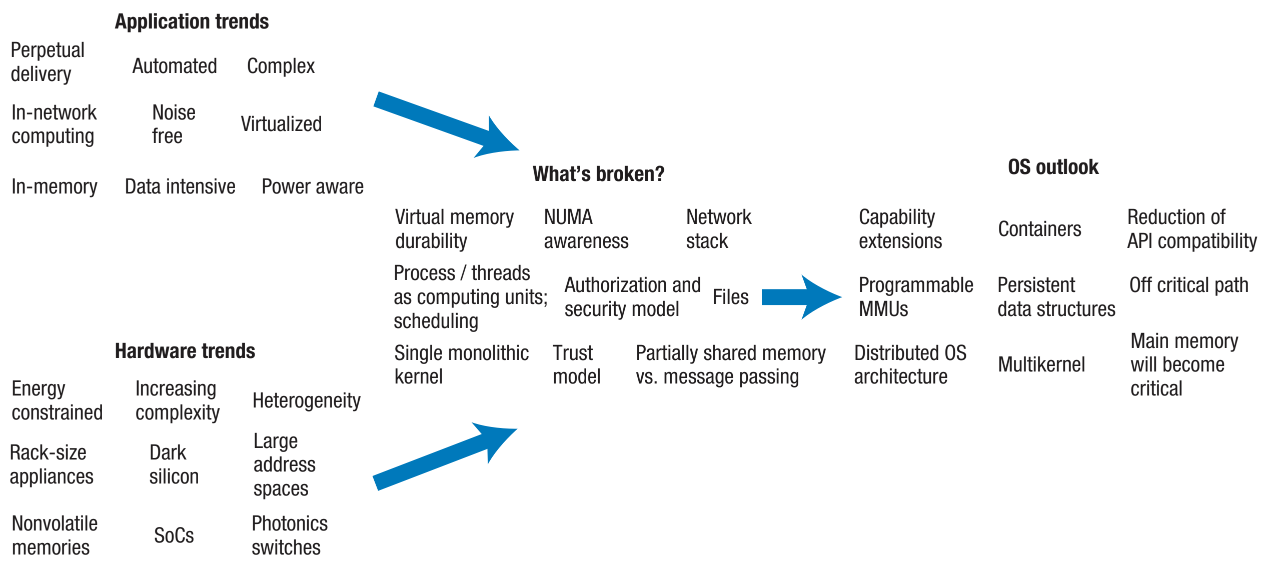

FIGURE 1. Impact of application and hardware trends on OSs, what's broken in current OS design, and the OS outlook. SoCs: systems on chip; NUMA: nonuniform memory access; MMUs: memory managing units.

support several thousand of these). The kernel is therefore bypassed by the data plane because it constitutes a serial bottleneck even in modern, highly optimized network stacks, and because it avoids expensive kernel crossings—slower cores relative to the network mean that every CPU cycle on the data path counts. This happened first in virtual machine monitors and spurred the adoption of technologies such as Single-Root I/O Virtualization (SR-IOV), but recent research shows the benefits of structuring a general-purpose OS this way as well.[9]

The key change in perspective comes with the realization that the NIC is no longer an interface, per se—it's a network switch. Specifically, it's an asymmetric switch with a very high port count—every send and receive queue constitutes a port as well as the physical layer—that makes forwarding decisions at all layers of the traditional protocol stack. The role of the future OS is to implement the control plane for this switch.

Data storage

Finally, POSIX-like files don't make sense at scale. Large-scale data processing applications access datasets through distributed parallel file systems like the Hadoop Distributed File System (HDFS) that expose how data is sharded across storage nodes, or through higher-level record-oriented distributed object stores. Files also disappear at the small end of the scale: phone applications store their data in isolated containers through an API that hides any underlying Linux or iOS file systems, and in practice can transparently replicate such data across crowd services that use the data. The file as a concrete vector of bytes on local storage means little to most application programmers, let alone users. Its deserialized structure is what's important.

Recently, it has been recognized that data durability doesn't necessarily imply writing to stable storage. Once the necessary availability mechanisms are in place to replicate application data structures across active nodes in a large online service (including across datacenters), some data can even be permanently held in main memory. With the rise of persistent main memory, a great deal of long-lived application data will never be serialized to disk. Applications will deal with persistent state differently, and the OS must support this in a useful way.

OUTLOOK

We conclude with a few bold—perhaps reckless—predictions, and some reflections on the process by which OS designs are created and evolve over time. Figure 1 visually summarizes our predictions. For more information, see the "Opportunities for a New OS" sidebar.

Architecture

The future OS will be distributed in architecture. A single kernel isn't going to work with heterogeneous processors, memory that isn't accessible from all cores, or partial cache coherence. There will be multiple kernels in a single machine, and some kind of message passing is inevitable. The result will be something between a single machine and a traditional cluster—the low message latencies of future hardware and performance requirements of parallel applications will necessitate much more global coordination in scheduling, memory allocation, and bandwidth allocation than is achievable in current clusters. This shouldn't be confused with traditional embedded system architecture, which is similar because it runs dedicated kernels and applications compared to general-purpose OSs and applications.

OPPORTUNITIES FOR A NEW OS

Figure A represents a bell-curve distribution of the number of deployed instances of Linux versus the system size. On the high-end system side (laptops through supercomputers), the bell curve reaches a maximum for the two-socket Intel computer, the most commonly deployed Linux machine. The market is much more fragmented on the low-end system side (such as Linux deployed as Android on mobile phones), so we focus primarily on the high end, acknowledging that the number of mobile phones has surpassed the number of mobile computers. Sensors in cyber-physical systems connected to the Internet of Things (IoT) are shown on the far left side of the curve (small systems), and the largest Linux deployments (ranging from high-performance computing systems to exascale systems) are on the far right.

The Linux OS is optimized for a sweet spot—the maximum of the bell curve covering the area bounded by the extremes. Going outside these zones will compromise the optimizations for this sweet spot. Thus, the Linux community has less interest in going outside the current zones, where innovation can happen.

In the 1980s and 1990s, there were many different OS versions, each bringing a certain degree of innovation. However, this also caused a fragmentation of customers followed by some consolidation, such as that around the Open Software Foundation and Unix International. Eventually, the open source community prevailed and Linux started dominating.

Linux—which purely followed the Innovator's Dilemma model (C.M. Christensen, *The Innovator's Dilemma*, Harper Business, 2011)—was absorbing innovation from earlier systems (such as Digital Unix, HP-UX, and IBM AIX) until about 5 years ago, when innovation started happening within Linux itself. Most recently, innovation has occurred within the IoT and high-end spectrums of the bell curve. New OSs are being created with seemingly opposite yet symmetrically similar requirements. This is illustrated in Figure A by the new OSs listed on each side of the curve.

Power savings is equally needed at the lower end (such as battery lifetime) and the higher end (such as recurrent costs). Real-time requirements and synchronicity have similar goals in both domains, as well as low-latency communication, code size, and minimal APIs. The same minimal kernel can be deployed on both ends of the bell curve, meeting similar demands from different types of applications.

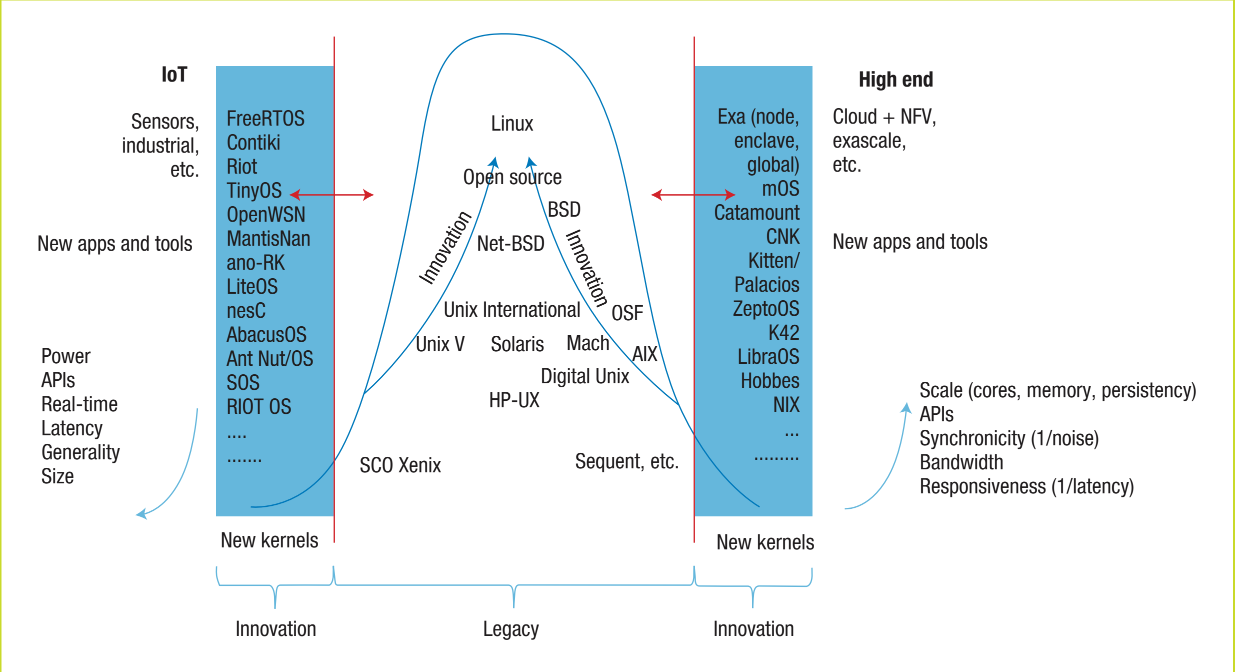

Figure A. Number of deployed OS instances on computers (excluding phones). NFV is network function virtualization.

An interesting question is whether this can be solved by architecting a new multikernel OS structure or by evolving current cluster middleware into a different performance regime. We favor the former. Today's datacenters are coordinated by an ad hoc collection of schedulers, resource allocators, SDN controllers, application deployment tools, configuration management systems, and so on. In a marketing context, this is sometimes referred to as a "datacenter OS," but it lacks any coherent architecture to date. We believe this ecosystem won't work well with tightly coupled clusters.

An objection to this view is that the world is moving toward a large collection of per-application virtual machines or containers, which are easy to support on existing OSs. This is arguably an issue of perspective: the virtual machines or containers still need a hypervisor underneath, which is an OS. Moreover, we've already observed that applications are increasingly parallel and distributed.

Containers are important: they make it easy to deploy traditional applications in a virtualized traditional OS environment. Their low overhead allows applications to be restructured into much more scalable microservices, and enables a model of continuous re-engineering, refactoring, and redeployment. However, they don't yet adequately address the challenges of future hardware and scalable applications.

Memory

We predict that main memory will become critical compared to all other resources because it'll be the biggest influence on application performance. In most application scenarios, main memories will be very large and mostly persistent, and only cold data will be kept on disk. This is already a trend in databases and key-value stores (such as SAP HANA, Redis, and Memcached), and memory technology trends make this likely to continue. Memory and storage will converge.

Memory contents will persist across reboots, and files will be partially replaced with persistent in-memory data structures. This doesn't necessarily make life easier, though: a future OS is going to have to provide something like a sophisticated transactional facility to make persistent main memory usable. Sometimes remembering all the data isn't desirable across reboot.

Because main memory itself will be heterogeneous and distributed, we expect that the OS memory management interface will change. Instead of a virtual address space, applications will be given a much more explicit handle on the regions of physical memory allocated to them, and more freedom in safely programming the MMU to exploit hardware translation features from application runtimes. Demand paging is likely to be an optional library rather than a fundamental part of the OS.

Networking

Our third prediction concerns networking, both inside a machine and at the interface between a machine and the wider network. OS software will increasingly get off the critical data path between application threads, replaced by sophisticated "user-safe" hardware multiplexing/de-multiplexing and filtering functionality, and library code to interface to it. The OS will function as the control plane of a heterogeneous network of smart NICs, queues, cores, and DMA engines. The internal OS abstractions for this will resemble a more advanced form of those being formulated for SDNs.

Security

Current real-world computer security at all stack levels is in a poor state. Low-level primitives that will scale up to very large numbers of principals seem like a good place to start; for example, capability extensions to existing machine architectures,[10] which have repercussions that propagate up the software stack. We believe it's imperative to completely rethink the implementation of system security, and several research systems look very promising.

Applications

The OS interface will have to change, but it's unreasonable to expect application developers to start afresh. We believe the differences will start to appear in application platforms and language runtimes. For example, relational databases have already recognized the need to re-architect how they're written for modern hardware without replacing SQL.

FUTURE DIRECTIONS

These challenges make for an exciting time in OS research—a lot is up for grabs. Changes in hardware and applications have necessitated a shift in thinking not only about resource management in computers, but also about many of the constraints that OS designers have worked under.

These constraints have included the need to conform to whatever the hardware provides. OS system issues have been mostly ignored by hardware folks—computer scientist and professor Andrew Tanenbaum lamented that "no hardware designer should be allowed to produce any piece of hardware until three software guys have signed off on it"—but the OS community's response has been at best mildly passive-aggressive. However, this is changing: systems software people are becoming more assertive,[11] hardware is becoming easier to change, and the boundary between the two is becoming blurred through the increased use of field-programmable gate arrays and complex SoCs. In addition, the hardware product cycle is getting shorter and simulation tools are becoming more powerful, giving systems software developers earlier access to new hardware.

The need for API compatibility is also declining and moving higher up the stack. To perform well, strict POSIX compliance in an OS practically forces the implementation to resemble Unix internally, but many applications today are written to a much higher-level interface, which can be made portable across different low-level APIs. In cases where POSIX (or Windows) compatibility is necessary, virtual machines or library OSs can perform well without needing to natively support a legacy API.

This article concerns what the new OS should look like and what it should do, not how to engineer it. The latter is just as interesting a topic and broad enough to warrant a separate article. However, the challenges facing engineering OSs are similar. For example, formal methods are approaching the point where they can effectively be used to design and implement a uniprocessor microkernel with strong correctness proofs,[12] and OS researchers have enthusiastically embraced such tools and techniques. The key question moving forward is whether such ideas can catch up with the complexity, heterogeneity, and concurrency of modern hardware. C

ABOUT THE AUTHORS

DEJAN MILOJIČIĆ is a senior researcher at Hewlett Packard Labs in Palo Alto, California. His research interests include OSs, distributed systems, and systems management. Milojičić received a PhD in computer science from the University of Kaiserslautern. He was the 2014 IEEE Computer Society president. Contact him at dejan.milojicic@hpe.com.

TIMOTHY ROSCOE is a professor of computer science at ETH Zurich. His research interests include networks, OSs, and distributed systems. Roscoe received a PhD from the University of Cambridge. Contact him at timothy.roscoe@inf.ethz.ch.

ACKNOWLEDGMENTS

We thank Kirk Bresniker, Nigel Edwards, Paolo Faraboschi, Alexander Merritt, and Gerd Zellweger for reviewing this article and providing valuable comments. The feedback from anonymous reviewers helped us substantially improve the article.

REFERENCES

1. D. Milojičić, "Operating Systems—Now and in the Future," *IEEE Concurrency*, vol. 7, no. 1, 1999, pp. 12–21.
2. D.M. Ritchie and K. Thompson, "The Unix Time-Sharing System," *Comm. ACM*, vol. 17, no. 7, 1974, pp. 365–375.
3. C.M. Christensen, *The Innovator's Dilemma*, Harper Business, 2011.
4. H. Esmaeilzadeh et al., "Dark Silicon and the End of Multicore Scaling," *IEEE Micro*, vol. 32, no. 3, 2012, pp. 122–134.
5. K.M. Bresniker, S. Singhal, and R.S. Williams, "Adapting to Thrive in a New Economy of Memory Abundance," *Computer*, vol. 48, no. 12, 2015, pp. 44–53.
6. D. Vantrease et al., "Corona: System Implications of Emerging Nanophotonic Technology," *Proc. 35th Int'l Symp. Computer Architecture* (ISCA 08), 2008, pp. 153–164.
7. P. Faraboschi et al., "Beyond Processor-centric Operating Systems," *Proc. 15th USENIX Conf. Hot Topics in Operating Systems* (HotOS 15), 2015; http://dl.acm.org/citation.cfm?id=2831107.
8. S. Gerber et al., "Not Your Parents' Physical Address Space," *Proc. 15th USENIX Conf. Hot Topics in Operating Systems* (HotOS 15), 2015; www.usenix.org/system/files/conference/hotos15/hotos15-paper-gerber.pdf.
9. S. Peter et al., "Arrakis: The Operating System Is the Control Plane," *ACM Trans. Computer Systems*, vol. 33, no. 4, 2015, article 11.
10. R.N.M. Watson et al., "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," *Proc. IEEE Symp. Security and Privacy* (SP 15), 2015, pp. 20–37.
11. J.C. Mogul et al., "Mind the Gap: Reconnecting Architecture and OS Research," *Proc. 13th USENIX Conf. Hot Topics in Operating Systems* (HotOS 13), 2013; http://dl.acm.org/citation.cfm?id=1991596.1991598.
12. K. Gerwin et al., "seL4: Formal Verification of an OS Kernel," *Proc. ACM SIGOPS 22nd Symp. Operating System Principles* (SOSP 09), 2009, pp. 207–220.