Your computer is already a distributed system. Why isn't your OS?

Andrew Baumann* Simon Peter* Adrian Schüpbach* Akhilesh Singhania*
Timothy Roscoe* Paul Barham† Rebecca Isaacs†

*Systems Group, Department of Computer Science, ETH Zurich
†Microsoft Research, Cambridge

First published in *12th Workshop on Hot Topics in Operating Systems*, Monte Verità, Switzerland, May 2009.

1 Introduction

We argue that a new OS for a multicore machine should be designed ground-up as a distributed system, using concepts from that field. Modern hardware resembles a networked system even more than past large multiprocessors: in addition to familiar latency effects, it exhibits node heterogeneity and dynamic membership changes.

Cache coherence protocols encourage OS designers to selectively ignore this, except for limited performance reasons. Commodity OS designs have mostly assumed fixed, uniform CPU architectures and memory systems.

It is time for researchers to abandon this approach and engage fully with the distributed nature of the machine, carefully applying (but also modifying) ideas from distributed systems to the design of new operating systems. Our goal is to make it easier to design and construct robust OSes that effectively exploit heterogeneous, multicore hardware at scale. We approach this through a new OS architecture resembling a distributed system.

The use of heterogeneous multicore in commodity computer systems, running dynamic workloads with increased reliance on OS services, will face new challenges not addressed by monolithic OSes in either general-purpose or high-performance computing.

It is possible that existing OS architectures can be evolved and scaled to address these challenges. However, we claim that stepping back and reconsidering OS structure is a better way to get insight into the problem, regardless of whether the goal is to retrofit new ideas to existing systems, or to replace them over time.

In the next section we elaborate on why modern computers should be thought of as networked systems. Section 3 discusses the implications of distributed systems principles that are pertinent to new OS architecture, and Section 4 describes a possible architecture for an OS following these ideas. Section 5 lays out open questions at the intersection of distributed systems and OS research, and Section 6 concludes.

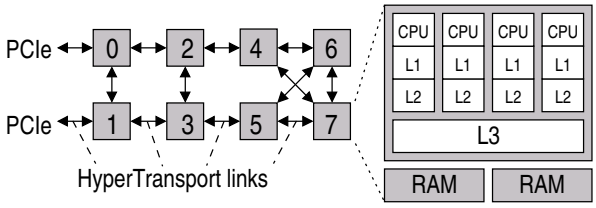

Figure 1: Node layout of a commodity 32-core machine

2 Observations

A modern computer is undeniably a networked system of point-to-point links exchanging messages: Figure 1 shows a 32-core commodity PC server in our lab[1]. But our argument is more than this: distributed systems (applications, networks, P2P systems) are historically distinguished from centralized ones by three additional challenges: node heterogeneity, dynamic changes due to partial failures and other reconfigurations, and latency.

Modern computers exhibit all these features.

Heterogeneity: Centralized computer systems traditionally assume that all the processors which share memory have the same architecture and performance tradeoffs. While a few computers have had heterogeneous main processors in the past, this is now becoming the norm in the commodity computer market: vendor roadmaps show cores on a die with differing instruction set variants, graphics cards and network interfaces are increasingly programmable, and applications are emerging for FPGAs plugged into processor sockets.

Managing diverse cores with the same kernel object code is clearly impossible. At present, some processors (such as GPUs) are special-cased and abstracted as devices, as a compromise to mainstream operating systems which cannot easily represent different processing archi-

[1] A Tyan Thunder S4985 board with M4985 Quad CPU card and 8 AMD "Barcelona" quad-core processors. This board is 3 years old.

Access	cycles	normalized to L1	per-hop cost
L1 cache	2	1	-
L2 cache	15	7.5	-
L3 cache	75	37.5	-
Other L1/L2	130	65	-
1-hop cache	190	95	60
2-hop cache	260	130	70

Table 1: Latency of cache access for the PC in Figure 1.

tectures within the same kernel. In other cases, a programmable peripheral itself runs its own (embedded) OS.

So far, no coherent OS architecture has emerged which accommodates such processors in a single framework. Given their radical differences, a distributed systems approach, with well-defined interfaces between the software on these cores, seems the only viable one.

Dynamicity: Nodes in a distributed system come and go, as a result of changes in provisioning, failures, network anomalies, etc. The hardware of a computer from the OS perspective is not viewed in this manner.

Partial failure in a single computer is not a mainstream concern, though recently failure of parts of the OS has become one [15], a recognition that monolithic kernels are now too complex, and written by too many people, to be considered a single unit of failure.

However, other sources of dynamicity within a single OS are now commonplace. Hot-plugging of devices, and in some cases memory and processors, is becoming the norm. Increasingly sophisticated power management allows cores, memory systems, and peripheral devices to be put into a variety of low-power states, with important implications for how the OS functions: if a peripheral bus controller is powered down, all the peripherals on that bus become inaccessible. If a processor core is suspended, any processes on that core are unreachable until it resumes, unless they are migrated.

Communication latency: The problem of latency in cache-coherent NUMA machines is well-known. Table 1 shows the different latencies of accessing cache on the machine in Figure 1, in line with those reported by Boyd-Wickizer *et al.* for a 16-core machine [5][2]. Accessing a cache line from a different core is up to 130 times slower than from local L1 cache, and 17 times slower than local L2 cache. The trend is towards more cores and an increasingly complex memory hierarchy.

Because most operating systems coordinate data structures between cores using shared memory, OS designs focus on optimizing this in the face of memory latency. While locks can scale to large machines [12], locality of data becomes a performance problem [2], since fetching remote memory effectively causes the hardware to perform an RPC call. In Section 3 we suggest insights from distributed systems which can help mitigate this.

Summary: A single machine today consists of a dynamically changing collection of heterogeneous processing elements, communicating via channels (whether messages or shared-memory) with diverse, but often highly significant, latencies. In other words, it has all the classic characteristics of a distributed, networked system.

Why are we not programming it as such?

3 Implications

We now present some ideas, concepts, and principles from distributed systems which we feel are particularly relevant to OS design for modern machines. We draw parallels with existing ideas in operating systems, and where there is no corresponding notion, suggest how the idea might be applied. We have found that viewing OS problems as distributed systems problems either suggests new solutions to current problems, or fruitfully casts new light on known OS techniques.

Message passing vs. shared memory: Traversing a shared data structure in a modern cache-coherent system is equivalent to a series of synchronous RPCs to fetch remote cache lines. For a complex data structure, this means lots of round trips, whose performance is limited by the latency and bandwidth of the interconnect.

In distributed systems, "chatty" RPCs are reduced by encoding the high-level operation more compactly; in the limit, this becomes a single RPC or code shipping. When communication is expensive (in latency or bandwidth), it is more efficient to send a compact message encoding a complex operation than to access the data remotely. While there was much research in the 1990s into distributed shared virtual memory on clusters (e.g. [1]), it is rarely used today.

In an OS, a message-passing primitive can make more efficient use of the interconnect and reduce latency over sharing data structures between cores. If an operation on a data structure and its results can each be compactly encoded in less than a cache line, a carefully written and efficient user-level RPC [3] implementation which leaves the data where it is in a remote cache can incur less overhead in terms of total machine cycles. Moreover, the use of message passing rather than shared data facilitates interoperation between heterogeneous processors.

Note that this line of reasoning is independent of the overhead for synchronization (e.g. through scalable locks). The performance issues arise less from lock contention than from data locality issues, an observation which has been made before [2].

[2]Numbers for main memory are complex due to the broadcast nature of the cache coherence protocol, and do not add to our argument.

Explicit message-based communication is amenable to both informal and formal analysis, using techniques such as process calculi and queueing theory. In contrast, although they have been seen by some as easier to program, it is notoriously difficult to prove correctness results about, or predict the performance of, systems based on implicit shared-memory communication.

Long ago, Lauer and Needham [11] pointed out the equivalence of shared-memory and message passing in OS structure, arguing that the choice should be guided by what is best supported by the underlying substrate. More recently, Chaves *et al.* [7] considered the same choice in the context of an operating system for an early multiprocessor, finding the performance tradeoff biased towards message passing for many kernel operations. We claim that hardware today strongly motivates a message-passing model, and refine this broad claim below.

Replication of data is used in distributed systems to increase throughput for read-mostly workloads and to increase availability, and there are clear parallels in operating systems. Processor caches and TLBs replicate data in hardware for performance, with the OS sometimes handling consistency as with TLB shootdown.

The performance benefits of replicating in software have not been lost on OS designers. Tornado [10] replicated (as well as partitioned) OS state across cores to improve scalability and reduce memory contention, and fos [17] argues for "fleets" of replicated OS servers. Commodity OSes such as Linux now replicate cached pages and read-only data such as program text [4].

However, replication in current OSes is treated as an optimization of the shared data model. We suggest that it is useful to instead see replication as the *default model*, and that programming interfaces should reflect this – in other words, OS code should be written as if it accessed a local replica of data rather than a single shared copy.

The principal impact on clients is that they now invoke an agreement protocol (propose a change to system state, and later receive agreement or failure notification) rather than modifying data under a lock or transaction. The change of model is important because it provides a uniform way to synchronize state across heterogeneous processors that may not coherently share memory.

At scale we expect the overhead of a relatively long-running, split-phase (asynchronous) agreement protocol over a lightweight transport such as URPC to be less than the heavyweight global barrier model of inter-processor interrupts (IPIs), particularly as the usual batching and pipelining optimizations also apply with agreement. In effect, we are trading increased latency off against lower overhead for operations.

Of course, sometimes sharing is cheap. Replication of data across closely coupled cores, such as those sharing L2 or L3 cache, is likely to perform substantially slower than spinlocks. However, we can reintroduce sharing and locks (or transactions) for groups of cores as an optimization behind the replication interface.

This is a reversal of the scalability trend in kernels whereby replication and partitioning is used to optimize locks and sharing; in contrast, we advocate using locks and limited sharing to optimize replica maintenance.

Consistency: Maintaining the consistency of replicas in current operating systems is a fairly simple affair. Typically an initiator synchronously contacts all cores, often via a global IPI, and waits for a confirmation. In the case of TLB shootdown, where this occurs frequently, it is a well-known scalability bottleneck. Some existing optimizations for global TLB shootdown are familiar from distributed systems, such as deferring and coalescing updates. Uhlig's TLB shootdown algorithm [16] appears to be a form of asynchronous single-phase commit.

However, the design space for agreement and consensus protocols is large and mostly unexploited in OS design. Operations such as page mapping, file I/O, network connection setup and teardown, etc. have varying consistency and ordering requirements, and distributed systems have much to offer in insights to these problems, particularly as systems become more concurrent and diverse.

Furthermore, the ability of consensus protocols to agree on ordering of operations even when some participants are (temporarily) unavailable seems relevant to the problem of ensuring that processors which have been powered down subsequently resume with their OS state consistent before restarting processes. This is tricky code to write, and the OS world currently lacks a good framework within which to think about such operations.

Just as importantly, reasoning about OS state as a set of consistent replicas with explicit constraints on the ordering of updates seems to hold out more hope of assuring correctness of a heterogeneous multiprocessor OS than low-level analysis of locking and critical sections.

Network effects: Perhaps surprisingly, routing, congestion, and queueing effects within a computer are already an issue. Conway and Hughes [8] document the challenges for platform firmware in setting up routing tables and sizing forwarding queues in a HyperTransport-based multiprocessor, and point out that link congestion (as distinct from memory contention) is a performance problem, an effect we have replicated on AMD hardware in our lab. These are classical networking problems.

Closer to the level of system software, routing problems emerge when considering where to place buffers in memory as data flows through processors, DMA controllers, memory, and peripherals. For example, data that arrives at a machine and is immediately forwarded back over the network should be placed in buffers close to the

NIC, whereas data that will be read in its entirety should be DMAed to memory local to the computing core [14].

Heterogeneity (and interoperability) have long been key challenges in distributed systems, and have been tackled at the data level using standardized messaging protocols, and at the interface level using logical descriptions of distributed services that software can reason about (e.g. [9]), the most ambitious being the ontology languages used for the semantic web.

We have proposed an analogous, though simpler, approach based on constraint logic programming to allow an OS (and applications) to make sense of the diverse and complex hardware on which it finds itself running [14].

4 The multikernel architecture

In this section, we sketch out the *multikernel architecture* for an operating system built from the ground up as a distributed system, targeting modern multicore processors, intelligent peripherals, and heterogeneous multiprocessors, and incorporating the ideas above.

We do not believe that this is the only, or even necessarily the best, structure for such a system. However, it is useful for two related reasons. Firstly, it represents an extreme point in the design space, and hence serves as a useful vehicle to investigate the full consequences of viewing the machine as a networked system. Secondly, it is designed with total disregard for compatibility with either Windows or POSIX. In practice, we can achieve compatibility with sub-optimal performance by running a VMM over the OS [13], and this gives us the freedom to investigate OS APIs better suited to both modern hardware and the scheduling and I/O requirements of modern concurrent language runtimes.

Message-based communication: Cross-core sharing in a multikernel is avoided by default; instead, each core runs its own, local *OS node*, as shown in Figure 2.

In keeping with the message-passing model, all communication between cores is asynchronous (split-phase). On current commodity hardware, we use URPC [3] as our message transport, with recourse to IPIs only when necessary for synchronization. Other transports may be used over non-coherent links (such as PCIe), or for future hardware. Other than URPC buffers, *no data structures whatsoever* are shared between OS nodes. One of the consequences of this is that the OS code for different cores can be implemented entirely differently (and, for example, specialized for a given architecture).

Replication and consistency: Global consistency of replicated state in the OS is handled by message passing between OS nodes. All OS operations from user-space processes that affect global state are split-phase, and are

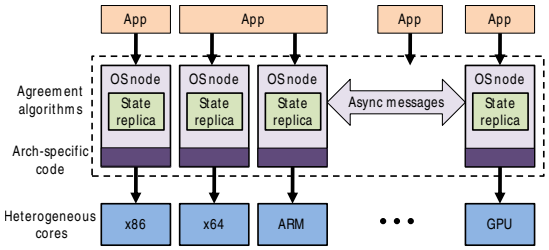

Figure 2: The multikernel architecture

mediated by the local OS node, which obtains agreement where required from the other OS nodes in the system and performs privileged operations locally.

For example, an OS node would change a user-space memory mapping by a distributed two-phase commit, first tentatively changing the local mapping, then initiating agreement among other cores possessing the mapping, and finally committing the change (or aborting if a conflicting mapping on another core preempted it).

Heterogeneity: We address heterogeneity in two ways. First, all or part of the OS node can be specialized to the core on which it runs. Second, we maintain a rich and detailed representation of machine hardware and performance tradeoffs in a service [14] that facilitates online reasoning about code placement, routing, buffer allocation, etc.

5 Open questions

We do not advocate blindly importing distributed systems ideas into OS design, and applying such ideas usefully in an OS is rarely straightforward. However, many of the challenges lead to their own interesting research directions.

What are the appropriate algorithms, and how do they scale? Intuitively, distributed algorithms based on messages should scale just as well as cache-coherent shared memory, since at some level the latter is based on the former. Passing compact encodings of complex operations in our messages should show clear wins, but this is still to be demonstrated empirically. It is ironic that, years after microkernels failed due to the high cost of messages versus memory access, OS design may adopt message passing for the opposite reason.

There are many more relevant areas in distributed computing than we have mentioned here. For example, leadership and group membership algorithms may be useful in handling hot-plugging of devices and CPUs.

Second-guessing the cache coherence protocol. On current commodity hardware, the cache coherence pro-

tocol is ultimately our message transport. Good performance relies on a tight mapping between high-level messages and the movement of cache lines, using techniques like URPC. At the same time, cache coherence provides facilities (such as reliable broadcast) which may permit novel optimizations in distributed algorithms for agreement and the like. To further complicate matters, some parts of the machine (such as main memory) may be cache-coherent but others (such as programmable NICs and GPUs) might not be, and our algorithms should perform well over this heterogeneous network.

The illusion of shared memory. Our focus is scaling the OS, and thereby improving performance of OS-intensive workloads, but the same arguments apply to application code, an issue we intend to investigate. Naturally, a multikernel should provide applications with a shared-memory model if desired. However, while it could be argued that shared memory is a simpler programming model for applications, systems like Disco [6] have been motivated by the opposite claim: that it is easier to build a scalable application from nodes (perhaps small multiprocessors) communicating using messages.

A separate question concerns whether future multicore designs will remain cache-coherent, or opt instead for a different communication model (such as that used in the Cell processor). A multikernel seems to offer the best options here. As in some HPC designs, we may come to view scalable cache-coherency hardware as an unnecessary luxury with better alternatives in software.

Where does the analogy break? There are important differences limiting the degree to which distributed algorithms can be applied to OS design. Many arise from the hardware-based message transport, such as fixed transfer sizes, no ability to do in-network aggregation, static routing, and the need to poll for incoming messages. Others (reliable messaging, broadcast, simpler failure models) may allow novel optimizations.

Why stop at the edge of the box? Viewing a machine as a distributed system makes the boundary between machines (traditionally the network interface) less clear-cut, and more a question of degree (overhead, latency, bandwidth, reliability). Some colleagues have therefore suggested extending a multikernel-like OS across physical machines, or incorporating further networking ideas (such as Byzantine fault tolerance) within a machine. We are cautious (even skeptical) about these ideas, even in the long term, but they remain intriguing.

Perhaps less radical is to look at how structuring a single-node OS as a distributed system might make it more suitable as part of a larger physically distributed system, in an environment such as a data center.

6 Conclusion

Modern computers are inherently distributed systems, and we miss opportunities to tackle the OS challenges of new hardware if we ignore insights from distributed systems research. We have tried to come out of denial by applying the resulting ideas to a new OS architecture, the multikernel.

An implementation, Barrelfish, is in progress.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2), 1996.

[2] J. Appavoo, D. Da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM TOCS*, 25(3), 2007.

[3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM TOCS*, 9(2):175–198, 1991.

[4] M. J. Bligh, M. Dobson, D. Hart, and G. Huizenga. Linux on NUMA systems. In *Ottawa Linux Symp.*, Jul 2004.

[5] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proc. 8th OSDI*, Dec 2008.

[6] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM TOCS*, 15(4):412–447, 1997.

[7] E. M. Chaves, Jr., P. C. Das, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel–Kernel communication in a shared-memory multiprocessor. *Concurrency: Pract. & Exp.*, 5(3), 1993.

[8] P. Conway and B. Hughes. The AMD Opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[9] J.-P. Deschrevel. The ANSA model for trading and federation. Architecture Report APM.1005.1, APM Ltd., Jul 1993. http://www.ansa.co.uk/ANSATech/93/Primary/100501.pdf.

[10] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. 3rd OSDI*, 1999.

[11] H. C. Lauer and R. M. Needham. On the duality of operating systems structures. In *Proc. 2nd Int. Symp. on Operat. Syst., IRIA*, 1978. reprinted in *ACM Operat. Syst. Rev.*, 13(2), 1979.

[12] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9:21–65, 1991.

[13] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and virtue. In *Proc. 11th HotOS*, San Diego, CA, USA, May 2007.

[14] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Workshop on Managed Many-Core Systems*, Boston, MA, USA, Jun 2008.

[15] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. 19th SOSP*, pages 207–222, 2003.

[16] V. Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Germany, Jun 2005.

[17] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *Operat. Syst. Rev.*, 43(2), Apr 2009.