Mind the Gap: Reconnecting Architecture and OS Research

Jeffrey C. Mogul
Jeff.Mogul@hp.com
HP Labs, Palo Alto, CA

Andrew Baumann
Andrew.Baumann@microsoft.com
Microsoft Research, Redmond, WA

Timothy Roscoe
troscoe@inf.ethz.ch
ETH Zurich

Livio Soares
livio@eecg.toronto.edu
University of Toronto

Abstract

The goal of a computer system is to run an application workload securely, reliably, efficiently, and fast. A computer's hardware architecture and operating system exist to support this goal, and it would be nice if they cooperated as effectively as possible. Yet there is a growing gap between architectural research and OS research, which seems to be the result of poor communication about what actually matters.

In this paper, we discuss this gap and what to do about it. We illustrate the opportunities for closing the gap using examples from some recent OS research.

1 Introduction

For too long, operating systems researchers and developers have pretty much taken whatever computer architects have dished out. With occasional exceptions (e.g., virtualization support), architecture researchers do not appear to have sought or encouraged innovations that would improve the execution environment for an OS. Even worse, many do not bother to simulate and report on OS behavior when evaluating their proposals.

Times have changed: architects are running out of new ideas that lead to significant application-level performance improvements; we must now rely on improved parallelism. But parallelism stresses the very issues that operating systems research has focused on: distribution, resource management, I/O, etc. Also, many modern applications spend significant execution time in OS functions; it really does matter whether a CPU works well on OS code. We believe that closer collaboration between OS and architecture researchers could yield real benefits.

Alas, the disconnect between OS and architectural research seems to be growing, at a time when we should be trying to shrink it. In this paper, we discuss some problems arising from this gap, try to identify its causes, and consider ways to bridge it. We illustrate our discussions with examples drawn from recent OS research.

2 What has caused the gap?

Scientific computing papers often, and somewhat amusingly, refer to CPU time spent in the OS as "noise" (e.g., [19]). Perhaps for HPC users, the operating system really is just an annoyance, but for most computers, from sensor-net nodes through handhelds and laptops to servers, the OS does useful work, and often a lot of it.

There is some evidence that, for many real-world applications, plenty of execution time is spent in the OS [9]. (We assume a loose definition of "the OS" – it's more than just the kernel, since in many cases people have moved OS functionality into user-mode libraries.) Perhaps this is not yet *ample* evidence, although we suspect this is mostly for lack of a systematic study.

But computer architects, from the evidence available in the scientific literature, assume that the OS does not exist – except perhaps when it magically manages application-thread resources that hardware cannot. Architecture papers commonly use application-only benchmarks, and seldom account for the interference between application and OS execution (there are, of course, counter-examples [9]). In short, while architecture papers sometimes pay lip service to the OS, they rarely discuss the impact of architecture on OS behavior.

Meanwhile, a typical OS paper usually uses the phrase "on commodity hardware". As a community, we assume we are stuck with whatever flaws the hardware has.

We see several infrastructural reasons why architecture researchers have been ignoring the OS: lack of quantitative evidence for the importance of OS execution; lack of an effective simulation environment; and lack of appropriate benchmarks.

2.1 How important is OS execution?

One obvious cultural difference between OS and architecture researchers is that OS researchers implicitly assume that operating system performance matters. But how do we *know* that OS performance really does matter? We typically test our system improvements on a small set of applications, or even microbenchmarks, and then (usually with scant evidence) generalize to declare that we've done something truly useful. Various papers that have tried to demonstrate the importance of OS execution (e.g., [8, 11]) generally pick a few applications for their benchmarks, so it is hard convince architecture researchers that, in the general case, OS execution matters.

We know of no quantitative study that has attempted to measure the importance of OS execution in the wild rather than on benchmarks. As a baby step in this direction, we obtained `collectl` logs[1] from several production Linux servers running a variety of applications: a weather forecasting system (WRF), NFS server, PolyServe storage system (PS), and academic Grid system, with unknown workload. These logs contain samples taken every 1–10 seconds (they were originally collected

[1] http://collectl.sourceforge.net/

for various other purposes) of the percentage of time spent in user and in kernel mode, from which we computed the fraction of non-idle CPU time in kernel mode.[2]

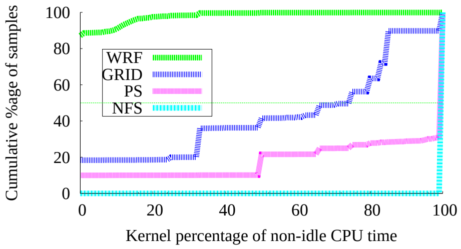

Figure 1: CDFs of non-idle time in kernel mode

Fig. 1 shows that kernel-mode execution varies from almost 0% of non-idle cycles (for WRF, a compute-bound program) to almost 100% (for NFS, which runs entirely in kernel mode); both PS and GRID spend a considerable fraction of time in kernel mode. While the systems in this small and arbitrary sample might not be representative, the results suggest that OS execution cycles really do matter.

2.2 The limitations of simulation

Given the complexity of modern silicon, it has been almost impossible to do architectural research except via simulation. (Hardware emulation is just now becoming a viable option, which we discuss further in Sec. 5.) However, most widely-available, well-supported simulators do not do full-system simulation of low-level architectural behavior – that is, you can't run a real OS on them. There are a few exceptions, such as M5 [2] and Mambo [3], but until recently, these did not support the x86 ISA (M5's x86 support has just recently become barely usable). So it has been hard to validate against modern hardware, or to make a truly convincing case.

The lack of a cycle-level simulator that can run a real operating system has created a dilemma for architectural researchers, and appears to be one of the main reasons why they tend to ignore OS code.

Is x86 the problem? The x86 architecture currently dominates the server and desktop markets, notwithstanding some recent inroads by others, and OS researchers and developers have focussed most of their recent efforts on x86. Because of the simulator problem, however, some architects have resisted using x86.

Dean Tullsen [20] remarks that x86 is so idiosyncratic and complex that doing architectural research on an x86 simulator forces one to "spend all your time solving things specifically broken on the x86 rather than fundamental architectural problems." He thus intentionally uses a "dead ISA" – but this makes it harder to incorporate a modern OS.

2.3 Inappropriate or inadequate benchmarks

Architecture researchers believe in quantitative measurements, which is good; they believe in shared benchmarks, which is good; but the benchmarks they use (SPEC CPU, SPLASH, PARSEC) seldom involve the operating system. This is bad.

There are some benchmarks that stress operating system functions (e.g., SPECweb, RUBiS, TPC-W).[3] Architecture researchers almost never use these, for a few reasons. First, they are hard to get running, and often have complex parameter settings. Second, we often want to simulate networks of computers, and this greatly complicates the problem of getting something running.

Third, even if one has a full-system simulator, getting results in a reasonable number of days requires running the benchmark for just a few seconds (or less) of simulated time, requiring the benchmark to be seriously perverted – most of these benchmarks are not designed to give useful results so quickly. David Patterson has observed that architects do not understand whether this time is enough to provide valid results on OS-intensive benchmarks [10]; also, architects do not understand how many simulation trials are required for statistical validity, given that OS behavior is often non-deterministic.

Dean Tullsen has observed that "history has shown that the best way to spawn a ton of [architecture] research is to provide a tool," such as a new benchmark suite. But the challenge is to make a benchmark as easy to run as SPECint – with OS benchmarks, "minor things in the OS create way too much noise in the results." [20] If we want architecture researchers to think about support for operating systems, we will probably need to help them with a suite of benchmarks that stress the things that we care about, and are pre-packaged to run easily (without a lot of thought about parameters) on cycle-level simulators.

There are a lot of possibilities for OS-relevant benchmarks; in addition to those listed above, one might include Hadoop, or a virtualization management workload [15]. (Micro-benchmarks, such as LMbench, also have their uses, but must be used with care. Null systemcalls don't exercise TLB coverage or cache flush penalty, which only kicks in after a few hundred cycles.) The trick will be getting them to run, with useful results, on a simulator: e.g., a benchmark that expresses the "essence of Hadoop" in just 1 second.

3 Design principles for architecture

In this section, we outline design principles for new features or extensions to processor architecture, illustrated by what we see as missed opportunities in previous extensions, which failed to consider the needs of the OS.

P1: Facilitate resource multiplexing. The role of an OS is to share the hardware and enforce protection be-

[2] We recognize that "kernel mode" and "operating system" can be different things, even on a monolithic kernel, but `collectl` cannot measure time spent in daemons, libraries, etc.

[3] Modern applications are often composed of many interacting processes or threads, but even these benchmarks don't really exercise this.

tween applications. Thus, any new hardware functionality must be efficiently multiplexed (or virtualized) by the OS. Often good ideas are rendered useless because they assume there's only one program running.

For example, Intel's recent Single-chip Cloud Computer[4] (SCC) [6] provides 8KB of scratchpad memory per core to support fast inter-core message passing. Since an 8KB region is mapped by only two pages, mediating access to this resource forces the use of kernel mode to send a message. Kernel entry cost ($\approx 10\mu$s) overwhelms the underlying message-transfer cost (only 10ns).

P2: Keep mechanisms orthogonal. New features should be orthogonal to existing ones, not overlaid on them. One cannot anticipate all uses of a feature, so coupling it to another architectural mechanism needlessly restricts usability.

For example, the AMD-V virtualization extensions support a tagged TLB (an old idea but new to x86) which allows an expensive TLB flush to be avoided when context switching. However, since the tag was added to the VM control block, TLB tags are only usable by VM guests, and the TLB must still be flushed when switching between non-VM address spaces.

A corollary is: **Don't enforce arbitrary limits.** OSes are good at virtualizing finite resources so that they look (almost) infinite, and architects should not stymie this process. For example, the limited size of tags in TLBs is not a problem if they are exposed: the OS uses them as a cache for a larger set of process identifiers.

P3: Avoid over-abstraction. Hardware designers should not abstract new processor functionality and features from the OS, nor allow (legitimate) concerns about backward compatibility to prevent the future use of new mechanisms in unintended ways.

For example, Intel's introduction of simultaneous multithreading (hyper-threading) to the x86 architecture was completely backwards-compatible. So as to require no changes to existing multiprocessor OSes, hardware threads appear simply as processors. An unfortunate consequence is that there is no efficient way for code running on one thread of a multi-threaded core to observe the state of, interrupt or signal the other thread: it must raise a heavyweight inter-processor interrupt (IPI).

P4: Stay independent of kernel architecture. With suitable checks, all new mechanisms should work correctly and efficiently in user mode as well as kernel mode. Without rehashing the microkernel debate, we claim many OS functions need not, and should not, incur the cycle cost or security risks of kernel entry.

For example, AMD's proposed advanced synchronization facility (ASF) [4] adds hardware support for memory transactions via instructions that register interest in an area of memory (e.g. a set of cache lines) and work like `setjmp`: subsequent writes to the region by another core cause a return to the instruction, having discarded all local writes to the region, and with a register value used to conditionally branch to rollback code.

While a plausible implementation of transaction rollback, this facility could have had a range of other (possibly more important) applications, e.g. the notification mechanism we propose in Sec. 4.1. Unfortunately, this is another case of over-abstraction: while it could be expressed as a *simpler* asynchronous subroutine call (saving state), one register value is clobbered, making it impossible to resume from the abort point.

Worse, it assumes a specific OS structure and kernel-mediated usage model: the state saved on abort (program counter and stack pointer) is only available in kernel mode through model-specific registers (MSRs), requiring both an expensive trap and a slow (10s or 100s of cycles) instruction to access.

4 Modest proposals

We offer examples of hardware features that could support recent advances in OS research. We draw on our own published work to make the discussion concrete, but are not the only researchers to propose these facilities.

4.1 Inter-core communication

Construction of a scalable OS can benefit from primitives for inter-core communication and coordination:

Lightweight inter-core messages: Your computer is a distributed system [1], and we argued that the OS should be viewed as a distributed system, communicating internally by messages rather than shared memory. But, as we wrote, "On current commodity hardware" (that phrase again!) "the cache coherence protocol is ultimately our message transport."

True core-to-core message-passing would be better – a mechanism which already exists to support cache coherence, but is not exposed to software. This mechanism could be exposed as a means to proactively write data to a remote core's cache or scratchpad memory, avoiding the stall imposed by the cache-coherence protocol for the receiver to fetch the message payload. Beehive [18] and SCC provide this, but SCC doesn't allow safe, efficient use by multiple applications, and Beehive doesn't try.

An alternative core-to-core messaging interface could allow a sender to insert cache lines into a remote core's caches. This would be advisory: as with software prefetch instructions, lines would be transferred with low priority to a mid-level cache (e.g. L2), allowing data to be placed close to where it will be used without impacting the execution of the remote core or stalling the sender.

Lightweight inter-core notifications: Efficient data transfer is not enough: messages require notification

[4] We are grateful for Intel for access to SCC hardware and many helpful discussions.

too.[5] Such a facility must be extremely lightweight, but not necessarily reliable (one can always fall back to polling). Unfortunately, today IPIs are the only option.

We favor lightweight control transfer, whereby a core is vectored to a specific instruction address in the same protection domain, saving only the minimum context needed to resume afterward. We want to change the control flow of another core *iff a specific application is running*, not just notify an entire core. Authorization is required: if the target core is not running the application, the sender can fall back on IPIs or wait for polling.

4.2 Faster system calls

On many (not all) architectures, a kernel crossing is alarmingly expensive, and includes much unnecessary state management. Processor designers have devoted little effort to efficient context switching.

FlexSC [13] attempts to alleviate this for Linux using exception-less system calls, and uses shared memory and software polling for inter-core system calls. FlexSC is forced to batch multiple invocations into a single control transfer to amortize the cost of both same-core and inter-core system calls. Better architectural support for inter-core communication, combined with a lightweight local kernel call, would make FlexSC much more efficient.

4.3 Software controlled cache management

Caches are a large (and increasing) factor in software performance, but are mostly functionally transparent to software. We believe exposing greater control over cache behavior to the OS could have significant performance benefits. We discuss two examples.

Controlling cache coherence: Most commercial architectures implement system-wide cache-coherence, but as the number of cores and private caches grows, mandatory coherence looks increasingly like over-abstraction of resources. Partly for this reason, some experimental systems such as SCC and Beehive are non-coherent, and some research OSes like Barrelfish run without cache coherence. Given that we will run applications that rely on hardware coherence, the hardware should allow the OS to turn off coherence when we know it is not needed, e.g. for an address space, region, or core.

Selective coherence introduces challenges: turning coherence on or off requires explicit cache flushing or invalidation. This must be possible from user mode, so that applications can directly manage non-coherent memory.

Software managed cache replacement: Caches are getting bigger. Hardware-only replacement policies are becoming difficult and inefficient, and the OS should manage this real-estate – it has semantic information not available to hardware. Dynamic partitioning of shared caches [16] or reducing space allocated to “polluting” memory pages [14] improves performance. Today, these techniques must be implemented indirectly using *page coloring*, which is crude and expensive. Architects and OS researchers should collaborate to design a clean, efficient interface.

4.4 Better performance counters

One traditional OS function is to manage resources for contending processes and threads. As hardware becomes more complex, with more resources to consider, this task becomes harder, largely because the details are hidden from the OS. Hardware performance counters (HPCs) would seem to be an ideal interface between hardware and OS, and potentially allow the OS to improve application performance significantly [12, 14, 16]. However, making these techniques useful beyond research OSes requires architectural change.

We get the impression that chip designers underestimate the value of HPCs, and do not really understand how software could use them. (These problems are *not* the fault of architecture researchers, who have tried to get chip vendors to fix things, and failed [10].) We urge architects and vendors to consider HPCs as a general OS facility, rather than simply support for code optimization.

Most HPCs are poorly documented, and inconsistent even between CPUs from the same vendor. HPCs should be usable outside the kernel, and therefore easy for the OS to multiplex among applications [22]. There are too few HPCs to be generally useful, so one has to virtualize them, which causes inaccuracy. Sometimes the overhead of using them is too high, which necessitates complex tricks to minimize uses.

Flexible specification of the events counted (filters on event parameters) would greatly reduce both overhead and complexity of using HPCs within the OS. Furthermore, enhancing the HPCs with context logging functionality (such as that provided with Intel’s precise event-based sampling (PEBS) facility) allows the OS to infer much information about memory operations, and the ability to spill event counts to memory avoids the trap of arbitrary limits on HPCs as a resource. A combination of PEBS-style state logging and filter programmability would open up a range of OS design ideas.

5 Closing the gap

If OS researchers merely complain or issue wish-lists to the architecture community, this will not encourage collaboration. We suggest concrete steps to help draw the communities together – these are starting points that raise further questions, and maybe even research directions.

Venues: High-profile publishing venues can motivate collaboration between researchers. ASPLOS was created to bring OS and architecture researchers together, but while both communities attend, very few submissions fo-

[5] Architects (including the SCC designers) seem to ignore notification, perhaps due to a focus on HPC workloads where one application spins on a barrier or to receive.

cus on work which includes both architecture and OS research. Perhaps PC chairs and members could make an effort to encourage collaborative papers.

Emulators: Sec. 2.2 described numerous problems with simulation. David Patterson says "simulators are dead" – they do not parallelize, and since CPU cores are not getting faster, the ratio of simulated to elapsed time is not going to increase [10]. Instead, FPGA-based re-programmable emulators, already extensively used by processor vendors, are the only effective way to test architectural ideas on long-running benchmarks [17]. Such hardware is now available to academics [5, 21]. Some emulators [21] can run a real OS, but emulator builders face a tension between adding full OS support, and using time and gates to implement their own favorite designs.

For emulation to help bridge the gap, researchers must be able to learn a system and share results. The NetFPGA project [7] has been successful in supporting research into hardware packet routers, an area that previously was forbiddingly difficult due to lack of cooperation from router vendors.

Providing a solid, common platform for multi-core OS and architecture research will require considerable investment and support, but is likely to pay dividends.

Benchmarks: Since benchmarks play such a large role in driving architecture research, they should be aligned with the needs of OS implementation. The high-level goal for such benchmarks is clear: they should encourage innovation in computer architecture and OS design which improves the performance of OS-intensive application workloads. Even the process of devising (and, subsequently revising) suitable benchmarks will have to involve representatives from both communities, and thereby help to close the gap.

Microbenchmarks are a good starting point, e.g. measuring kernel entry or exit, interrupt latency, context switch time (suitably defined), etc. Microbenchmarks are easier to decouple from a particular operating system, encouraging innovation in that space as well.

This in turn means these benchmarks should measure the performance of a combination of hardware and software. If the software is fixed by the benchmark (e.g. as with SPEC), it is hard to introduce architectural innovations in the OS programming interface to hardware. Furthermore, the benchmarks must deliver valid results in brief runs on simulators, or must be ported to (not just "portable to") emulation-based platforms.

6 Summary

Architecture researchers should explore how to support OS execution, and view the OS as a performance-critical part of a system. However, they should not view it as fixed (Linux or Windows) but open a dialog with OS researchers about what is possible or desirable. Conversely, OS researchers should not blindly accept commodity hardware, and open their minds to evolving architecture alongside OS design.

That said, we are wary of uncritically embracing the idea that hardware can be changed arbitrarily to suit the OS, or vice versa. The constraint of commodity hardware in OS research has tended to keep us honest, and we do not want to see papers justify designs despite serious problems that "can be fixed with suitable hardware."

In the end this requires not merely dialog, but close collaboration at the hardware/software interface.

References

[1] A. Baumann et al. Your computer is already a distributed system. Why isn't your OS? In *Proc. HotOS*, 2009.

[2] N. L. Binkert et al. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[3] P. Bohrer et al. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Perf. Eval. Rev.*, 31(4):8–12, 2004.

[4] J. Chung et al. ASF: AMD64 extension for lock-free data structures and transactional memory. In *Proc. MICRO*, Dec. 2010.

[5] J. D. Davis, C. P. Thacker, and C. Chang. BEE3: Revitalizing computer architecture research. Tech. Rep. MSR-TR-2009-45, Microsoft Research, 2009.

[6] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proc. ISSCC*, Feb. 2010.

[7] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: reusable router architecture for experimental research. In *Proc. PRESTO*, pages 1–7, 2008.

[8] D. Nellans, R. Balasubramonian, and E. Brunvand. Interference aware cache designs for operating system execution. Tech. Rep. UUCS-09-002, U. Utah, Feb. 2009.

[9] D. Nellans, R. Balasubramonian, and E. Brunvand. OS execution on multi-cores: Is out-sourcing worthwhile? *SIGOPS Oper. Syst. Rev.*, 43(2):104–105, 2009.

[10] D. Patterson. Pers. comm., 2010.

[11] J. A. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proc. ASPLOS*, pages 245–256, Nov. 2000.

[12] K. Shen et al. Hardware counter driven on-the-fly request signatures. In *Proc. ASPLOS*, pages 189–200, 2008.

[13] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. OSDI*, Oct. 2010.

[14] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Proc. MICRO*, pages 258–269, Nov. 2008.

[15] V. Soundararajan and J. M. Anderson. The impact of management operations on the virtualized datacenter. In *Proc. ISCA*, pages 326–337, 2010.

[16] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. ASPLOS*, pages 121–132, 2009.

[17] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A case for FAME: FPGA architecture model execution. In *Proc. ISCA*, pages 290–301, 2010.

[18] C. Thacker. *Beehive: A many-core computer for FPGAs (v5)*. MSR Silicon Valley, Jan. 2010. http://projects.csail.mit.edu/beehive/BeehiveV5.pdf.

[19] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proc. ICS*, pages 303–312, 2005.

[20] D. Tullsen. Pers. comm., 2010.

[21] J. Wawrzynek et al. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2):46–57, Mar. 2007.

[22] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *Proc. HotOS*, 2007.