Agile Paging for Efficient Memory Virtualization

Virtualization provides benefits for many workloads, but the associated overhead is still high. The cost comes from managing two levels of address translation with either nested or shadow paging. This article introduces agile paging, which combines the best of both nested and shadow paging within a page walk to exceed the performance of both techniques.

Jayneel Gandhi
Mark D. Hill
Michael M. Swift
University of Wisconsin—Madison

Two important trends in computing are evident. First, computing is becoming more data-centric, wherein low-latency access to a very large amount of data is critical. Second, virtual machines are playing an increasingly critical role in server consolidation, security, and fault tolerance as substantial amounts of computing migrate to shared resources in cloud services. Because software accesses data using virtual addresses, fast address translation is a prerequisite for efficient data-centric computation and for providing the benefits of virtualization to a wide range of applications. Unfortunately, the growth in physical memory sizes is exceeding the capabilities of the most widely used virtual memory abstraction—paging—which has worked for decades.

Translation look-aside buffer (TLB) sizes have not grown in proportion to growth in memory sizes, causing a problem of limited TLB reach: the fraction of physical memory that TLBs can map reduces with each hardware generation. There are two key factors causing limited TLB reach: first, TLBs are on the critical path of accessing the L1 cache and thus have remained small in size, and second, memory sizes and the workload's memory demands have increased exponentially. This has introduced significant performance overhead due to TLB misses causing hardware page walks. Even the TLBs in the recent Intel Skylake processor architecture cover only 9 percent of a 256-Gbyte memory. This mismatch between TLB reach and memory size will keep growing with time.

In addition, our experiments show virtualization increases page-walk latency by two to three times compared to unvirtualized execution. The overheads are due to two levels of page tables: one in the guest virtual machine (VM) and the other in the host virtual machine monitor (VMM). There are two techniques to manage these two levels of page tables: nested paging and shadow paging. In this article, we explain the tradeoffs between the two techniques that intrinsically lead to high overheads of virtualizing memory. With current hardware and software, the overheads of virtualizing memory are hard to minimize, because a VM exclusively uses one technique or the other. This effect, combined with limited TLB reach, is detrimental for many virtualized applications and makes virtualization unattractive for big-memory applications.[1]

This article addresses the challenge of high overheads of virtualizing memory in a

Published by the IEEE Computer Society

Table 1. Tradeoffs provided by memory virtualization techniques as compared to base native. Agile paging exceeds the best of both worlds.

Properties	Base native	Nested paging	Shadow paging	Agile paging
Translation look-aside buffer (TLB) hit	Fast (gVA→hPA)	Fast (gVA→hPA)	Fast (gVA→hPA)	Fast (gVA→hPA)
Memory accesses per TLB miss	4	24	4	Approximately 4 to 5 on average
Page table updates	Fast: direct	Fast: direct	Slow: mediated by the virtual machine monitor (VMM)	Fast: direct
Hardware support	Native page walk	Nested + native page walk	Native page walk	Nested + native page walk with switching

*gVA→hPA: guest virtual address to host physical address.

comprehensive manner. It proposes a hardware/software codesign called *agile paging* for fast virtualized address translation to address the needs of many different big-memory workloads. Our goal, originally set forth in our paper for the 43rd International Symposium on Computer Architecture,[2] is to minimize memory virtualization overheads by combining the hardware (nested paging) and software (shadow paging) techniques, while exceeding the best performance of both individual techniques.

Techniques for Virtualizing Memory

A key component enabling virtualization is its support for virtualizing memory with two levels of page tables:

- gVA→gPA: guest virtual address (gVA) to guest physical address translation (gPA) via a per-process guest OS page table.
- gPA→hPA: guest physical address to host physical address (hPA) translation via a per-VM host page table.

Table 1 shows the tradeoffs between nested paging and shadow paging, the two techniques commonly used to virtualize memory, and compares them to our agile paging proposal.

Nested Paging

Nested paging is a widely used hardware technique to virtualize memory. The processor has two hardware page-table pointers to perform a complete translation: one points to the guest page table (`gcr3` in x86-64), and the other points to the host page table (`ncr3`).

In the best case, the virtualized address translation has a hit in the TLB to directly translate from gVA to hPA with no overheads. In the worst case, a TLB miss needs to perform a nested page walk that multiplies overheads vis-à-vis native (that is, unvirtualized 4-Kbyte pages), because accesses to the guest page table also require translation by the host page table. Note that extra hardware is required for nested page walk beyond the base native page walk. Figure 1a depicts virtualized address translation for x86-64. It shows how page table memory references grow from a native 4 to a virtualized 24 references. Although page-walk caches can elide some of these references,[3] TLB misses remain substantially more expensive with virtualization.

Despite the expense, nested paging allows fast, direct updates to both page tables without any VMM intervention.

Shadow Paging

Shadow paging is a lesser-used software technique to virtualize memory. With shadow paging, the VMM creates on demand a shadow page table that holds complete translations from gVA→hPA by merging entries from the guest and host tables.

In the best case, as in nested paging, the virtualized address translation has a hit in the TLB to directly translate from gVA to hPA

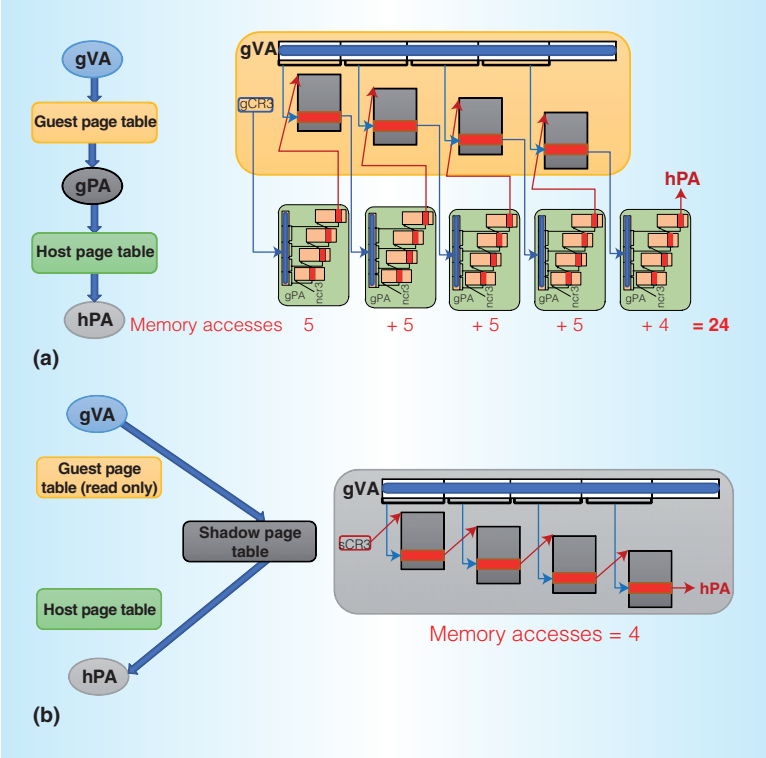

Figure 1. Nested paging has a longer page walk as compared to shadow paging, but nested paging allows fast, in-place updates whereas shadow paging requires slow, mediated updates (guest page tables are read-only). (a) Nested paging. (b) Shadow paging.

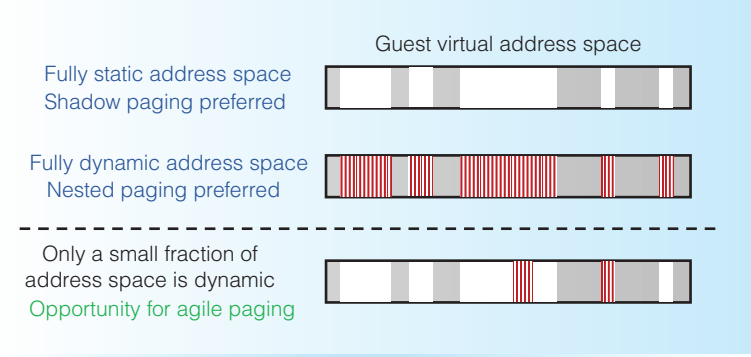

Figure 2. Opportunity that agile paging uses to improve performance. Portions in white denote static portions, stripes denote dynamic portions, and solid gray denotes unallocated portions of the guest virtual address space.

with no overheads. On a TLB miss, the hardware performs a native page walk on the shadow page table. The native page table pointer points to the shadow page table (`scr3`). Thus, the memory references required for shadow page table walk are the same as a base native walk. For example, x86-64 requires up to four memory references on a TLB miss for shadow paging (see Figure 1b). In addition, as a software technique, there is no need for any extra hardware support for page walks beyond base native page walk.

Even though TLB misses cost the same as native execution, this technique does not allow direct updates to the page tables, because the shadow page table needs to be kept consistent with guest and host page tables.[4] These updates occur because of various optimizations, such as page sharing, page migrations, setting accessed and dirty bits, and copy-on-write. Every page table update requires a costly VMM intervention to fix the shadow page table by invalidating or updating its entries, which causes significant overheads in many applications.

Opportunity

Shadow paging reduces overheads of virtualizing memory to that of native execution if the address space does not change. Our key observation is that empirically page tables are not modified uniformly: some regions of an address space see far more changes than others, and some levels of the page table, such as the leaves, are updated far more often than the upper-level nodes. For example, code regions might see little change over the life of a process, whereas regions that memory-map files might change frequently. Our experiments showed that generally less than 1 percent and up to 5 percent of the address space changes in a 2-second interval of guest application execution (see Figure 2).

Proposed Agile Paging Design

We propose agile paging as a lightweight solution to reduce the cost of virtualized address translation. We use the opportunity we just described to combine the best of shadow and nested paging by using

- shadow paging with fast TLB misses for the parts of the guest page table that remain static, and
- nested paging for fast in-place updates for the parts of the guest page tables that dynamically change.

In the following subsections, we describe the mechanisms that enable us to use both

constituent techniques at the same time for a guest process, and we discuss policies used by the VMM to select shadow or nested mode.

Mechanism: Hardware Support

Agile paging allows both techniques for the same guest process—even on a single address translation—using modest hardware support to switch between the two. Agile paging has three hardware architectural page table pointers: one each for shadow, guest, and host page tables. If agile paging is enabled, virtualized page walks start in shadow paging and then switch, in the same page walk, to nested paging if required.

To allow fine-grained switching from shadow paging to nested paging on any address translation at any level of the guest page table, the shadow page table needs to logically support a new switching bit per page table entry. This notifies the hardware page table walker to switch from shadow to nested mode. When the switching bit is set in a shadow page table entry, the shadow page table holds the hPA (pointer) of the next guest page table level. Figure 3a depicts the use of the switching bit in the shadow page table for agile paging. Figure 3b shows a page walk that is possible with agile paging. The switching is allowed at any level of the page table.

Mechanism: VMM Support

Like shadow paging, the VMM for agile paging manages three page tables: guest, shadow, and host. Agile paging's page table management is closely related to that of shadow paging, but there are subtle differences.

Guest page table (gVA→gPA). With all approaches, the guest page table is created and modified by the guest OS for every guest process. The VMM in shadow paging, though, controls access to the guest page table by marking its pages read-only. With agile paging, we leverage the support for marking guest page tables read-only with one subtle change. The VMM marks as read-only just the parts of the guest page table covered by the partial shadow page table. The rest of the guest page table (handled by nested mode) has full read-write access.

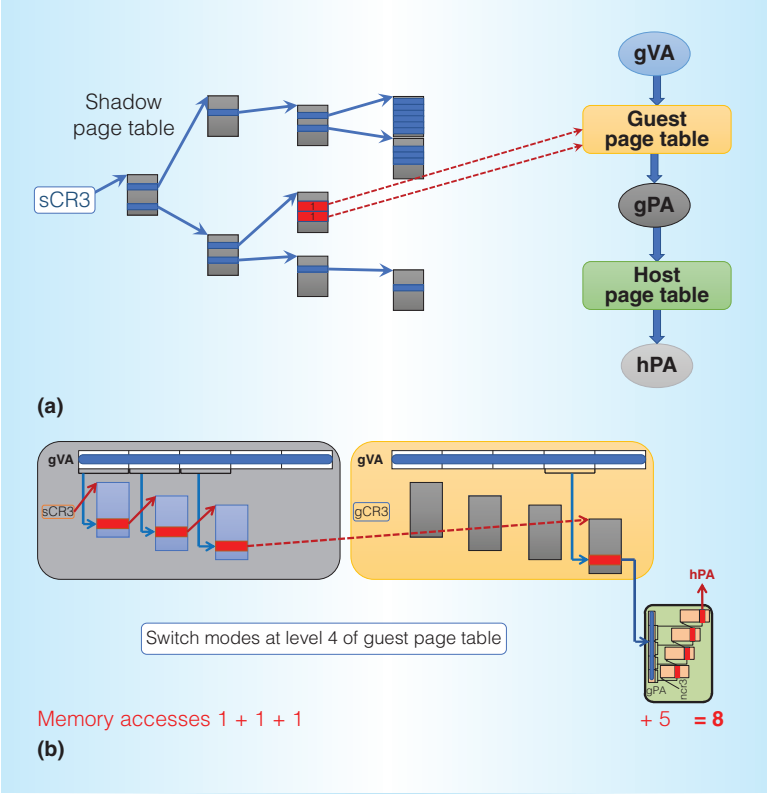

Figure 3. Agile paging support. (a) Mechanism for agile paging: when the switching bit is set, the shadow page table points to the next level of the guest page table. (b) Example page walk possible with agile paging, wherein it switches to nested mode at level four of the guest page table.

Shadow page table (gVA→hPA). For all guest processes with agile paging enabled, the VMM creates and maintains a shadow page table. However, with agile paging, the shadow page table is partial and cannot translate all gVAs fully. The shadow page table entry at each switching point holds the hPA of the next level of the guest page table with the switching bit set. This enables hardware to perform the page walk correctly with agile paging using both techniques.

Host page table (gPA→hPA). The VMM manages the host page table to map from gPA to hPA for each VM. As with shadow paging, the VMM merges this page table with the guest page table to create a shadow page table. The VMM must update the shadow page table on any changes to the host page table. The host page table is updated only by the VMM, and during that update,

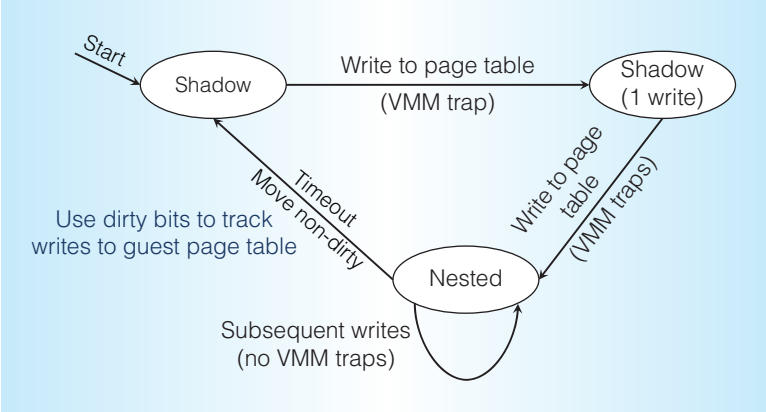

Figure 4. Policy to move a page between nested mode and shadow mode in agile paging.

the shadow page table is kept consistent by invalidating affected entries.

Policy: What Level to Switch?

Agile paging provides a mechanism for virtualized address translation that starts in shadow mode and switches at some level of the guest page table to nested mode. The purpose of a policy is to determine whether to switch from shadow to nested mode for a single virtualized address translation and at which level of the guest page table the switch should be performed.

The ideal policy would determine that page table entries are changing rapidly enough and the cost of corresponding updates to the shadow page table outweighs the benefit of faster TLB misses in shadow mode, and so translation should use nested mode. The policy would quickly detect the dynamically changing parts of the guest page table and switch them to nested mode while keeping the rest of the static parts of the guest page table under shadow mode.

To achieve this goal, a policy will move some parts of the guest page table from shadow to nested mode and vice versa. We assume that the guest process starts in full shadow mode, and we propose a simple algorithm for when to change modes.

Shadow→Nested mode. We start a guest process in the shadow mode to allow the VMM to track all updates to the guest page table (the guest page table is marked *read only* in shadow mode, requiring VMM interventions for updates). Our experiments showed that the updates to a single page of a guest page table are bimodal in a 2-second time interval: only one update or many updates (for example, 10, 50, 100). Thus, we use a two-update policy to move a page of the guest page table from shadow mode to nested mode: two successive updates to a page trigger a mode change. This allows all subsequent updates to frequently changing parts of the guest page table to proceed without VMM interventions.

Nested→Shadow mode. Once we move parts of the guest page table to the nested mode, all updates to those parts happen without any VMM intervention. Thus, the VMM cannot track if the parts under the nested mode have stopped changing and thus can be moved back to the shadow mode. So, we use dirty bits on the pages containing the guest page table as a proxy to find these static parts of the guest page table after every time interval, and we switch those parts back to the shadow mode. Figure 4 depicts the policy used by agile paging.

To summarize, the changes to the hardware and VMM to support agile paging are incremental, but they result in a powerful, efficient, and robust mechanism. This mechanism, when combined with our proposed policies, helps the VMM detect changes to the page tables and intelligently make a decision to switch modes and thus reduce overheads.

Our original paper has more details on the agile paging design to integrate page walk caches, perform guest context switches, set accessed/dirty bits, and handle small or short-lived processes. It also describes possible hardware optimizations.[2]

Methodology

To evaluate our proposal, we emulate our proposed hardware with Linux and prototype our software in Linux KVM.[5] We selected workloads with poor TLB performance from SPEC 2006,[6] BioBench,[7] Parsec,[8] and big-memory workloads.[9] We report overheads using a combination of hardware performance counters from native and virtualized application executions, along with

TLB performance emulation using a modified version of BadgerTrap[10] with a linear performance model. Our original paper has more details on our methodology, results, and analysis.[2]

Evaluation

Figure 5 shows the execution time overheads associated with page walks and VMM interventions with 4-Kbyte pages and 2-Mbyte pages (where possible). For each workload, four bars show results for base native paging (B), nested paging (N), shadow paging (S), and agile paging (A). Each bar is split into two segments. The bottom represents the overheads associated with page walks, and the top dashed segment represents the overheads associated with VMM interventions.

Agile paging outperforms its constituent techniques for all workloads and improves performance by 12 percent over the best of nested and shadow paging on average, and performs less than 4 percent slower than unvirtualized native at worst. In our original paper,[2] we show that more than 80 percent of TLB misses are covered under full shadow mode, thus having four memory accesses for TLB misses. Overall, the average number of memory accesses for a TLB miss comes down from 24 to between 4 and 5 for all workloads.

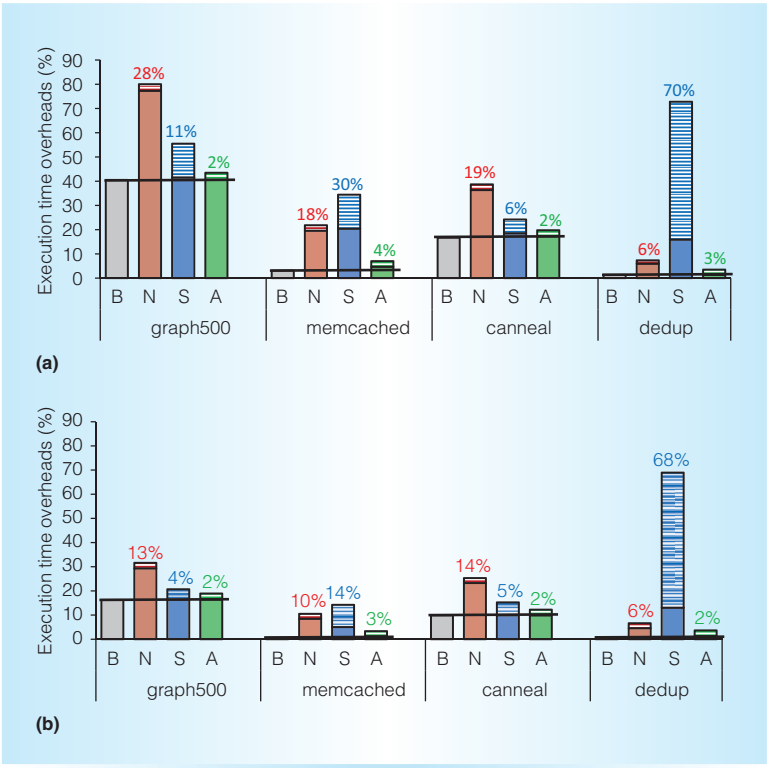

Figure 5. Execution time overheads for (a) 4-Kbyte pages and (b) 2-Mbyte pages (where possible) with base native (B), nested paging (N), shadow paging (S), and agile paging (A) for four representative workloads. All virtualized execution bars are in two parts: the bottom solid parts represent page walk overheads, and the top hashed parts represent VMM intervention overheads. The numbers on top of the bars represent the slowdown with respect to the base native case.

We and others have found that the overheads of virtualizing memory can be high. This is true in part because guest processes currently must choose between nesting paging with slow nested page table walks and shadow paging, in which page table updates cause costly VMM interventions. Ideally, one would want to use nested paging for addresses and page table levels that change and use shadow paging for addresses and page table levels that are relatively static.

Our proposal—agile paging—approaches this ideal. With agile paging, a virtualized address translation usually starts in shadow mode and then switches to nested mode only if required to avoid VMM interventions. Moreover, agile paging's benefits could be greater in the future, because Intel has recently added a fifth level to its page table[11] that makes a virtualized nested page walk up to 35 memory references, and emerging nonvolatile memory technology promises vast physical memories. MICRO

Acknowledgments

This work is supported in part by the US National Science Foundation (CCF-1218323, CNS-1302260, CCF-1438992, and CCF-1533885), Google, and the University of Wisconsin (John Morgridge chair and named professorship to Hill). Hill and Swift have significant financial interests in AMD and Microsoft, respectively.

References

1. J. Buell et al., "Methodology for Performance Analysis of VMware vSphere Under Tier-1 Applications," *VMware Technical J.*, vol. 2, no. 1, 2013, pp. 19–28.

2. J. Gandhi, M.D. Hill, and M.M. Swift, "Agile Paging: Exceeding the Best of Nested and Shadow Paging," *Proc. 43rd Int'l Symp. Computer Architecture*, 2016, pp. 707–718.
3. R. Bhargava et al., "Accelerating Two-Dimensional Page Walks for Virtualized Systems," in Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, 2008, pp. 26–35.
4. K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 2–13.
5. A. Kivity et al., "KVM: The Linux Virtual Machine Monitor," *Proc. Linux Symp.*, vol. 1, 2007, pp. 225–230.
6. J.L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006, pp. 1–17.
7. K. Albayraktaroglu et al., "BioBench: A Benchmark Suite of Bioinformatics Applications," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software*, 2005, pp. 2–9.
8. C. Bienia et al., "The Parsec Benchmark Suite: Characterization and Architectural Implications," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques*, 2008, pp. 72–81.
9. A. Basu et al., "Efficient Virtual Memory for Big Memory Servers," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 237–248.
10. J. Gandhi et al., "BadgerTrap: A Tool to Instrument x86-64 TLB Misses," *SIGARCH Computer Architecture News*, vol. 42, no. 2, 2014, pp. 20–23.
11. *5-Level Paging and 5-Level EPT*, white paper, Intel, Dec. 2016.

Jayneel Gandhi is a research scientist at VMware Research. His research interests include computer architecture, operating systems, memory system design, virtual memory, and virtualization. Gandhi has a PhD in computer sciences from the University of Wisconsin–Madison, where he completed the work for this article. He is a member of ACM. Contact him at gandhij@vmware.com.

Mark D. Hill is the John P. Morgridge Professor, Gene M. Amdahl Professor of Computer Sciences, and Computer Sciences Department Chair at the University of Wisconsin–Madison, where he also has a courtesy appointment in the Department of Electrical and Computer Engineering. His research interests include parallel computer system design, memory system design, and computer simulation. Hill has a PhD in computer science from the University of California, Berkeley. He is a fellow of IEEE and ACM. He serves as vice chair of the Computer Community Consortium. Contact him at markhill@cs.wisc.edu.

Michael M. Swift is an associate professor in the Computer Sciences Department at the University of Wisconsin–Madison. His research interests include operating system reliability, the interaction of architecture and operating systems, and device driver architecture. Swift has a PhD in computer science from the University of Washington. He is a member of ACM. Contact him at swift@cs.wisc.edu.