Preserving Virtual Memory by Mitigating the Address Translation Wall

Abhishek Bhattacharjee
Rutgers University

Modern computer systems at all scales, from datacenters to wearable and biomedical devices, depend on virtual memory. Virtual memory gives software developers the illusion that memory is always sufficient and linear, easing the task of programming. The hardware and OS manage the relationship between virtual and physical memory address spaces. Perhaps the biggest testament to virtual memory's success is that programmers do not even think about it when writing code today. And yet, consider what would happen in its absence. Without virtual memory, it would be practically impossible to program modern systems, which have a complex assortment of on- and off-package memory devices, hard disks, solid-state disks, and more. Programmers would have to rewrite their code for every change in memory capacity or configuration. We would not be able to run multiple applications concurrently on computer systems, because applications would be able to overwrite one another's memory. Malicious programs would be able to corrupt the memory of other programs. In short, virtual memory is fundamental to system programmability, code portability, memory protection, system security, and indeed the very success of computing.

The Challenges Facing Virtual Memory Today

Troublingly, virtual memory is under threat today. The main problem is this: the core virtual memory abstraction was conceived decades ago, and its basic components have remained largely unchanged since. In that time, hardware and software have changed dramatically. Massive mainframes made with discrete electronic components have evolved into systems integrating not just tens or hundreds of CPUs, but also exotic specialized hardware. Hardware accelerators for graphics, video and signal processing, face recognition, and deep learning are being rapidly developed. This fleet of emerging hardware targets new and sophisticated algorithms on vast sets of data, maintains them in big key-value stores, interacts with users using speech and gestures, and enables new paradigms like virtual and augmented reality. And yet, remarkably, we continue to use traditional virtual memory concepts in this drastically altered computing landscape.

Consequently, virtual memory has become a system performance bottleneck. Consider virtual memory performance for an application that analyzes a large graph. Graph processing often involves chasing pointers over terabytes of data in irregular and unpredictable ways, with poor memory access locality. Poor access locality is known to stress hardware caches, degrading system performance. But recent studies reveal a lesser-known but crucial insight—poor access locality hampers the performance of the key hardware component of virtual memory, the translation look-aside buffer (TLB) cache. TLBs are used to translate the virtual to physical addresses and often consume as much as 20 to 40 percent of the runtime on these sorts of workloads.[1–4]

Industry's Response

This performance crisis constitutes an "address translation wall," analogous to the notion of the memory wall that has plagued the computing industry for several decades. These performance problems have prompted a strong response from processor vendors, who are designing increasingly sophisticated virtual memory support today. Figure 1 compares the virtual memory components of a typical system we might use today with those found in systems just 10 to 15 years ago. Overall, the changes have been staggering.

For example, architects traditionally implemented a single TLB to cache frequently accessed entries in the page

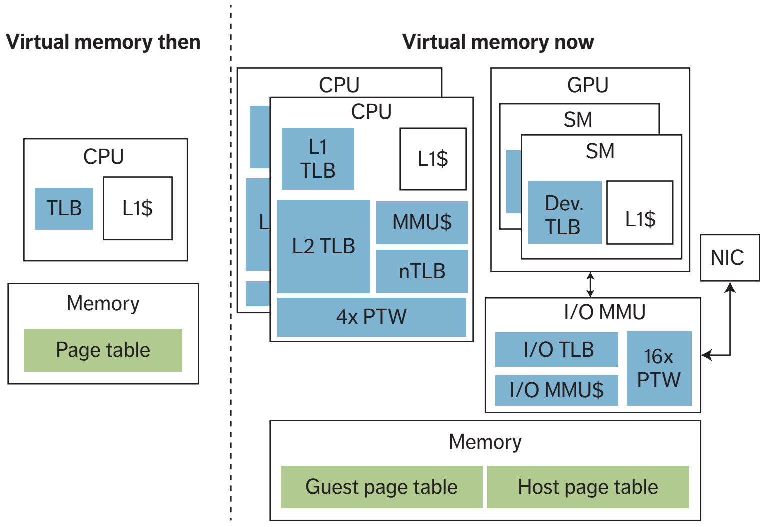

Figure 1. While traditional virtual memory implementations consisted of a single hardware translation look-aside buffer (TLB) and software page table, today's virtual memory stack comprises a complex assortment of multilevel TLBs, hardware page table walkers (PTWs), memory management unit (MMU) caches, nested TLBs, and multiple page tables for guest operating systems and hypervisors. Furthermore, this assortment of hardware is integrated not only in CPUs but also in hardware accelerators such as GPUs and network interface cards (NICs).

table maintained in main memory. Contrast this with the virtual memory support used today. Vendors are budgeting ever-increasing chip area for TLBs in a bid to improve capacity and reduce misses. Consequently, modern processors have extremely large and highly associative two-level TLBs per CPU—for example, Intel's Skylake chip uses 64-entry level-1 (L1) TLBs and 12-way, 1,536-entry level-2 (L2) TLBs. These structures require almost as much area as L1 caches today, and can consume as much as 10 to 15 percent of the chip energy.[5,6] Additionally, the fact that each CPU needs its own dedicated TLBs means that translation coherence, analogous to cache coherence, becomes a first-class performance bottleneck, too, often consuming more than 10 percent of the workload runtime.[7–9]

Worse still, despite these increases in TLB capacity, misses remain unavoidable. Put simply, modern workloads are often memory-intensive and have poor access locality. The problem is compounded by trends such as virtualization, which requires multiple page tables (see Figure 1) for guest operating systems and the hypervisor. Consequently, processor vendors also implement hardware to accelerate page table lookup. For example, CPUs traditionally responded to TLB misses by trapping to the OS and executing a lightweight OS routine to look up the page table. Today, these mode-switch overheads are deemed too expensive. Instead, CPUs are equipped with hardware page table walkers (PTWs), which can perform page table lookups without context-switching the application from the CPU. Furthermore, modern PTWs can often service multiple misses in parallel (for example, AMD's Ryzen and Skylake chips can service two to four TLB misses in parallel per CPU). PTWs, in turn, interface with new caching structures that store different portions of the radix-tree-based page tables. Consider, for example, MMU caches[10,11] and nested TLBs,[12] which cumulatively take up almost as much space as the L1 TLB in each CPU today.

Perhaps even more radical changes to virtual memory can be seen in the non-CPU components of modern systems. For example, Figure 1 shows that hardware accelerators such as GPUs and network interface cards (NICs) require address translation support, too.[13,14] This support takes the form of large TLBs with several thousands of entries in the devices themselves, as well as dedicated I/O MMUs, which maintain even larger TLBs, MMU caches, and heavily multithreaded hardware PTWs.[15]

The bottom line is that engineers are investing significant effort and resources in tackling the problems faced by virtual memory today. And yet, despite these efforts, the address translation wall remains a vexing problem with real-world consequences. An important recent example of this can be found in systems used for mining crypto-currencies such as Bitcoin and Ethereum. In particular, Ethereum miners find that their workloads face performance cliffs from inadequate TLB capacity on the GPUs they use. Consequently, TLB capacity is one of the first-order design parameters they consider in their choice of GPU architecture.[16–18]

Promising Research Approaches

The research community has not remained blind to these problems and has proposed several innovative solutions.

Hardware–Software Codesign

Beyond obviously important work on topics such as superpages (including recent work on Ingens[19] and noncontiguous superpages[20]), a particularly intriguing idea is that of direct segments, first proposed by Arkaprava Basu and colleagues.[1] This work goes back to virtual memory basics and asks what aspects of virtual memory are being used by modern workloads. It turns out that for an important and wide class of memory-intensive workloads, there is little paging activity or fine-grained memory protection usage. Furthermore, most memory accesses in these workloads are to large anonymous regions of allocated memory space. This is because these workloads often initialize memory at startup and are generally run on systems in which memory capacity is more than ample, as they are latency critical. These seemingly simple observations yield a powerful insight: If the OS can provide applications with a direct segment memory abstraction (essentially acting as a more massive and flexible version of a superpage), while retaining the paging abstraction for the remainder of the address space, TLB misses can be reduced dramatically. This brand of hardware–software codesign is an exciting direction and has been followed up with ideas such as range translations.[21]

Another promising direction is that studied by Hanna Alam and colleagues,[22] who ask a different question. Suppose there are situations in which it may be possible for application developers to posit virtual-to-physical page mappings amenable to fast address translation. In these cases, what kinds of mechanisms should the OS expose to the programmer to implement these mechanisms efficiently? In a sense, this idea adds to a rich body of work that decouples virtual-to-physical mappings from access permissions. Naturally, this idea begs the question, how successfully can programmers identify and use such mapping schemes? Studying such approaches, particularly in the context of separating memory protection from translation (most recently represented by the CHERI capability system[23]), may be fruitful for upcoming big-memory systems.

Hardware Approaches

While approaches that require hardware–software codesign are promising, purely hardware approaches are also valuable. For example, we have shown that real-world applications and OSes often (although they don't have to) allocate memory in a manner where tens of contiguous virtual pages are mapped to tens of physical pages.[2,24,25] This enables lightweight TLB coalescing, in which a single entry can store information about multiple contiguous mappings. Such hardware schemes are easy to implement and require no OS or software changes. Consequently, TLB coalescing schemes are being adopted by industry (for example, AMD's Ryzen chip supports TLB coalescing today). Furthermore, these types of approaches are equally applicable to caching structures beyond TLBs, such as MMU caches.[10] Looking ahead, recent work by Chang Hyun Park and colleagues has further expanded on this notion of coalescing.[26] We believe that it will be interesting to study whether there may be lightweight techniques at the OS level that can create more patterns amenable to these types of hardware coalescing efforts.

Equally intriguingly, several recent studies suggest that there is performance to be extracted from nontraditional TLB designs. Recent work by Jee Ho Ryoo and colleagues is an exciting example of this.[3] Conventional wisdom suggests that TLBs must be small to ensure fast access time. However, Ryoo's part-of-memory TLBs show that alternate designs may be possible for multilevel TLB hierarchies, in which it may be beneficial to back latency-critical L1/L2 TLBs with slower but considerably larger in-DRAM TLBs. This is an interesting direction worthy of further investigation, and it may be an especially promising approach to enabling address translation support for emerging near-memory processing accelerators, too.

Recent work also targets address-translation problems beyond capacity issues. For example, consider studies on mechanisms to accelerate TLB misses. This problem is particularly pertinent on GPUs, where TLB misses are unavoidably frequent. In particular, studies show the need for heavily multithreaded hardware PTWs for good GPU address translation performance.[13–15] Other approaches, targeting CPUs, leverage the notion of TLB speculation.[27,28] The basic idea with this approach is to speculate on the value of a physical page associated with a virtual page. If the translation is absent in the TLB, the page table walk is performed in the background to verify whether speculation was correct, whereas the CPU continues to execute independent instructions out of order. Past work suggests that there are low-overhead ways to perform this prediction with reasonably high accuracy, and we believe that future work may discover additional opportunities for speculation.

Looking Beyond TLB Hit Rates and Miss Penalties

Some recent work from our group also suggests that there may be traditionally overlooked aspects of address translation contributing to performance overheads. Specifically, we ask the question, when a memory access prompts a TLB miss, what is the overhead of its replay once the TLB miss is handled? Although it seems intuitive that page table walks that miss in the cache are almost always followed by cache misses for the replay, replays have not traditionally been optimized for better performance. Consequently, we have proposed techniques that trigger prefetches of replay data into caches when TLB misses occur, improving performance.[29]

Beyond capacity and miss penalties, translation coherence is fast becoming a major performance sink on modern systems. In particular, we are beginning to integrate memory devices with differing latency, bandwidth, and density characteristics on the same system, using them to realize heterogeneous memory architectures. To fully benefit from the complementary characteristics of these architectures, pages must be migrated among them. Consequently, recent studies show that translation coherence, which is currently an expensive operation implemented in software on most systems, consumes 10 to 30 percent of the application runtime. Many recent techniques in software have been proposed to solve this problem,[8,9] but it may also be time to consider mechanisms that enable hardware translation coherence.[7,30–32] A particularly intriguing approach to achieve this may be to overlay translation coherence atop existing cache-coherence protocols. After all, translation coherence operations are invoked when page tables are modified, which already invokes cache-coherence messages. Folding translation coherence atop cache coherence has several useful properties. For example, translation coherence scaling challenges could be addressed with techniques already used to achieve cache-coherence scaling. Moreover, one could verify both types of coherence jointly. Pioneering work on the UNITD protocol proposed by Bogdan Romanescu and colleagues shows how one might architect such joint coherence protocols.[30] Our recent work on HATRIC builds on UNITD's initial proposal, but we believe that this remains a rich area for further exploration.[7]

Correctness Issues

Finally, a word of caution: as we propose solutions to the address translation wall, we must also carefully address the design verification challenges that they will inevitably pose. This task is particularly crucial because the virtual memory hardware–software interface is notoriously prone to design bugs.[33,34] As systems integrate features like concurrent PTWs and TLB coalescing, performance may be improved, but system complexity and hence the scope of design bugs is worsened. While the research community has begun tackling the challenges of address translation verification, with seminal work by Romanescu and colleagues[34] and our follow-up studies,[33] much remains to be done.

Successfully preserving virtual memory will require rearchitecting the hardware–software interface so that these layers operate in tandem, rather than at odds with one another. Encouragingly, there is evidence that both chip vendors and OS designers are willing to innovate at this layer, as seen by a recent implementation of CPU TLB coalescing techniques and rapid changes in GPU address-translation hardware. But several important open problems persist, and new ones are presenting themselves rapidly. As just one example, a recent work by Javier Picorel and colleagues looks at the challenges posed by address translation on near-memory accelerators.[35] The bottom line is that these trends present both an opportunity and a challenge for researchers in computer systems. The evolving landscape of hardware and software means that virtual memory abstraction is in flux, but also that simple mechanisms to mitigate the address translation wall are likely to be useful to real-world systems and products.

References

1. A. Basu et al., "Efficient Virtual Memory for Big Memory Servers," *Proc. 40th Ann. Int'l Symp. Computer Architecture* (ISCA 13), 2013, pp. 237–248.
2. B. Pham et al., "CoLT: Coalesced Large-Reach TLBs," *Proc. 45th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO), 2012, doi:10.1109/MICRO.2012.32.
3. J.H. Ryoo et al., "Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB," *Proc. 44th Ann. Int'l Symp. Computer Architecture* (ISCA 17), 2017, pp. 469–480.
4. M.-M. Papadopoulou et al., "Prediction-Based Superpage-Friendly TLB Designs," *Proc. IEEE 21st Int'l Symp. High Performance Computer Architecture* (HPCA), 2015, doi:10.1109/HPCA.2015.7056034.
5. A. Basu, M. Hill, and M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," *Proc. 39th Ann. Int'l Symp. Computer Architecture* (ISCA 12), 2012, pp. 297–308.
6. V. Karakostas et al., "Energy-Efficient Address Translation," *Proc. Int'l Symp. High Performance Computer Architecture* (HPCA), 2016, doi:10.1109/HPCA.2016.7446100.
7. Z. Yan et al., "Hardware Translation Coherence for Virtualized Systems," *Proc. 44th Ann. Int'l Symp. Computer Architecture* (ISCA 17), 2017, pp. 430–443.
8. M. Oskin and G.H. Loh, "A Software-Managed Approach to Die-Stacked DRAM," *Proc. Int'l Conf. Parallel Architecture and Compilation* (PACT), 2015, pp. 188–200.
9. N. Amit, "Optimizing the TLB Shootdown Algorithm with Page Access Tracking," *Proc. USENIX Ann. Conf.*, 2017, pp. 27–39.
10. A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," *Proc. 46th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO), 2013, pp. 383–394.
11. T.W. Barr, A.L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," *Proc. 37th Ann. Int'l Symp. Computer Architecture,* 2010, pp. 48–59.
12. R. Bhargava et al., "Accelerating Two-Dimensional Page Walks for Virtualized Systems," *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2008, pp. 26–35.
13. B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," *Proc. 19th Int'l Conf. Architectural Support for Programming*

Languages and Operating Systems, 2014, pp. 743–758.

14. J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," *Proc. IEEE 20th Int'l Symp. High Performance Computer Architecture* (HPCA), 2014; doi:10.1109/HPCA.2014.6835965.
15. J. Vesely et al., "Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS), 2016, doi: 10.1109/ISPASS.2016.7482091.
16. "ETH Mining: Lower VRAM GPUs to be Rendered Unprofitable in Time," *Tech Power Up*, blog, 19 June 2017; www.techpowerup.com/234482/eth-mining-lower-vram-gpus-to-be-rendered-unprofitable-in-time.
17. "Ethereum Hashrate Drop for Radeon RX400/RX500 GPUs Is Incoming," *Crypto Mining Blog*, 18 June 2017; http://cryptomining-blog.com/8822-ethereum-hashrate-drop-for-radeon-rx400rx500-gpus-is-incoming.
18. I. King, "Chipmakers Nvidia, AMD Ride Cryptocurrency Wave—for Now," *SlashDot*, blog, 17 July 2017; www.bloomberg.com/news/articles/2017-07-17/chipmakers-nvidia-amd-ride-cryptocurrency-wave-for-now.
19. Y. Kwon et al., "Coordinated and Efficient Huge Page Management with Ingens," *Proc. 12th USENIX Conf. Operating Systems Design and Implementation* (OSDI 16), 2016, pp. 705–721.
20. Y. Du et al., "Supporting Superpages in Non-Contiguous Physical Memory," *Proc. IEEE 21st Int'l Symp. High Performance Computer Architecture* (HPCA), 2015, doi:10.1109/HPCA.2015.7056035.
21. V. Karakostas et al., "Redundant Memory Mappings for Fast Access to Large Memories," *Proc. 42nd Ann. Int'l Symp. Computer Architecture* (ISCA 15), 2015, pp. 66–78.
22. H. Alam et al., "Do It Yourself Virtual Memory Translation," *Proc. 44th Ann. Int'l Symp. Computer Architecture* (ISCA 17), 2017, pp. 457–468.
23. J. Woodruff et al., "The CHERI Capability Model: Revisiting RISC in an Age of Risk," *Proc. 41st Ann. Int'l Symp. Computer Architecture* (ISCA 14), 2014, pp. 457–468.
24. B. Pham et al., "Increasing TLB Reach by Exploiting Clustering in Page Translations," *Proc. IEEE 21st Int'l Symp. High Performance Computer Architecture* (HPCA), 2014, doi:10.1109/HPCA.2014.6835964.
25. G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," *Proc. 22nd Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2017, pp. 435–448.
26. C.H. Park et al., "Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocation," *Proc. 44th Ann. Int'l Symp. Computer Architecture* (ISCA 17), 2017, pp. 444–456.
27. B. Pham et al., "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?" *Proc. 48th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO), 2015, doi:10.1145/2830772.2830773.
28. T.W. Barr, A.L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," *Proc. 38th Ann. Int'l Symp. Computer Architecture* (ISCA 11), 2011, pp. 307–318.
29. A. Bhattacharjee, "Translation-Triggered Prefetching," *Proc. 22nd Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2017, pp. 63–76.
30. B.F. Romanescu et al., "Unified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All," *Proc. IEEE 21st Int'l Symp. High Performance Computer Architecture* (HPCA), 2010, doi:10.1109/HPCA.2010.5416643.
31. A. Awad et al., "Avoiding TLB Shootdown with Self-Invalidating TLB Entries," to be published in *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT), 2017.
32. C. Villavieja et al., "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT), 2011, doi:10.1109/PACT.2011.65.
33. D. Lustig et al., "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface," *Proc. 21st Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2016, pp. 233–247.
34. B.F. Romanescu, A.R. Lebeck, and D.J. Sorin, "Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency," *Proc. 15th Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), 2010, pp. 323–334.
35. J. Picorel, D. Jevdjic, and B. Falsafi, "Near-Memory Address Translation," to be published in *Proc. 26th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT), 2017.

Abhishek Bhattacharjee is an associate professor of computer science at Rutgers University and a CV Starr Fellow at the Neuroscience Institute at Princeton University. Contact him at abhib@cs.rutgers.edu.