Range Translations for Fast Virtual Memory

Jayneel Gandhi
University of Wisconsin—Madison
Vasileios Karakostas
Furkan Ayar
Adrián Cristal
Barcelona Supercomputing Center
Mark D. Hill
University of Wisconsin—Madison
Kathryn S. McKinley
Microsoft
Mario Nemirovsky
ICREA
Michael M. Swift
University of Wisconsin—Madison
Osman S. Ünsal
Barcelona Supercomputing Center

Modern workloads suffer high execution-time overhead due to page-based virtual memory. The authors introduce range translations that map arbitrary-sized virtual memory ranges to contiguous physical memory pages while retaining the flexibility of paging. A range translation reduces address translation to a range lookup that delivers near-zero virtual memory overhead.

Virtual memory is a crucial abstraction in modern computer systems. It delivers benefits such as security due to process isolation and improved programmer productivity due to simple linear addressing. Each process has a large a private virtual address space managed at granularity of fixed-size pages, which are typically 4 Kbytes in size. The operating system and hardware use a page table with a virtual-to-physical page map to simplify software and hardware memory management.

With virtual memory, the processor must translate every load and store generated by a process from a virtual to a physical address. Because address translation is on processors' critical path, a translation look-aside buffer (TLB) accelerates translation by caching the most recently used page table entries (PTEs). Paging delivers high performance when TLB hits service most of the address translations. However, a TLB miss triggers a costly hardware page table walk, which may require multiple memory accesses (up to four memory accesses in x86-64) to fetch the PTE.

Unfortunately, modern workloads experience execution time overheads of up to 50 percent because of paging.[1] Two opposing technology trends are at the root of this problem:

- Physical memory is growing exponentially cheaper and bigger (see Figure 1), which allows workloads to store ever-increasing large datasets in memory.
- TLB sizes have grown slowly because TLBs are on the processor's critical path to access memory (see Table 1).

This problem is commonly called *limited TLB reach*—the fraction of physical memory that TLBs can map reduces with each hardware generation. For instance, the TLB in Intel's recent Skylake processors covers only 9 percent of a 256-Gbyte memory. We expect this mismatch between TLB reach and memory size to become worse with newer memory technologies, which promise petabytes to zetabytes of physical memory, and to increase the paging overheads because of the time required by page walks.

As the sidebar, "Efforts to Address Limited Translation Look-Aside Buffer Reach," explains, previous approaches to address this

Published by the IEEE Computer Society

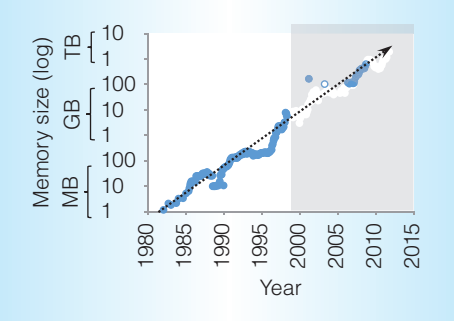

Figure 1. Physical memory sizes purchased with $10,000 for the past 35 years show exponential growth. The cost is inflation-adjusted to the 2011 US dollar. Data is collected from jcmit.com.

Table 1. Translation look-aside buffer (TLB) sizes in Intel processors for past 15 years are growing slowly.

Year	Processor	L1 TLB size	L2 TLB size
1999	Pentium III	72	0
2004	Pentium 4	64	0
2008	Nehalem	96	512
2012	Ivy Bridge	100	512
2014	Haswell	100	1,024
2015	Skylake	100	1,552

problem include hierarchical TLBs (adding larger but slower L2 TLBs), multipage mappings (mapping several pages with a single TLB entry), huge pages (mapping much larger aligned memory with a single TLB entry), and direct segments (providing a single arbitrarily large segment along with standard paging). None of these approaches delivers a complete solution that solves the TLB reach problem while retaining flexible memory use.

Our goal, originally set forth in our paper for the 42nd International Symposium on Computer Architecture,[2] is a transparent and robust virtual memory implementation that has no alignment restrictions and near-zero overhead across workloads while retaining the flexibility of paging.

Many applications present an unexploited opportunity to reduce address translation overhead: they naturally exhibit an abundance of contiguity in their virtual address space. Figure 2 plots the number of pages and contiguous virtual page ranges required to map all of an application's address space for seven representative workloads. All of the workloads require less than 112 ranges to map their entire virtual address space. If the OS can map a range contiguously, a single entry is sufficient to translate from a virtual to a physical range. Hence, a modest number of ranges have the potential to efficiently perform address translation for the majority of virtual memory addresses—orders of magnitude less than with regular or even huge PTEs. This article proposes a hardware/

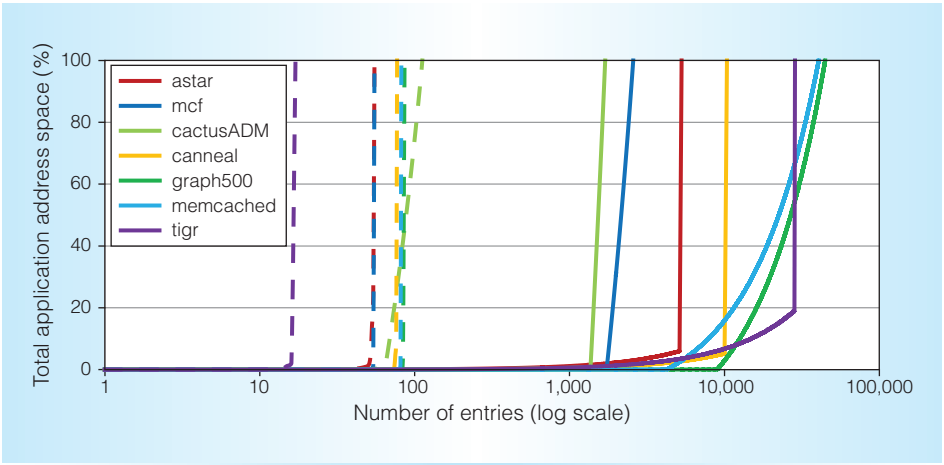

Figure 2. Cumulative distribution function of the application's memory (percentage) that *N* translation entries map with pages (solid) and with optimal ranges (dashed), for seven representative applications. Ranges map all applications' memory with one to four orders of magnitude fewer entries than pages.

Efforts to Address Limited Translation Look-Aside Buffer Reach

Researchers have proposed and used several prior approaches to reduce paging overheads. Table A shows the benefits and drawbacks of each proposal and compares it to our proposal of Redundant Memory Mappings (RMM).

Hierarchical TLBs

Hierarchical TLBs increase TLB reach by increasing TLB size. Each TLB entry still maps one page (see Figure A1), but a larger and slower L2 TLB caches page table entries to reduce expensive page walks. The combined (L1 + L2) TLB reach is increased, but it has not kept pace with the growth of physical memory.

Multipage Mappings

Multipage mappings exploit contiguity in groups of virtual and physical pages by mapping a small number of pages (typically 8 to 16) with a single TLB entry (see Figure A2). These approaches leverage the default OS memory allocator that creates either small blocks of contiguous physical pages to contiguous virtual pages (sub-blocked TLBs[1] and CoLT[2]) or a small set of contiguous virtual pages to a cluster of physical pages (Clustered TLB[3]). These approaches increase TLB reach by a small fixed multiple. Because multipage mappings impose size-alignment restrictions, they require effort by the OS to exploit, and they do not increase TLB reach enough to meet the needs of applications that use modern gigabyte-to-terabyte physical memories.

Huge Pages

Huge pages map a larger aligned fixed-size region of memory with a single TLB entry (see Figure A3). For instance, the x86-64 architecture has 2-Mbyte and 1-Gbyte pages.[4,5] Huge pages increase the TLB reach substantially, but their effectiveness is reduced by the size alignment restriction: the OS can allocate them only when the available physical memory is both size-aligned and contiguous. Moreover, many current processors provide limited TLB entries for huge pages, which further reduces their benefits on modern workloads.

Direct Segments

Direct segments is a hardware/software approach that maps a single unlimited range of contiguous virtual memory to contiguous physical

Table A. Comparison of Redundant Memory Mappings (RMM) with previous approaches for reducing virtual memory overhead. RMM achieves the best of many worlds.

Requirements	Hierarchical TLBs	Multipage mappings	Huge pages	Direct segments	RMM
Flexible alignment	✓	✗	✗	✓	✓
Arbitrary reach	✗	✗	✗	✓	✓
Multiple entries	✓	✓	✓	✗	✓
Transparent to applications	✓	✓	✓	✗	✓
Applicable to all workloads	✓	✓	✓	✗	✓

software co-design called Redundant Memory Mappings (RMM) that realizes the potential of ranges to improve virtual memory performance.

Design Overview

We introduce the key concept of range translation that exploits the virtual memory contiguity in modern workloads to perform address translation much more efficiently than paging. Inspired by direct segments, a range translation is a mapping from contiguous virtual pages to contiguous physical pages of arbitrary size with uniform protection bits. A range translation uses BASE and LIMIT virtual addresses. To translate a virtual address to a physical address, the hardware adds the virtual address to the physical OFFSET of the corresponding range. Range translations are base-page aligned and have no other size or size-alignment restrictions.

We implement range translations in the RMM architecture. RMM employs hardware/software co-design to map the entire virtual address space with standard paging and redundantly map ranges with range translations. Because range translations are backed by page mappings in RMM, the operating system can flexibly choose between using range translations or not, which retains the benefits of paging for fine-grained memory management when necessary. Figure 3 shows how a few range translations map parts of the process's address space in addition to pages in RMM. This design addresses the limitations and

memory with a single hardware entry, while the rest of the virtual address space uses standard paging.[6] Direct segment entry consists of BASE, LIMIT, and OFFSET registers that eliminate page walks within the segment (see Figure A4). The OS maps a virtual address to a direct segment or page, but never both.

Although direct segments provide the foundation for our work, they are not general or transparent. They map only a single segment and require developers to explicitly allocate the direct segment during startup. Although some "big memory" applications can preallocate a single large range, many cannot. Many applications instead tend to allocate several large ranges (see Figure 2 in the main article). These limitations caused the industry to push back against direct segments.

References

1. M. Talluri and M.D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," *Proc. 6th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 171–182.
2. B. Pham et al., "CoLT: Coalesced Large Reach TLBs," *Proc. 45th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2012, pp. 258–269.
3. B. Pham et al., "Increasing TLB Reach by Exploiting Clustering in Page Translations," *Proc. IEEE 20th Int'l Symp. High Performance Computer Architecture*, 2014, pp. 558–567.
4. J. Corbet, "Transparent Huge Pages in 2.6.38," 2011; www.lwn.net/Articles/423584.
5. M. Gorman, "Huge Pages Part 1 (Introduction)," 2010; http://lwn.net/Articles/374424.
6. Basu et al., "Efficient Virtual Memory for Big Memory Servers," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 237–248.

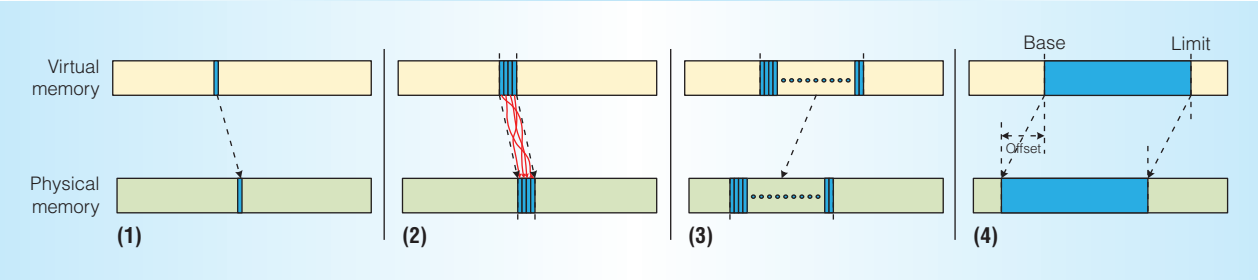

Figure A. Memory mapped by one entry with various proposals: (1) hierarchical translation look-aside buffers, (2) multipage mappings, (3) huge pages, and (4) direct segments. Each proposal tries to increase the reach of the TLB.

combines the advantages of previous approaches (see Table A in the sidebar).

The RMM system efficiently caches range translations in a hardware range TLB to increase TLB reach, manages range translations using a per-process software range table just like the page table, and increases physical contiguity to increase the range size, which results in a modest number of range translations per process using eager paging. Table 2 summarizes these new components and their relationship to paging.

Compared to prior approaches, RMM delivers multiple arbitrarily large regions of memory with range translations, improves performance transparently without programmer intervention, and enhances robustness because the OS manages memory with both ranges and pages. On a range of workloads, RMM reduces the cost of virtual memory to less than 1 percent on average.

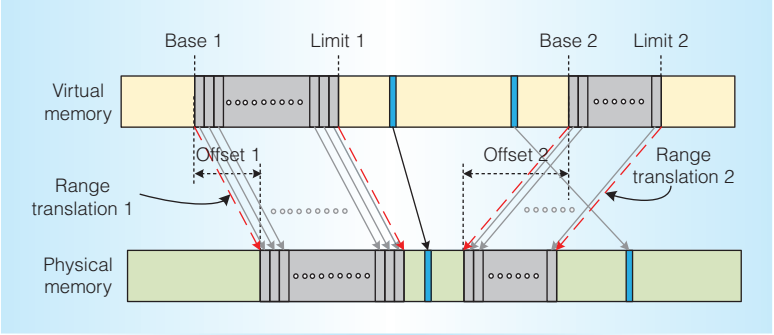

Figure 3. RMM design. The application's memory space is represented redundantly by both pages and range translations.

Range TLB

The range TLB is a hardware cache that holds multiple range translations. Each entry can perform address translation for an unlimited range of contiguous virtual pages that are

Table 2. RMM overview with page and range translation.

Design components	Page translation (x86-64)	+ Range translation
Architecture	TLB	Range TLB
	Page table	Range table
	CR3 register	CR-RT register
	Page table walker	Range table walker
OS	Page table management	Range table management
	Demand paging	Eager paging

mapped to contiguous physical pages with uniform protection bits. Each range TLB entry consists of a virtual range and translation. The virtual range stores the $BASE_i$ and $LIMIT_i$ of the virtual address range. The translation stores the $OFFSET_i$ that holds the start of the range in physical memory minus $BASE_i$, and the protection bits.

We design a fully associative range TLB. The right side of Figure 4 illustrates the range TLB and its logic with N (for example, 32) entries. The range TLB is accessed in parallel with the last-level page TLB (for example, the L2 TLB, shown in Figure 4). The hardware compares the virtual page number that misses in the L1 TLB, testing $BASE_i \leq$ virtual page number $< LIMIT_i$ for all ranges in parallel in the range TLB. On a hit, the range TLB returns the $OFFSET_i$ and protection bits for the corresponding range translation and calculates the corresponding PTE for the L1 TLB. It adds the requested virtual page number to the hit $OFFSET_i$ value to produce the physical page number and copies the protection bits from the range translation. On a miss, the hardware fetches the corresponding range translation—if it exists—from the range table. Our original paper contains more details and optimizations on the hardware and OS design.[2]

Range Table

The range table is an architecturally visible per-process data structure that stores the process's range translations in memory and is redundant to the page table. The operating system manages range table entries.

A range table implementation should facilitate fast lookup of a virtual address to a range translation and should be inherently compact and cache friendly. To this end, we propose a B-Tree data structure with ($BASE_i$, $LIMIT_i$) as keys and $OFFSET_i$ and protection bits as values to store range translations in the range table. Figure 5 shows how the range translations are stored in the range table and also shows each node's design. Each node accommodates four range translations and points to five children, allowing up to 124 range translations in three levels. Each range table node fits in two cache lines. All pointers use physical addresses and facilitate hardware walking. With this design, a single 4-Kbyte page can hold a range table with 128 range translations.

A hardware walker loads range translations from the range table on a range TLB miss. Analogous to the page table pointer register (CR3 in x86-64), RMM requires a CR-RT register to point to the physical address of the range table root for walking.

On a miss to the range TLB and page TLB, RMM first fetches the missing translation from the page table and installs it in the higher-level TLB so that the processor can continue executing the pending operation. To identify whether a miss in the range TLB can be resolved to a range, RMM adds a range bit to the PTE, which indicates whether a page is part of a range table entry. The page table walker fetches the PTE, and if the range bit is set, accesses the range table in the background and updates the range TLB with the missing entry. This approach avoids increasing page-walk latency and does not access the range table for pages that are not redundantly mapped.

Eager Paging

Effective range translation requires both virtual contiguity, which occurs naturally, and

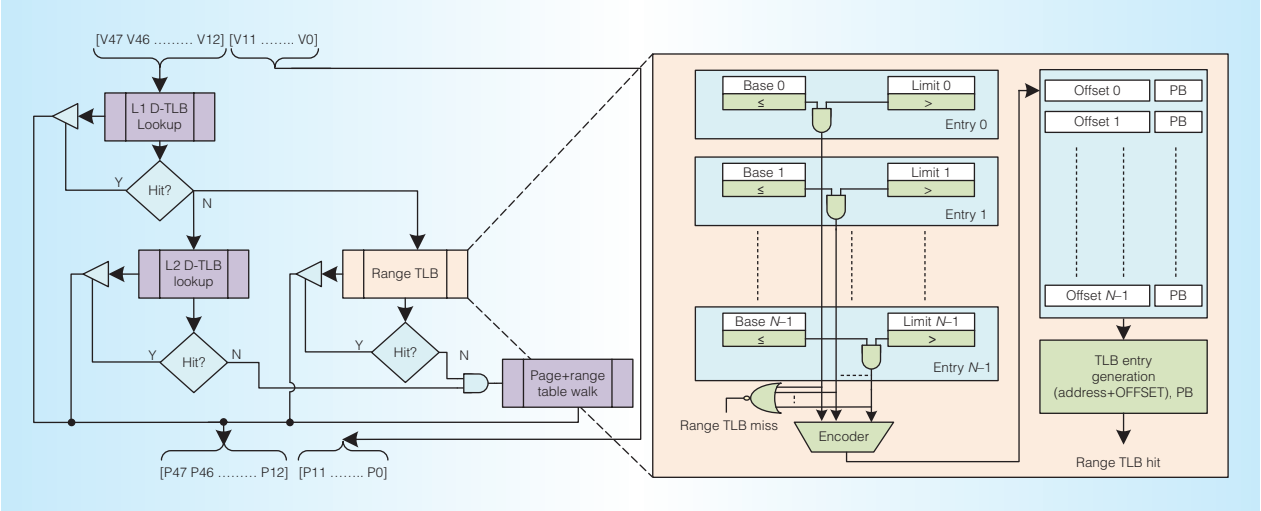

Figure 4. Range TLB caches range translations and is accessed in parallel with the last-level page TLB. The left side shows a range TLB introduced in parallel to the L2 TLB, and the right side shows the structure and lookup operation in a range TLB.

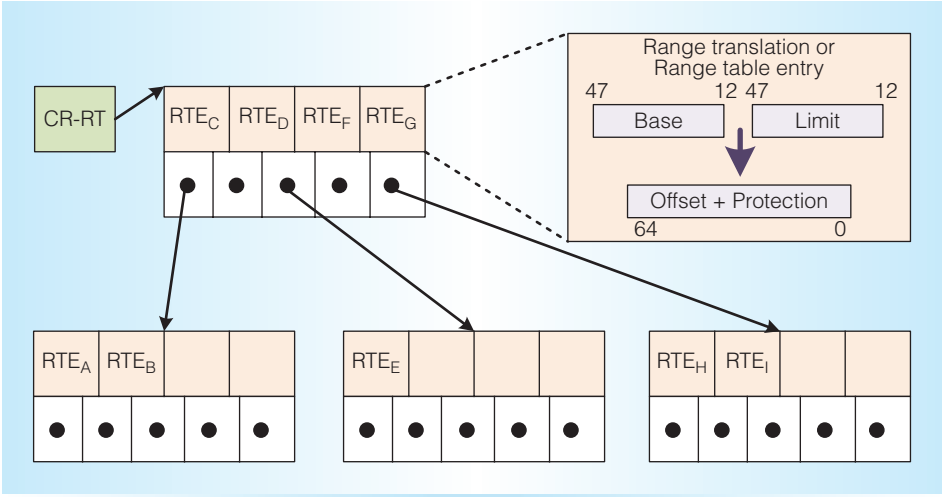

Figure 5. The range table stores the range translations for a process in memory. The OS manages the range table entries based on the applications' memory management operations.

physical contiguity, which may not. To enhance physical contiguity, RMM modifies the OS memory-allocation mechanism with eager paging.

The default allocation policy—demand paging—allocates physical pages at access time and may degrade contiguity, because it allocates single pages even when large regions of physical memory are available. For example, if the application accesses contiguous pages out-of-order, the OS may not allocate these pages in contiguous physical pages, even though there are contiguous free pages.

Eager paging generates large range translations by allocating consecutive physical pages to consecutive virtual pages eagerly at allocation time, rather than lazily on demand. When the application allocates memory, the OS establishes one or more range translations for the entire request and updates the corresponding range and PTEs. Figure 6 shows the simplified pseudocode for eager paging

```
Compute the memory fragmentation;
if memory fragmentation ≤ threshold then
       // use eager paging
       while number of pages > 0 do
         for (i = MAX_ORDER-1; i ≥ 0; i--) do
           if freelist[i] ≥ 0 and 2^i ≤ number of pages then
             allocate block of 2^i pages;
             for all 2^i pages of the allocated block do
               construct and set the PTE;
             end
             add the block to the range table;
             number of pages – = 2^i;
             break;
           end
         end
       end
else
       // high memory fragmentation – use demand paging
       for (i = 0; i < number of pages; i++) do
         allocate the PTE;
         set the PTE as invalid so that the first access
will trigger a page fault and the page will get allocated;
       end
end
```

Figure 6. RMM memory allocator pseudocode for an allocation request of a number of pages. When memory fragmentation is low, RMM uses eager paging to allocate pages at request time, in order to create the largest possible range for the allocation request. Otherwise, RMM uses default demand paging to allocate pages at access time.

based on Linux's buddy page allocator. The OS always updates the page table and the range table consistently. Eager paging increases latency during allocation and could induce fragmentation, because the OS must instantiate all pages in memory, even though the application never uses the allocated physical pages. However, the OS can reclaim unused pages at the end of a range or an entire range if memory pressure increases.

Methodology

We implemented our OS modifications in the Linux kernel v3.15.5 and defined RMM hardware with respect to a recent Intel x86-64 Sandy Bridge dual-socket Xeon E5-2430 core (L1 TLB entries: 64 for a 4-Kbyte page, 32 for a 2-Mbyte page, 4 for a 1-Gbyte page; L2 TLB entries: 512 for a 4-Kbyte page). We chose a 32-entry fully associative range TLB accessed in parallel with the L2 page TLB because we estimated that it could meet the L2's timing constraints. We selected workloads with poor TLB performance from SPEC 2006,[3] BioBench,[4] Parsec,[5] and big-memory workloads.[1] We report overheads using a combination of hardware performance counters from native application executions and TLB performance emulation using a modified version of BadgerTrap[6] with a linear performance model. Compared to cycle-accurate simulation, we reduced weeks of simulation time by orders of magnitude. Our original paper has more details on our methodology, results, and analysis.[2]

Evaluation

Figure 7 compares the overhead spent in page walks for RMM to other techniques. The 4-Kbyte and 2-Mbyte transparent huge pages (THP, the native Linux approach of using 2-Mbyte pages when possible)[7] and 1-Gbyte[8] configurations show the measured overhead for the three available page sizes. All other configurations are emulated. The DS bars show direct segments[1] results and the RMM bars show the 32-entry range TLB results.

Our results show that RMM performs well on all configurations for all workloads, substantially improving over other approaches. RMM eliminates the vast majority of page walks, significantly outperforms huge pages (THP and 1-Gbyte), and achieves similar or better performance than direct segments, but has none of its limitations. Overall, RMM has negligible overhead of less than 1 percent—it essentially eliminates virtual memory overheads for many workloads. Our original paper also analyzes energy, hardware costs, and the impact of eager paging on execution time and the memory footprint.[2]

In a subsequent paper, which appeared in the 22nd International Symposium on High Performance Computer Architecture,[9] we characterized and then reduced the energy of address translation. We showed that L1 TLB hits consume the majority of address translation energy. For instance, Sandy Bridge performs 12 address comparisons on every memory reference hit. We introduced the Lite mechanism that reduces energy by dynamically downsizing the L1 TLBs when huge pages or range translations reduce pressure on them.

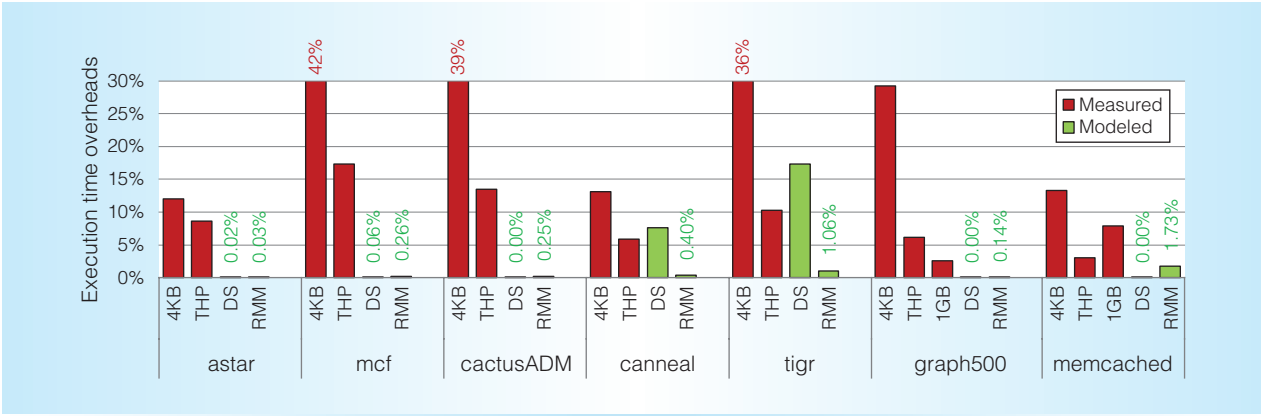

Figure 7. Execution time overheads due to page walks for seven representative workloads. The 1-Gbyte page size applies only to big-memory workloads.

Limited TLB reach is a well-known problem. To address this problem, vendors have increased hardware support for huge pages and slowly increased TLB sizes. However, fixed-size pages will always fall short because of their alignment requirements and because they induce internal fragmentation. As memory sizes continue to increase more aggressively than TLB sizes, the virtual memory overheads that manifest in today's systems with 4-Kbyte pages will manifest similarly in tomorrow's systems with huge pages. Our evaluation shows that such cases already exist. Furthermore, range translations should pave the way for emerging workloads, such as in-memory computing, which leverage the growth in physical memory to store huge datasets for low latency and real-time data analysis.

In conclusion, we believe RMM has the potential to follow the same path as Madhusudhan Talluri and Mark Hill's work,[10] which bootstrapped research on transparent huge pages. It also required changes to both hardware and operating systems but is now common in modern processors. MICRO

Acknowledgments

Jayneel Gandhi and Vasileios Karakostas were the lead authors of this article. This work is supported in part by the European Union (FEDER funds) under contract TIN2012-34557, the European Union's Seventh Framework Programme (FP7/2007- 2013) under the ParaDIME project (GA no. 318693), the National Science Foundation (CCF-1218323, CNS-1302260, and CCF-1438992), Google, and the University of Wisconsin (Kellett Award and named professorship to Mark D. Hill). Karakostas is also supported by an FPU research grant from the Spanish MEC. Hill has a significant financial interest in AMD.

References

1. A. Basu et al., "Efficient Virtual Memory for Big Memory Servers," *Proc. 40th Ann. Int'l Symp. Computer Architecture*, 2013, pp. 237–248.
2. V. Karakostas et al., "Redundant Memory Mappings for Fast Access to Large Memories," *Proc. 42nd Ann. Int'l Symp. Computer Architecture*, 2015, pp. 66–78.
3. J.L. Henning, "SPEC CPU2006 Benchmark Descriptions," *Computer Architecture News*, vol. 34, no. 4, 2006, pp. 1–17.
4. 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.
5. 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.
6. 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.

7. J. Corbet, "Transparent Huge Pages in 2.6.38," 2011; www.lwn.net/Articles/423584.
8. M. Gorman, "Huge Pages Part 1 (Introduction)," 2010; http://lwn.net/Articles/374424.
9. V. Karakostas et al., "Energy-Efficient Address Translation," *Proc. 22nd Ann. Symp. High Performance Computer Architecture*, 2016, pp. 631–643.
10. M. Talluri and M.D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," *Proc. 6th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 171–182.

Jayneel Gandhi is a PhD student in the Computer Sciences Department at University of Wisconsin–Madison. He received an MS in computer engineering from North Carolina State University and an MS in computer sciences from the University of Wisconsin–Madison. He is a student member of the ACM. Contact him at jayneel@cs.wisc.edu.

Vasileios Karakostas is a PhD student in the Computer Architecture Department at Universitat Politecnica de Catalunya and a researcher in the Computer Architecture for Parallel Paradigms group at Barcelona Supercomputing Center. He received an MS in computer architecture, networks, and systems from the Universitat Politecnica de Catalunya. He is a student member of ACM and IEEE. Contact him at vasilis.karakostas@bsc.es.

Furkan Ayar is an MS student in the Computer Engineering Department at Yildiz Technical University. He received a BS in computer engineering from Dumlupinar University. He performed the work for this article while an intern at the Barcelona Supercomputing Center. Contact him at frkn.ayar@gmail.com.

Adrián Cristal is a scientific researcher at the Spanish National Research Council (CISC-IIIA) and a comanager of the Computer Architecture for Parallel Paradigms research group at the Barcelona Supercomputing Center. He received a PhD in computer science from the Polytechnic University of Catalonia. Contact him at adrian.cristal@bsc.es.

Mark D. Hill is the John P. Morgridge Professor, the Gene M. Amdahl Professor of Computer Sciences, and the 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. He received a PhD in computer science from the University of California, Berkeley. He is a Fellow of IEEE and ACM. Contact him at markhill@cs.wisc.edu.

Kathryn S. McKinley is a principal researcher at Microsoft. She received a PhD in computer science from Rice University. She is a Fellow of IEEE and ACM. Contact her at mckinley@microsoft.com.

Mario Nemirovsky is a Catalan Institution for Research and Advanced Studies (ICREA) Senior Research Professor at the Barcelona Supercomputing Center. He received a PhD in electrical and computer engineering from the University of California, Santa Barbara. Contact him at mario.nemirovsky@bsc.es.

Michael M. Swift is an associate professor in the Computer Sciences Department at the University of Wisconsin–Madison. He received a PhD in computer science from the University of Washington. He is a member of ACM. Contact him at swift@cs.wisc.edu.

Osman S. Ünsal is a comanager of the Computer Architecture for Parallel Paradigms research group at the Barcelona Supercomputing Center. He received a PhD in electrical and computer engineering from the University of Massachusetts, Amherst. He is a member of IEEE and ACM. Contact him at osman.unsal@bsc.es.