Range Translations for Fast Virtual Memory

Jayneel Gandhi[1]* Vasileios Karakostas[2,3]* Furkan Ayar[4] Adrián Cristal[2,3,5] Mark D. Hill[1]
Kathryn S. McKinley[6] Mario Nemirovsky[7] Michael M. Swift[1] Osman S. Ünsal[3]

[1]University of Wisconsin-Madison [2]Universitat Politecnica de Catalunya [3]Barcelona Supercomputing Center
[4]Yildiz Technical University [5]Spanish National Research Council (IIIA-CISC) [6]Microsoft Research
[7]ICREA Senior Research Professor at Barcelona Supercomputing Center

***Abstract*— Modern workloads suffer high execution-time overhead due to page-based virtual memory. We 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.**

Keywords—Virtual Memory; Memory Management; Translation Lookaside Buffer

INTRODUCTION

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 very large private virtual address space managed at granularity of fixed size pages, typically 4 KB in size. The operating system (OS) and hardware use a page table with a one-to-one virtual-to-physical page map to simplify software and hardware memory management.

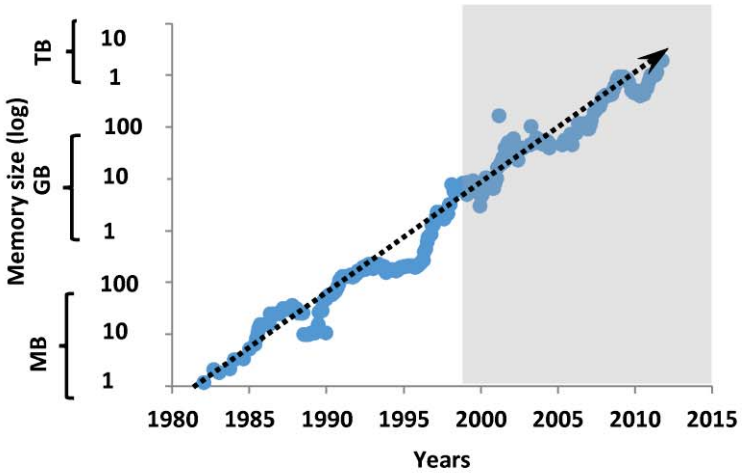
With virtual memory, the processor must translate every load and store generated by a process from a virtual to physical address. Because address translation is on processors' critical path, a Translation Lookaside Buffer (TLB) accelerates translation by caching the most recently used Page Table Entries (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 4 memory accesses in x86-64) to fetch the PTE.

Growing Overheads of Paging

Unfortunately, modern workloads are experiencing execution time overheads of up to 50% due to paging [2]. The following two opposing technology trends are at the root of this problem:

1. Physical memory is growing exponentially cheaper and bigger (Figure 1(a)) allowing modern workloads to store ever increasing large data sets in memory.
2. TLB sizes have grown slowly, because TLBs are on the processor's critical path to access memory (Figure 1(b)).

This problem is commonly called *limited TLB reach*—the fraction of physical memory that TLBs can map is reducing with each hardware generation. For instance, the TLB in Intel's recent Skylake processors covers only 9% of a 256 GB memory. We expect this mismatch between TLB reach and memory size (i) to keep growing, (ii) to become worse with newer memory technologies, which promise petabytes to zetabytes of physical memory, and (iii) to increase the overheads of paging due to the time required by page walks.



(a)

Year	Processor	L1 TLB size	L2 TLB size
1999	Pentium III	72	0
2004	Pentium 4	64	0
2008	Nehalem	96	512
2012	Ivybridge	100	512
2014	Haswell	100	1024
2015	Skylake	100	1552

(b)

Figure 1 (a) Physical memory sizes purchased with $10,000 for the last 35 years show exponential growth. (b) TLB sizes in Intel processors for last 15 years are growing slowly.

* Both authors contribute equally to this work.

Digital Object Indentifier 10.1109/MM.2016.10 0272-1732/$26.00 2016 IEEE

BACKGROUND: EFFORTS TO ADDRESS LIMITED TLB REACH

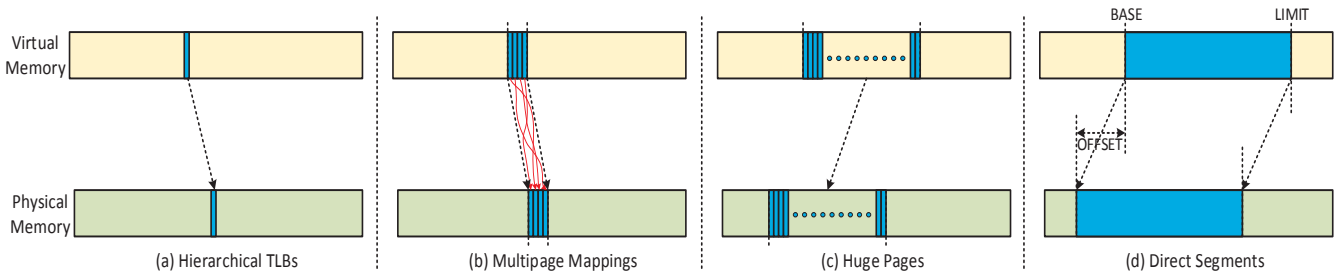


Figure 2 Memory mapped by one entry with various proposals.

Several prior approaches have been proposed and used to reduce paging overheads.

Hierarchical TLBs

Hierarchical TLBs increase TLB reach in response to stagnating L1 TLB sizes. Each TLB entry still maps one page (Figure 2(a)), but a larger and slower L2 TLB caches PTEs to reduce expensive page walks. The combined (L1 + L2) TLB reach increases, but 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 8-16 pages) with a single TLB entry (Figure 2(b)). These approaches leverage the default OS memory allocator that creates either (i) small blocks of contiguous physical pages to contiguous virtual pages (sub-blocked TLBs [12] and CoLT [11]), or (ii) a small set of contiguous virtual pages to a cluster of physical pages (Clustered TLB [10]). 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 much larger aligned fixed size region of memory with a single TLB entry (Figure 2(c)). For instance, the x86-64 architecture has 4 KB, 2 MB, and 1 GB pages [4,6]. Huge pages increase the TLB reach substantially, but their effectiveness is reduced by the size alignment restriction: the OS can only allocate them 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 are a hardware/software approach that map a single unlimited range of contiguous virtual memory to contiguous physical memory with a single hardware entry, while the rest of the virtual address space uses standard paging [2]. Direct segment entry consists of BASE, LIMIT, and OFFSET registers that eliminate page walks within the segment (Figure 2(d)). 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 nor transparent. They only map a single segment and require developers to explicitly allocate the direct segment during startup. While some 'big memory' applications can preallocate a single large range, many cannot. Many applications instead tend to allocate several large ranges (Figure 3). Since direct segments are not backed by pages, dynamically disabling them is not practical. Due to these limitations, direct segments received push-back from industry.

Table 1 Comparison of RMM with previous approaches for reducing virtual memory overhead. RMM achieves best of many worlds.

	Hierarchical TLBs	Multipage Mappings	Large Page	Direct Segments	RMM
Flexible alignment	✓	✗	✗	✓	✓
Arbitrary reach	✗	✗	✗	✓	✓
Multiple entries	✓	✓	✓	✗	✓
Transparent to applications	✓	✓	✓	✗	✓
Applicable to all workloads	✓	✓	✓	✗	✓

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

Goal

The goal of this work, originally appearing in the 42nd International Symposium on Computer Architecture (ISCA'15) [8] is a transparent and robust virtual memory implementation that has fast address translation and no alignment restrictions with near zero overhead across a variety of workloads while retaining the flexibility of paging.

Opportunity

Many applications exhibit an abundance of contiguity in their virtual address space. Figure 3 plots the number of pages and the number of contiguous virtual page *ranges* required to map all of an application's address space for 7 representative workloads. All the workloads require less than 112 ranges to map their entire virtual address space. If the OS can map this virtual contiguity to physical contiguity, a single entry is sufficient to translate from a virtual range to a physical range.

Digital Object Indentifier 10.1109/MM.2016.10 0272-1732/$26.00 2016 IEEE

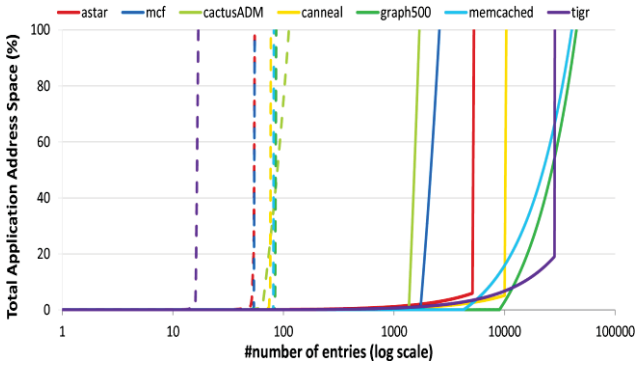


Figure 3 Cumulative distribution function of the application's memory (%) that N translation entries map with pages (solid) and with optimal ranges (dashed), for 7 representative applications. Ranges map all applications' memory with one to four orders of magnitude fewer entries than pages.

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 page table entries. This paper proposes a hardware/software co-design called *Redundant Memory Mappings* 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 between contiguous virtual pages mapped to contiguous physical pages of arbitrary size with uniform protection bits. A range translation uses BASE and LIMIT virtual addresses. To translate a virtual range address to 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 Redundant Memory Mappings (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. Since range translations are backed by page mappings in RMM, the operating system can flexibly choose between using range translations or not, retaining the benefits of paging for fine-grain memory management when necessary. Figure 4 shows how a few range translations map parts of the process's address space in addition to pages in RMM. This

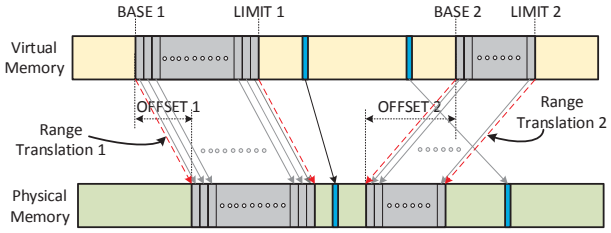


Figure 4 Redundant Memory Mappings design. The application's memory space is represented redundantly by both pages and range translations.

Table 2 Overview of Redundant Memory Mappings.

	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

design addresses the limitations and combines the advantages of previous approaches (see Table 1).

The RMM system (i) efficiently caches range translations in a hardware ***range TLB*** to increase TLB reach, (ii) manages range translations using a per-process software ***range table*** just like the page table, and (iii) increases physical contiguity to increase the range size resulting in 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 since 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% on average.

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 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 (PB).

We design a fully associative range TLB. The right side of Figure 5 illustrates the range TLB and its logic with N (e.g., 32) entries. The range TLB is accessed in parallel with the last-level page TLB (e.g., the L2 TLB as shown in Figure 5). 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 page table entry 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 (introduced next). The original paper contains more details and optimizations on the hardware and OS design [8].

Range Table

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

Digital Object Indentifier 10.1109/MM.2016.10 0272-1732/$26.00 2016 IEEE

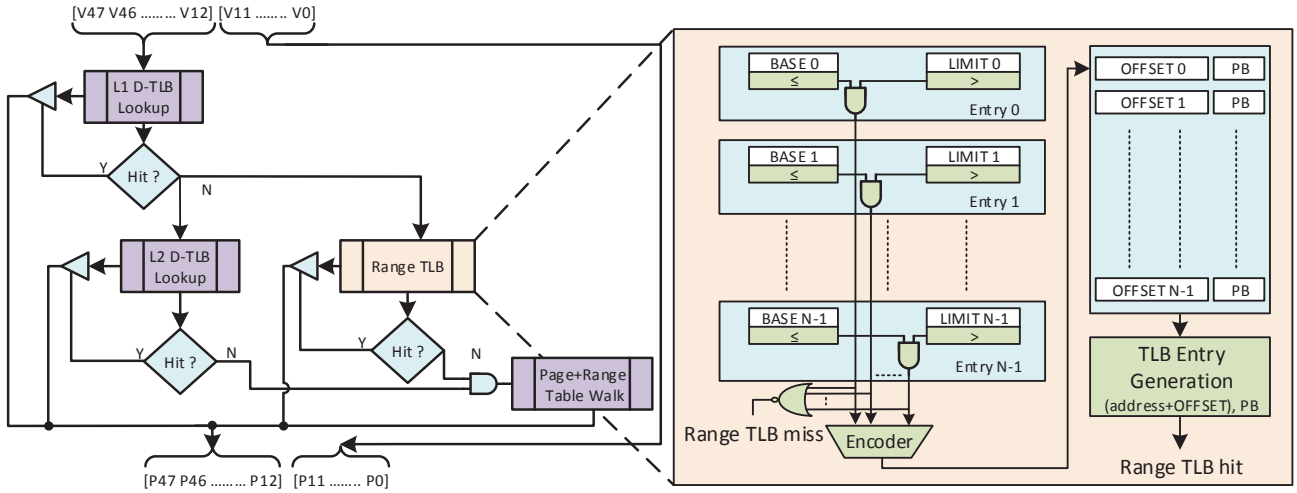


Figure 5 Range TLB caches range translations and is accessed in parallel with the last-level page TLB.

A range table implementation should facilitate fast lookup of a virtual address to a range translation, be inherently compact and be cache friendly. To this end, we propose 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 6 shows how the range translations are stored in the range table and the design of each node. Each node accommodates four range translations and points to five children, e.g., up to 124 range translations in three levels. Hence, each range table node fits in two cachelines. All pointers use physical addresses and facilitate hardware walking. With this design, a range table on a single 4 KB page can hold 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.

Handling Range TLB misses

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 or not, 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 range table entry. This approach prevents the increase in the latency of page walks, and skips accesses in the range table for pages that are not redundantly mapped.

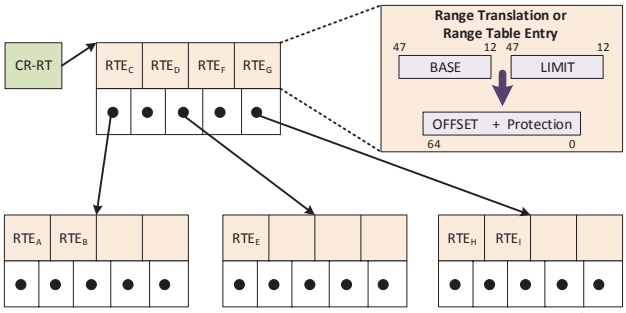


Figure 6 The range table stores the range translations for a process in memory. The OS manages the range table entries based on the applications memory management

EAGER PAGING

Effective range translation requires both virtual contiguity, which occurs naturally, and physical contiguity, which may not.

```
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 7 RMM memory allocator pseudocode for an allocation request of number of pages. When memory fragmentation is low, RMM uses eager paging to allocate pages at request-time, creating the largest possible range for the allocation request. Otherwise, RMM uses default demand paging to allocate pages at access-time.

Digital Object Indentifier 10.1109/MM.2016.10 0272-1732/$26.00 2016 IEEE

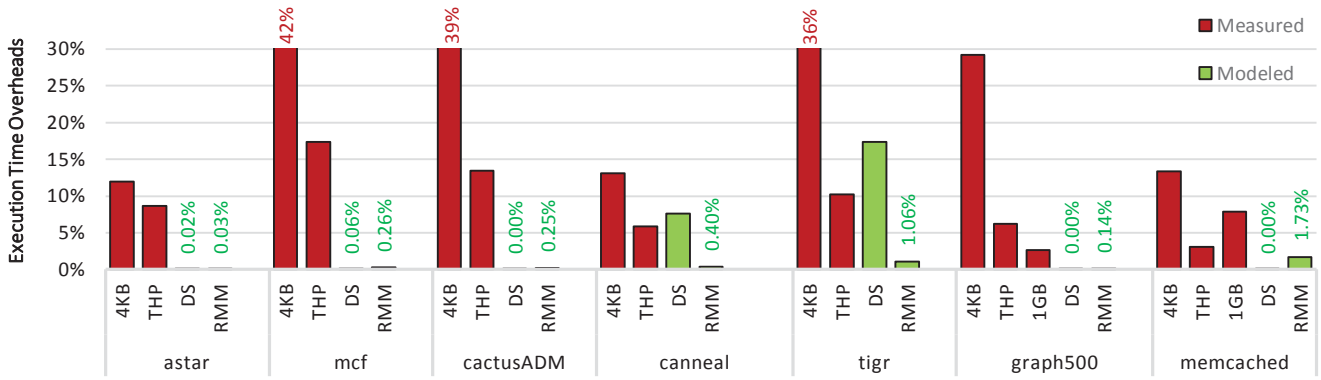


Figure 8 Execution time overheads due to page walks for 7 representative workloads. 1GB page is only applicable to big-memory workloads.

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 degrades contiguity, because (i) it allocates single pages even when large regions of physical memory are available, and because (ii) the OS may assign pages accessed out-of-order to non-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 page table entries. Figure 7 shows the simplified pseudocode for eager paging based on Linux's buddy page allocator. The OS always updates both the page table and the range table to consistently manage the entire memory. Eager paging increases latency during allocation and may induce fragmentation, because the OS must instantiate all pages in memory, even though the application never uses. However, the OS may reclaim unused pages at the end of a range or an entire range if memory pressure increases.

METHODOLOGY

We select workloads with poor TLB performance from SPEC 2006 [7], BioBench [1], Parsec [3] and big-memory workloads [2]. We implement our OS modifications in the Linux kernel v3.15.5 and define RMM hardware with respect to a recent Intel x86-64 Sandy Bridge Dual socket Xeon E5-2430 core (L1 TLB entries: 64 for 4KB page, 32 for 2MB page, 4 for 1GB page; L2 TLB entries: 512 for 4KB page). We choose a 32-entry fully associative range TLB accessed in parallel with the L2 page TLB, since we estimate that it can meet the L2's timing constraints. We report overheads using a combination of hardware performance counters from native application executions and TLB performance emulation using a modified version of BadgerTrap [5] with a linear performance model. Compared to cycle-accurate simulation, we reduce weeks of simulation time by orders of magnitude. The original paper has more details on methodology, results, and analysis [8].

EVALUATION

Figure 8 compares the overhead spent in page walks for RMM to other techniques. The 4 KB, 2 MB Transparent Huge Pages (THP) [4] and 1 GB [6] configurations show the measured overhead for the three available page sizes. All other configurations are emulated. The DS bars show direct segments [2] results and the RMM bars show the 32-entry range TLB results.

The 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 1GB), and achieves similar or better performance than direct segments, but has none of its limitations. Overall, redundant memory mappings achieve negligible overhead—essentially eliminating virtual memory overheads for many workloads to less than 1%. The original paper [8] also analyzes energy, hardware costs, and the impact of eager paging on execution time and memory footprint.

In a subsequent work at HPCA 2016 [9], we characterize and then reduce the energy of address translation. We show that L1 TLB *hits* consume the majority of address translation energy. For instance, Sandy Bridge performs 12 address comparisons on every memory reference hit. The key is to reduce energy by dynamically downsizing the L1 TLBs when huge pages or range translations reduce pressure on them.

CONCLUSION

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, we believe that this approach falls short. As memory sizes continue to increase more aggressively than TLB sizes, the virtual memory overheads that manifest in today's systems with 4KB pages will manifest similarly in tomorrow's systems with huge pages. Our evaluation shows that such cases already exist. Furthermore, range translations have the potential to pave the way for emerging workloads, such as in-memory computing, which leverage the growth in physical memory to store huge data sets for low latency and real time data analysis.

In conclusion, we believe RMM has the potential to follow the same path as Talluri and Hill's work [12], which bootstrapped research on transparent huge pages. It also required changes to both hardware and operating systems, but is now common in modern processors.

Digital Object Indentifier 10.1109/MM.2016.10 0272-1732/$26.00 2016 IEEE

REFERENCES

[1] Albayraktaroglu, K., Jaleel, A., Xue Wu, Franklin, M., Jacob, B., Chau-Wen Tseng, and Yeung, D. BioBench: A Benchmark Suite of Bioinformatics Applications. IEEE (2005), 2–9.

[2] Basu, A., Gandhi, J., Chang, J., Hill, M.D., and Swift, M.M. Efficient Virtual Memory for Big Memory Servers. *Proceedings of the 40th Annual International Symposium on Computer Architecture*, IEEE Computer Society (2013).

[3] Bienia, C., Kumar, S., Singh, J.P., and Li, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, (2008).

[4] Corbet, J. Transparent huge pages. 2011. www.lwn.net/Articles/423584/.

[5] Gandhi, J., Basu, A., Hill, M.D., and Swift, M.M. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Comput. Archit. News 42*, 2 (2014), 20–23.

[6] Gorman, M. Huge Pages/libhugetlbfs. 2010. http://lwn.net/Articles/374424/.

[7] Henning, J.L. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News 34*, 4 (2006), 1–17.

[8] Karakostas, V., Gandhi, J., Ayar, F., Cristal, A., Hill, M.D., McKinley, K.S., Nemirovsky, M., Swift, M.M., and Ünsal, O. Redundant memory mappings for fast access to large memories. *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, (2015), 66–78.

[9] Karakostas, V., Gandhi, J., Cristal, A., Hill, M.D., McKinley, K.S., Nemirovsky, M., Swift, M.M., and Ünsal, O. Energy-Efficient Address Translation. *In Proceedings of the 22nd Annual Symposium on High Performance Computer Architecture (HPCA '16)*, (2016).

[10] Pham, B., Bhattacharjee, A., Eckert, Y., and Loh, G.H. Increasing TLB reach by exploiting clustering in page translations. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, (2014), 558–567.

[11] Pham, B., Vaidyanathan, V., Jaleel, A., and Bhattacharjee, A. CoLT: Coalesced Large Reach TLBs. *Proceedings of 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM (2012).

[12] Talluri, M. and Hill, M.D. Surpassing the TLB performance of superpages with less operating system support. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, (1994).

Jayneel Gandhi is a PhD student in the Computer Sciences Department at University of Wisconsin-Madison. His research interests include computer architecture, operating systems, memory system design, virtual memory and virtualization. He has MS in both computer engineering and computer sciences from North Carolina State University and University of Wisconsin-Madison respectively. He is a student member of the ACM and is affiliated to SIGARCH and SIGMICRO. Contact him at jayneel@cs.wisc.edu.

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

Furkan Ayar is an MS student in Computer Engineering at Yildiz Technical University. His research interests include cyber security and operating systems. Ayar has a BS in computer engineering from Dumlupinar University. This work was performed while Ayar was on internship at Barcelona Supercomputing Center.

Adrián Cristal is Scientific Researcher at Spanish National Research Council (CSIC) and currently co-manager of the Computer Architecture for Parallel Paradigms research group at BSC. His interests include high-performance microarchitecture, multi- and many-core chip multiprocessors, transactional memory, and programming models. He received a PhD from the Computer Architecture Department at the Polytechnic University of Catalonia (UPC), Spain, and he has a BS and an MS in computer science from the University of Buenos Aires, Argentina.

Mark D. Hill is 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 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 the ACM. He serves in the leadership of the Computer Community Consortium.

Kathryn S. McKinley is a Principal Researcher at Microsoft. Her research interests include computer architecture, programming language implementation, and interactive web services. McKinley has a PhD in computer science from Rice University. She is a Fellow of the IEEE and the ACM and serves on the Boards of CRA and CRA-W.

Mario Nemirovsky is a Catalan Institution for Research and Advanced Studies (ICREA) Senior Research Professor at the Barcelona Supercomputing Center. His research interests include computer architectures, high performance computing, the Internet of Things, and emerging on-chip interconnect technologies. Nemirovsky has a PhD from the University of California, Santa Barbara.

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. He has a PhD in computer science from University of Washington. He is a member of the ACM.

Osman S. Ünsal is co-manager of Computer Architecture for Parallel Paradigms research group at Barcelona Supercomputing Center. His research interests include computer architecture, reliability and low-power computing. Ünsal has a PhD in electrical and computer engineering from University of Massachusetts, Amherst. He is a member of IEEE and ACM.

Direct questions and comments about this article to Jayneel Gandhi, 1210 W. Dayton Street, Madison WI 53706; jayneel@cs.wisc.edu.

Digital Object Indentifier 10.1109/MM.2016.10 0272-1732/$26.00 2016 IEEE