Redundant Memory Mappings for Fast Access to Large Memories

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

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

{vasilis.karakostas, adrian.cristal, mario.nemirovsky, osman.unsal}@bsc.es, frkn.ayar@gmail.com
{jayneel, markhill, swift}@cs.wisc.edu, mckinley@microsoft.com

Abstract

Page-based virtual memory improves programmer productivity, security, and memory utilization, but incurs performance overheads due to costly page table walks after TLB misses. This overhead can reach 50% for modern workloads that access increasingly vast memory with stagnating TLB sizes.

To reduce the overhead of virtual memory, this paper proposes Redundant Memory Mappings (RMM)*, which leverage ranges of pages and provides an efficient, alternative representation of many virtual-to-physical mappings. We define a* range *be a subset of process's pages that are virtually and physically contiguous. RMM translates each range with a single* range table entry*, enabling a modest number of entries to translate most of the process's address space. RMM operates in parallel with standard paging and uses a software range table and hardware range TLB with arbitrarily large reach. We modify the operating system to automatically detect ranges and to increase their likelihood with* eager page allocation*. RMM is thus transparent to applications.*

We prototype RMM software in Linux and emulate the hardware. RMM performs substantially better than paging alone and huge pages, and improves a wider variety of workloads than direct segments (one range per program), reducing the overhead of virtual memory to less than 1% on average.

1. Introduction

Virtual memory provides the illusion of a private and very large address space to each process. Its benefits include improved security due to process isolation and improved programmer productivity, since the operating system and hardware manage the mapping from per-process virtual addresses to physical addresses. Page-based implementations of virtual memory are ubiquitous in modern hardware. They divide physical memory into fixed-size pages, use a page table to map virtual pages to physical pages, and accelerate address lookups using Translation Lookaside Buffers (TLBs). When paging was introduced, it also delivered high performance, since TLBs serviced the vast majority of address translations.

Unfortunately, the performance of paging is suffering due to stagnant TLB sizes, whereas modern memory capacities continue to grow. Because TLB address translation is on the processors' critical path, it requires low access times which constrain TLB size and thus the number of pages that experience this access time. On a TLB miss, the system must walk the page table, which may incur additional cache misses. This problem is called limited *TLB reach*. Recent studies show that modern workloads can experience execution-time overheads of up to 50% due to page table walks [10, 12, 31]. This overhead is likely to grow, because physical memory sizes are still growing. Furthermore, many modern applications have an insatiable desire for memory—they increase their data set sizes to consume all available memory for each new generation of hardware [10, 21].

Previous research has focused on solving this problem by improving the efficiency of paging in the following three ways.

1. Multipage mappings use one TLB entry to map multiple pages (e.g., 8-16 pages per entry) [38, 39, 47]. Mapping multiple pages per entry increases TLB reach by a small fixed amount, but has alignment restrictions, and still leaves TLB reach far below modern gigabyte-to-terabyte physical memory sizes.
2. Huge pages map much larger fixed size regions of memory, on the orders of 2 MB to 1 GB on x86-64 architectures. Use of huge pages (THP [6] and libhugetlbfs [1]) increase TLB reach substantially, but also suffer from size and alignment restrictions and still have limited reach.
3. Direct segments provide a single arbitrarily large segment and standard paging for the remaining virtual address space [10, 23]. For applications that can allocate and use a single segment for the majority of their memory accesses, direct segments eliminate most of the paging cost. However, direct segments only support a single segment and require that application writers explicitly allocate a segment during startup.

ISCA'15, June 13-17, 2015, Portland, OR, USA
Copyright 2015 ACM. ISBN 978-1-4503-3402-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2749471

* Both authors contributed equally to this work.

	Transparent to application	Kernel support	Hardware support	# of entries	Maximum reach per entry	Application domain	No size-alignment restrictions
Multipage Mappings [47, 39, 38]	✓	✗	✓	512	32 KB to 16 MB	any	✗
Transparent Huge Pages [6, 36]	✓	✓	✓	32	2 MB	any	✗
libhugetlbfs [1]	✗	✓	✓	4	1 GB	big memory	✗
Direct segments [10]	✗	✓	✓	1	unlimited	big memory	✓
Redundant Memory Mappings	✓	✓	✓	N	unlimited	any	✓

Table 1: Comparison of Redundant Memory Mappings with previous approaches for reducing virtual memory overhead.

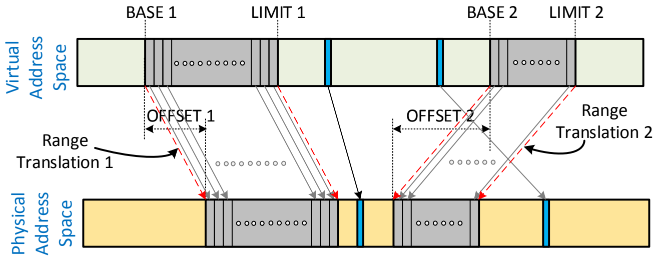

Figure 1: Range translation: an efficient representation of contiguous virtual pages mapped to contiguous physical pages.

The goal of our work is to provide a robust virtual memory mechanism that is transparent to applications and improves translation performance across a variety of workloads.

We introduce Redundant Memory Mappings (RMM) a novel hardware/software co-designed implementation of virtual memory. RMM adds a redundant mapping, in addition to page tables, that provides a more efficient representation of translation information for ranges of pages that are both physically and virtually contiguous. RMM exploits the natural contiguity in address space and keeps the complete page table as a fall-back mechanism.

RMM relies on the concept of *range translation*. Each range translation maps a contiguous virtual address range to contiguous physical pages, and uses BASE, LIMIT, and OFFSET values to perform translation of an arbitrary sized range. Range translations are only base-page-aligned and redundant to paging; the page table still maps the entire virtual address space. Figure 1 illustrates an application with two ranges mapped redundantly with paging as well as range translations.

Analogous to paging, we add a software managed *range table* to map virtual ranges to physical ranges and a hardware *range TLB* in parallel with the last-level page TLB to accelerate their address translation. Because range tables are redundant to page tables, RMM offers all the flexibility of paging and the operating system may use or revert solely to paging when necessary.

To increase contiguity in range translations, we extend the OS's default lazy demand page allocation strategy to perform eager paging. Eager paging instantiates pages in physical memory at allocation request time, rather than at first-access time as with demand paging. The resulting OS automatically maps most of process's virtual address space with orders of magnitude fewer ranges than paging with Transparent Huge Pages [6]. On a wide variety of workloads consuming between 350 MB – 75 GB of memory, we find that RMM has the potential to map more than 99% of memory for all workloads with 50 or fewer range translations (see Section 3's Table 2).

To evaluate this design, we implement RMM software support in Linux kernel v3.15.5. We emulate the hardware using a combination of hardware performance counters from an x86 execution and functional TLB simulation in BadgerTrap [22]—the same methodology as in prior TLB studies [10, 12, 23]. We compare RMM to standard paging, Clustered TLBs, huge (2 MB and 1 GB) pages, and direct segments (one range per program). RMM robustly performs substantially better than the former three alternatives on various workloads, and almost as fast as Direct segments when one range is applicable. However with RMM, more applications enjoy reductions in translation overhead without programmer intervention. Overall, RMM reduces the overhead of virtual memory to less than 1% on average.

In summary, the main contributions of this paper are:

- We show that diverse workloads exhibit an abundance of contiguity in their virtual address space.
- We propose Redundant Memory Mappings, a hardware/software co-design, which includes a fast and redundant translation mechanism for ranges of contiguous virtual pages mapped to contiguous physical pages, and operating system modifications that detect and manage ranges.
- We prototype RMM in Linux and evaluate it on a broad range of workloads. Our results show that a modest number of ranges map most of memory. Consequently, the range TLB achieves extremely high hit rates, eliminating the vast majority of costly page-walks compared to virtual memory systems that use paging alone.

2. Background

This section and Table 1 overview the closely related approaches to reducing paging overheads and compare them to RMM. Section 9 discusses related work more generally.

Multipage Mapping approaches, such as sub-blocked TLBs [47], CoLT [39] and Clustered TLBs [38], pack multiple Page Table Entries (PTEs) into a single TLB entry. These designs leverage *default* OS memory allocators that either (i) assign small blocks of contiguous physical pages to contiguous virtual pages (Sub-blocked TLBs and CoLT), or (ii) map small set of contiguous virtual pages to clustered sets of physical pages (Clustered TLB). However, they pack only a small

	Huge pages		Ideal RMM ranges		
Benchmark	4 KB	+ 2 MB	total	99% coverage	largest
astar	5129	+ 158	55	7	76.2%
mcf	1737	+ 839	55	1	99.0%
omnetpp	2041	+ 77	54	12	60.2%
cactusADM	1365	+ 333	112	49	2.4%
GemsFDTD	3117	+ 414	73	6	71.7%
soplex	4221	+ 411	61	5	41.9%
canneal	10016	+ 359	77	4	90.9%
streamcluster	1679	+ 55	78	14	83.8%
mummer	29571	+ 172	17	4	57.5%
tigr	28299	+ 235	16	3	97.9%
Graph500	8983	+ 35725	86	3	50.4%
Memcached	4243	+ 36356	82	2	98.6%
NPB:CG	2540	+ 26058	84	5	28.8%
GUPS	2210	+ 32803	92	1	99.7%

Table 2: Total translation entries mapping the application's memory with: (i) Transparent Huge Pages of 4 KB and 2 MB pages [6] and (ii) ideal RMM ranges of contiguous virtual pages to contiguous physical pages. (iii) Number of ranges that map 99% of the application's memory, and (iv) percentage of application memory mapped by the single largest range.

multiple of translations (e.g., 8-16) per entry, which limits their potential to reduce page-walks for large working sets.

Huge Pages using Transparent Huge Pages (THP) [6] and libhugetlbfs [1] increase the TLB reach by mapping very large regions with a single entry. The x86-64 architecture supports mixing 4 KB with 2 MB and 1 GB pages, while other architectures support more sizes [35, 41, 44]. The effectiveness of huge pages is limited by the size-alignment requirement: huge pages must have size-aligned physical addresses, and thus the OS can only allocate them when the available memory is size-aligned and contiguous [38, 39]. In addition, many commodity processors provide limited numbers of large page TLB entries, which further limits their benefit [10, 23, 31].

Direct segments [10] are a hardware/software approach that map a *single unlimited* range of contiguous virtual memory to contiguous physical memory using a single hardware segment, while the rest of the virtual address space uses standard paging. A virtual address is mapped by a direct segment or paging, but never both. Direct segments introduce BASE, LIMIT, and OFFSET registers to eliminate the page-walks within the segment. However, the mechanism requires that (i) applications explicitly allocate a direct segment during startup, and (ii) the OS can reserve a single large contiguous range of physical memory for a segment. Thus, direct segments are only suitable for big-memory workloads and require application changes.

Table 1 summarizes the characteristics of these approaches and compares them to RMM. RMM is completely transparent to applications and maps multiple ranges with no size-alignment restrictions, where each range contains an unrestricted amount of memory.

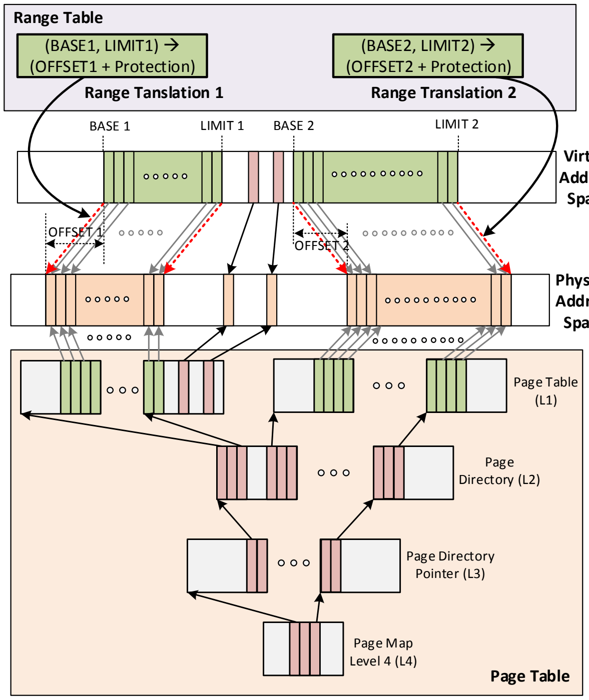

Figure 2: Redundant Memory Mappings design. The application's memory space is represented *redundantly* by both pages and range translations.

3. Redundant Memory Mappings

We observe that many applications naturally exhibit an abundance of contiguity in their virtual address space and the number of ranges needed to represent this contiguity is low.

Abundance of address contiguity. We quantify address contiguity by executing applications on x86-64 hardware (see Section 7 for workload and methodology details), and periodically scan the page table, measuring the size of virtual address ranges where all pages are mapped with the same permissions. Table 2 shows the minimum number of ranges of contiguous virtual pages that the OS could map to contiguous physical pages. The workloads require between 16 to 112 ranges to map their entire virtual address space. However, the number of ranges to cover 99% of the application's memory space falls to fewer than 50. Although a single range maps 90% or more of the virtual memory for 5 of the 14 workloads, the rest require multiple ranges. These results suggest that a small number of range translations have the potential to efficiently perform address translation for the majority of virtual memory addresses.

3.1. Overview

The above measurements motivate the RMM approach. (i) The OS uses best-effort allocation to detect and map contiguous virtual pages to contiguous physical pages in a range table in addition to mapping with the page table. (ii) The hardware range TLB caches multiple range translations providing an alternate translation mechanism, parallel to paging. (iii)

	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

Table 3: Overview of Redundant Memory Mapping

Most addresses fall in ranges and hit in the range TLB, but if needed, the system can revert to the flexibility and reduced fragmentation benefits of paging.

Definition: A *range translation* is a mapping between contiguous virtual pages mapped to contiguous physical pages with uniform protection bits (e.g., read/write). A range translation is of unlimited size and base-page-aligned. A range translation is identified by BASE and LIMIT addresses. To translate a virtual range address to physical address, the hardware adds virtual address to the OFFSET of the corresponding range. Figure 2 shows how RMM maps parts of the process's address space with both range translations and pages.

Redundant Memory Mappings (RMM) use *range translations* to perform address translation much more efficiently than paging for large regions of contiguous physical addresses. We introduce three novel components to manage ranges: (i) *range TLB*s, (ii) *range tables*, and (iii) *eager paging* allocation. Table 3 summarizes these new components and their relationship to paging. The *range TLB* hardware stores range translations and is accessed in parallel to the last-level page TLB (e.g., L2 TLB). The address translation hardware accesses the range and page TLBs in parallel after a miss at the previous-level TLB (e.g., L1 TLB). If the request hits in the range TLB or in the page TLB, the hardware installs a 4 KB TLB entry in the previous-level TLB, and execution continues. In the uncommon case that a request misses in both range TLB and page TLB, and the address maps to a range translation, the hardware fetches the page table entry to resume execution and optionally fetches a range table entry in the background.

RMM performance depends on the range TLB achieving a high hit ratio with few entries. To maximize the size of each range, RMM extends the OS page allocator to improve contiguity with an *eager paging* mechanism that instantiates a contiguous range of physical pages at allocation time, rather than the on-demand default, which instantiates pages in physical memory upon first access. The OS always updates both the page table and the range table to consistently manage the entire memory at both the page and range granularity.

4. Architectural Support

The RMM hardware primarily consists of the range TLB, which holds multiple range translations, each of which translates for an unlimited-size range. Below, we describe RMM as an extension to the x86-64 architecture, but the design applies to other architectures as well.

4.1. Range TLB

The range TLB is a hardware cache that holds multiple range translations. Each entry maps an unlimited range of contiguous virtual pages to contiguous physical pages. The range TLB is accessed in parallel with the last-level page TLB (e.g., the L2 TLB) and in case of hit, it generates the corresponding 4 KB entry in the previous-level page TLB (e.g., the L1 TLB).

We design the range TLB as a fully associative structure, because each range can be any size making standard indexing for set-associative structure hard. The right side of Figure 3 illustrates the range TLB and its logic with N (e.g., 32) entries. 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 map. The translation stores the OFFSET_i that holds the start of the range in physical memory minus BASE_i, and the protection bits (PB). Additionally, each range TLB entry includes two comparators for lookup operations.

Figure 3 illustrates accessing the range TLB in parallel with the L2 TLB, after a miss at the L1 TLB. The hardware compares the *virtual page number* that misses in the L1 TLB, testing $\text{BASE}_i \leq \text{virtual page number} < \text{LIMIT}_i$ 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. We explain this operation in Section 4.3 after discussing the range table in more detail.

The range TLB is accessed in parallel with the last-level page TLB and must return the lookup result (hit/miss) within the TLB access latency, which for the L2 TLB on recent Intel processors is ~7 cycles [28]. Unlike a page TLB, the range TLB is similar to N fully-associative copies of direct segment's base/limit/offset logic [10] or a simplified version of the range cache [48]: it performs two comparisons per entry instead of a single equality test. Our design can achieve this performance because the range TLB contains only a few entries and it can use fast comparison circuits [32]. Our results in Section 8 show that a 32-entry fully-associative range TLB eliminates more than 99% of the page-walks for most of our applications, at lower power and area cost than simply increasing the size of the corresponding L2 TLB. Note that our approach of accessing the range TLB in parallel to the last-level page TLB can be extended to the other translation levels closer to the processor (e.g., in parallel to the L1 TLB); we leave such analysis for future work.

Optimization. To reduce the dynamic energy cost of the fully associative lookups, we introduce an optional *MRU Pointer* that stores the most-recently-used range translation and thus reduces associative searches of the range TLB. The range TLB

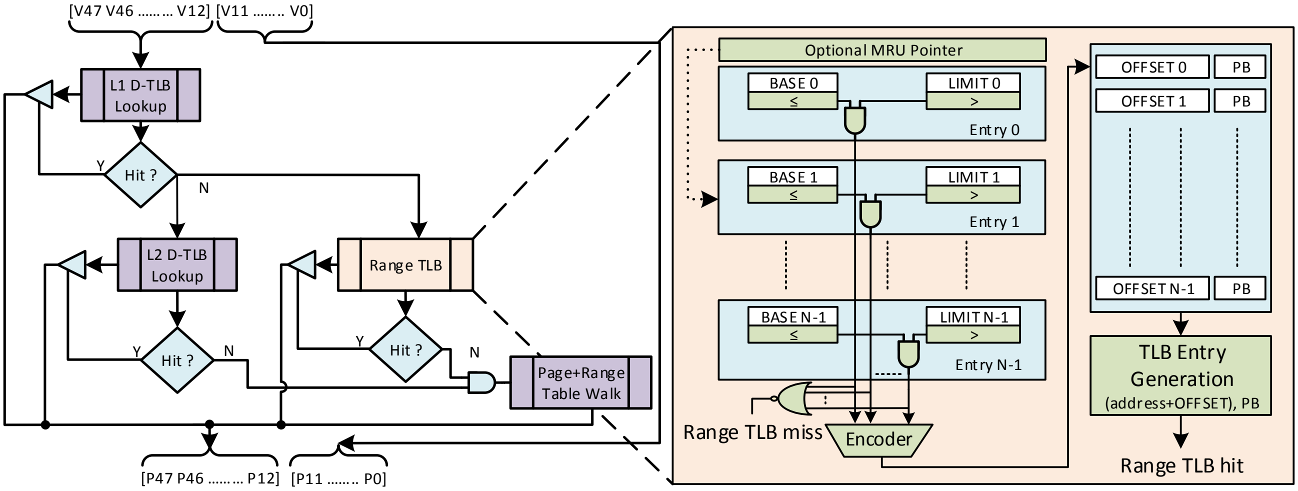

Figure 3: RMM hardware support consists primarily of a range TLB that is accessed in parallel with the last-level page TLB.

first checks the MRU Pointer and in case of a hit, skips the other entries. Otherwise, the range TLB checks all valid entries in parallel. Note that the MRU Pointer can serve translation requests faster than the corresponding page TLB and may further boost performance.

4.2. Range table

The range table is an *architecturally visible* per-process data structure that stores the process's range translations in memory. The role of the range table is similar to that of the page table. A hardware walker loads range translations from the range table on a range TLB miss, and the OS manages range table entries based on the application's memory management operations.

We propose using a B-Tree data structure with (BASE_i, LIMIT_i) as keys and OFFSET_i and protection bits as values to store the range table. B-trees are cache friendly and keep the data sorted to perform search and update operations in logarithmic time. Since a single B-Tree node may have multiple ranges and children, it is a dense representation of ranges.

The number of ranges per range table node defines the depth of the tree and the average number of node lookups to perform a search/update operation. Figure 4 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. Since each range translation is represented at page-granularity with the BASE (48 architectural bits -12 bits per page$=36$ bits), the LIMIT (36 bits), and the OFFSET and protection bits together (64-bits conventional PTE size), thus each range table node fits in two cache-lines. This design ensures the traversal of the range table is cache-friendly, accesses only a few cache lines per operation, and maintains the dense representation. Note that the range table is much smaller than a page table: a *single 4 KB page* stores 128 range translations, which is more than enough for almost all our workloads (Table 7). All the pointers to the children are physical addresses, which facilitate walking the range table in hardware.

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 to perform address translation, as we will explain next.

4.3. Handling misses in the range TLB

On a miss to the range TLB and corresponding page TLB, the hardware must fetch a translation from the memory. Two design issues arise with RMM at this point. First, should address translation hardware use the page table to fetch only the missing PTE or the range table to fetch the range translation? Second, how does the hardware determine if the missing translation is part of a range translation and avoid unnecessary lookups in the range table? Because ranges are redundant, there are several options.

Miss-handling order. RMM first fetches the missing translation from the page table, as all valid pages are guaranteed to be present, and installs it in the previous-level TLB so that the processor can continue executing the pending operation. This choice avoids additional latency from accessing the range table for pages that are not redundantly mapped. *In the background*, the range table walker hardware resolves whether the address falls in a range and if it does, updates the range table with the range table entry. Thus when both the range table and page TLB miss, the miss incurs the cost of a page-walk. Any updates to the range TLB occur *off the critical path*.

Identifying valid range translations. 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. Without this hint, available from redundancy, the range table walker would have to check the range table on every TLB miss. Alternatively, hardware could use prediction to decide whether to access the range table, which requires no changes to page table entries, but we did not evaluate this option.

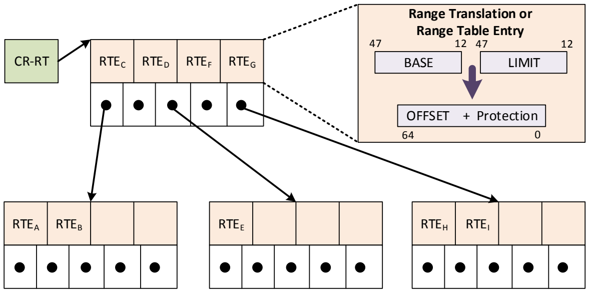

Figure 4: The range table stores the range translations for a process in memory. The OS manages the range table entries based on the applications memory management operations.

Walking the range table. Similar to the page table walker, RMM introduces the range table walker that consists of two comparators and a hardware state machine. The range table walker walks the range table in the background starting from the CR-RT register. The walker compares the missing address with the range translations in each range table node and follows the child pointers until it finds the corresponding range translation and installs it in the range TLB. To simplify the hardware, an OS handler could perform the range table lookup.
Shootdown. The OS uses the `INVLPG` instruction to invalidate stale virtual to physical translations (including changes in the protection bits) during the TLB shootdown process [16]. To ensure correct functionality, RMM modifies the `INVLPG` instruction to invalidate all TLB entries and any range TLB entry that contains the corresponding virtual page. The modified OS may thus use this instruction to keep all TLBs and the range TLB coherent through the TLB shootdown process. The OS may also associate each range TLB entry with an address space identifier, similar to TLB entries, to perform context switches without flushing the range TLB.

5. Operating System Support

RMM requires modest operating system (OS) modifications. The OS must create and manage range table entries in software and coordinate them with the page table. We modify the OS to increase the size of ranges with an eager paging allocation mechanism. We prototype these changes in Linux, but the design is applicable to other OSes.

5.1. Managing range translations

Similar to paging, the process control block in RMM stores a range table pointer (RT pointer) with the physical address of the root node of the range table. When the OS creates a process, it allocates space for the range table and sets the RT pointer. On every context switch, the OS copies the RT pointer to the CR-RT register and then the range table walker uses it to walk the range table.

The OS updates the range table when the application allocates or frees memory or the OS reclaims a page. The OS analyzes the contiguity of the affected page(s). Based on a *contiguity threshold* (e.g., 8 pages), the OS adds, updates, or removes a range translation from the range table. The OS avoids creating small range translations that could cause thrashing in the range TLB. The OS can modify the contiguity threshold dynamically, based on the current number and size of range translations, and the performance of the range TLB (option not explored). The OS updates the range bit in all the corresponding PTEs for the range to keep them consistent.

5.2. Contiguous memory allocation

Achieving a high hit ratio in the range TLB and thus low virtual memory overheads requires a small number of very large range translations that satisfy most virtual address translation requests. To this end, RMM modifies the OS memory allocation mechanism to use *eager paging*, which strives to allocate the largest possible range of contiguous virtual pages to contiguous physical pages. Eager paging requires modest changes to Linux's default buddy page allocator.
Default buddy allocator. The buddy allocator splits physical memory in blocks of 2^{order} pages, and manages the blocks using separate *free-lists* per block size. A kernel compile-time parameter defines the *maximum size of memory blocks* (2^{max_order}) and hence the total number of the free-lists. The buddy allocator organizes each free-list in power-of-two blocks and satisfies requests from the free-list of the smallest size. If a block of the desired 2^i size is not available (i.e., free-list[i] is empty), the OS finds the next larger 2^{i+k} size free block, going from $k = 1, 2, ...$ until it finds the smallest free block large enough to satisfy the request. The OS then iteratively splits a block in two, until it creates a free block of the desired 2^i size. It then assigns one free block to the allocation and adds any other free blocks it creates to the appropriate free-lists. When the application later frees a 2^i block, the OS examines its corresponding buddy block (identified by its address). If this block is free, the OS coalesces the two blocks, resulting in a 2^{i+1} block. The buddy allocator thus easily splits and merges blocks during allocations and deallocations respectively.

Despite contiguous pages in the buddy heap, in practice most allocations are of a single page because of demand paging. Operating systems use demand paging to reduce allocation latency by deferring page instantiation until the application actually references the page. Therefore, the application's allocation does not trigger OS allocation, but rather when the application first writes or reads a page, the OS allocates a single page (from free-list[0]). Demand allocation at *access-time* 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. Eager paging improves the generation of large range translations by allocating consecutive physical pages to consecutive virtual pages eagerly at allocation, rather than lazily on demand at access time. At *allocation request time*

```
compute the memory fragmentation;
if memory fragmentation $\leq$ threshold then
    // use eager paging;
    while number of pages > 0 do
        for (i = MAX_ORDER-1; i $\geq$ 0; i–) do
            if freelist[i] $\geq 0$ and $2^i \leq$ 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 5: 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 allocates pages at *access-time*.

(e.g., when the application performs an mmap, mremap or brk call), if the request is larger than the range threshold, the OS establishes one or more range translations for the entire request and updates the corresponding range and page table entries. We note that demand paging replaced eager paging in early systems. However, one motivation for demand paging was to limit unnecessary swapping in multiprogrammed workloads, which modern large memories make less common [10]. We find that the high cost of TLB misses, makes eager paging a better choice with RMM hardware in most cases.

Eager paging increases latency during allocation and may induce fragmentation, because the OS must instantiate all pages in memory, even those the application never uses. However unused memory is not permanently wasted. The OS could monitor memory use in range translations and reclaim ranges and pages with standard paging mechanisms, but we leave this exploration for future work. Allocating memory at request-time generates larger range translations compared to the access-time policy of demand paging and improves the effectiveness of RMM hardware.

Algorithm. Figure 5 shows simplified pseudocode for eager paging. If the application requests an allocation of size N×pages, eager paging allocates the 2^i block, as described above. This simple algorithm only provides contiguity up to the maximum managed block size. If the application requests more memory than the maximum managed block, the OS will allocate multiple maximum blocks. Two optimizations further improve contiguity. First, eager paging could sort the blocks in the free-lists, to coalesce multiple blocks and generate range translations larger than the maximum block. Second, to generate large range translations from allocations that are smaller than the maximum block, eager paging could request a block from a larger size free-list, assign the necessary pages, and return the remaining blocks to the corresponding smaller sized free-lists. These enhancements introduce additional trade-offs that warrant more investigation. Note that in our RMM prototype, we did not implement these two enhancements. Nonetheless, the simple eager paging algorithm generates large range translations for a variety of block sizes and exploits the clustering behavior of the buddy allocator [38, 39].

Finally, eager paging is only effective when memory fragmentation remains low and there is ample space to populate ranges at request time. If memory fragmentation or pressure increases, the OS may fall back to its default paging allocation.

6. Discussion

This section discusses some of the hardware and operating systems issues that a production implementation should consider, but leaves the implications for automatic and explicit memory management and for applications as future work.

TLB friendly workloads. If an application has small memory footprint and experiences a low page TLB miss rate, the range TLB may provide little performance benefit while increasing the dynamic energy due to range TLB accesses. The OS can monitor the memory footprint and then dynamically enable and disable the range TLB. The OS would still allocate ranges and populate the range table, but then it could selectively enable the range TLB based on performance-counter measurements and workload memory allocation.

Accessed & Dirty bits. The TLB in x86 processors is responsible for setting the *accessed bit* in the corresponding PTE in memory on the first access to a page and the *dirty bit* on the first write. The range TLB does not store per-page accessed/dirty bits for the individual pages that compose a range translation. Thus, on a range TLB hit, the range TLB cannot determine whether it should set the accessed or dirty bit. The OS may address this issue by setting the accessed and dirty bits for all the individual pages of a range translation eagerly at allocation time, instead of at access or write time. If the OS needs to reclaim or swap a page in an active range because of memory pressure, it may. Because the OS manages physical memory at the page-granularity—not at the range granularity—it may reclaim and swap individual pages by dissolving a range completely and then evicting and swapping

Suite	Description	Input	Memory
SPEC 2006	compute & memory intensive single-threaded workloads	astar	350 MB
		cactusADM	690 MB
		GemsFDTD	860 MB
		mcf	1.7 GB
		omnetpp	165 MB
		soplex	860 MB
PARSEC	RMS multi-threaded workloads	canneal	780 MB
		streamcluster	120 MB
BioBench	Bioinformatics single-threaded workloads	mummer	470 MB
		tigr	610 MB
Big memory	Generation, compression and search of graphs	Graph500	73 GB
	In-memory key-value cache	Memcached	75 GB
	NASA's high performance parallel benchmark suite.	NPB:CG	54 GB
	Random access benchmark	GUPS	67 GB

Table 4: Workload description and memory footprint.

pages individually. Another option is for the OS to break a range in to multiple smaller ranges and dissolve one of the resulting ranges.

Copy-on-write. Copy-on-write is a virtual memory optimization in which processes initially share pages and the OS only creates separate individual pages when one of the processes modifies the page. This mechanism ensures that these changes are only visible to the owning process and to no other process. To implement this functionality, copy-on-write uses per-page protection bits that trigger a fault when the page is modified. On a fault, the OS copies the page and updates the protection bits in the page table. With RMM, the range translations hold the protection bits at range granularity, not on individual pages. One simple approach is to use range translations for read-only shared ranges, but dissolve a range into pages when a process writes to any of its pages. Alternatively, the OS could copy the entire range translation on a fault.

Fragmentation. Long-running server and desktop systems will execute multiple processes at once and a variety of workload mixes. Frequent memory management requests from complex workloads may cause physical memory fragmentation and limit the performance of RMM. If the OS cannot find a sufficiently large range of free pages in memory, it should default to paging-only and disable the range TLB. However, abundant memory capacity coupled with fragmentation is not uncommon, since a few pages scattered throughout memory can cause considerable fragmentation [18]. In this case, the OS could perform full compaction [10, 39], or partial compaction with techniques adapted from garbage collection [17, 18].

7. Methodology

To evaluate virtual memory system performance on large memory workloads, we implement our OS modifications in Linux, define RMM hardware with respect to a recent Intel x86-64 Xeon core, and report overheads using a combination of hardware performance counters from application executions and functional TLB simulation.

	Description
Processor	Dual-socket Intel Xeon E5-2430 (Sandy Bridge), 6 cores/socket, 2 threads/core, 2.2 GHz
Memory	96 GB DDR3 1066MHz
OS	Linux kernel version 3.15.5
L1 DTLB	4 KB pages: 64-entry, 4-way associative 2 MB pages: 32-entry, 4-way associative 1 GB pages: 4-entry, fully associative
L1 ITLB	4 KB pages: 128-entry, 4-way associative 2 MB pages: 8-entry, fully associative
L2 TLB	4 KB pages: 512-entry, 4-way associative 2 MB pages:
range TLB	unrestricted sizes: 32-entry, fully associative

Table 5: System configurations and per-core TLB hierarchy.

RMM operating system prototype. We prototype the RMM operating system changes in Linux x86-64 with kernel v3.15.5. We implement the management of the range tables by intercepting all kernel memory-management operations. We implement range creation and eager paging by modifying the *mmap*, *brk* and *mremap* system calls. For our prototype range table, we implement a simple linked list rather than a B-tree. Because our applications spend only a tiny fraction of their time in the OS and the range TLB refill is not on the processor's critical path, this simplification does not affect our results.

We use a contiguity threshold of 32 KB (8 pages) to define the minimum size of a range translation. To increase the maximum size of a range, we increase the maximum allocation size in the buddy allocator to 2 GB, up from 4 MB by modifying the `max_order` parameter of the buddy allocator from 11 to 20. Because the default *glibc* memory management implementation does not coalesce allocations into fixed-size virtual ranges, we instead use the *TCMalloc* library [5]. In addition, we modify TCMalloc to increase the maximum allocation size from 256 KB to 32 MB.

RMM hardware emulation. We evaluate the RMM hardware described in Section 4 with Intel Sandy Bridge core shown in Table 5. 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.

To measure the overheads of RMM, we combine performance counter measurements from native executions with TLB performance emulation using a modified version of BadgerTrap [22]. Compared to cycle-accurate simulation on these workloads, this approach reduces weeks of simulation time by orders of magnitude. Previous virtual memory system performance studies use this same approach [10, 12, 23].

BadgerTrap instruments x86-64 TLB misses. We add a functional range TLB simulator in the kernel that BadgerTrap invokes. On each page L2 TLB miss, BadgerTrap performs a range TLB lookup. Note that the actual implementation would perform the range TLB lookup in parallel, rather than after the L2 TLB miss. This emulation may thus underestimate the benefit of the range TLB, because the real hardware will install

Performance Model	
Ideal execution time	$T_{ideal} = T_{2M} - C_{2M}$
Average page-walk cost	$AvgC_{4K/2M} = C_{4K/2M}/M_{4K/2M}$
Measured page-walk overhead	$Over_{4K/2M} = C_{4K/2M}/T_{ideal}$
Simulated page-walk overhead	$Over_{SIM} = M_{SIM} * AvgC_{4K}/T_{ideal}$
T: Total execution cycles	$M_{4K/2M}$: page-walks with 4K/2M
C: Cycles spent in page-walks	M_{SIM}: Simulated page-walks

Table 6: Performance model based on hardware performance counters and BadgerTrap.

a missing page table entry, even if the virtual address hits in the range TLB. The actual RMM implementation reduces traffic to the L2 page TLB on range TLB hits, freeing up page TLB entries and potentially making it more effective. This simulation methodology may itself perturb TLB behavior. To minimize this problem, we allocate a 2 MB page in the kernel for the simulator itself, which reduces the differences with an unmodified kernel to less than 5%.

Performance model. We estimate the impact of RMM on system performance with the following methodology. First, we run the applications on the real system (Table 5) with realistic input sets until completion and collect processor and TLB statistics using hardware performance counters. We use the Linux *perf* utility [4] to read the performance counters. We collect total execution cycles, misses for L2 TLB, and cycles spent in page-walks. Based on these measurements we calculate (i) the ideal execution time (no virtual memory overhead), (ii) the measured overhead spent in page-walks, and (iii) the estimated overhead with the simulated hardware mechanisms based on the fraction of reduced page-walks, using a simple linear model [10, 23] given in Table 6.

Benchmarks. RMM is designed for a wide range of applications from desktop applications to big-memory workloads executing on scale-out servers. To evaluate the effectiveness of RMM, we select workloads with poor TLB performance from SPEC 2006 [25], BioBench [7], Parsec [15] and big-memory workloads [10] as summarized in Table 4. We execute each application sequentially on a single test machine without rebooting between experiments.

8. Results

This section evaluates the cost of address translation, the impact of eager paging, and implications on energy of RMM, and shows substantial improvements in performance over current and proposed systems.

We compare RMM performance to the following systems. (i) We measure the virtual memory overheads of a commodity x86-64 processor (see Table 5) with 4 KB pages, 2 MB pages with transparent huge pages, and 1 GB pages with libhugetlbfs using hardware performance counters. (ii) We emulate multipage mappings in BadgerTrap. We implement the Clustered TLB approach [38] of Pham et al., configured with 512 fully-associative entries. Each entry indexes up to an 8-page cluster, shown best by Clustered TLB [38]. We use eager paging to increase the opportunities to form multipages, improving on the original implementation. (iii) We emulate the performance of ideal direct segments. We assume all fixed-size memory regions that live for more than 80% of a program's execution time can be coalesced in a single contiguous range, which can be used to estimate the reduction in TLB misses with direct segment hardware [10].

8.1. Performance analysis

Figure 6 shows the overhead spent in page-walks for RMM compared to other techniques. The 4 KB, 2 MB Transparent Huge Pages (THP) [6] and 1 GB [1] configurations show the *measured* overhead for the three different page sizes available on x86-64 processors. All other configurations are emulated. The CTLB bars show Clustered TLB [38] results. The DS bars show direct segments [10] results and the RMM bars show the 32-entry range TLB results.

RMM performs well on all configurations for all workloads, improving substantially over all the other approaches, except direct segments. RMM eliminates the vast majority of page-walks, significantly outperforms the Clustered TLB (CTLB), huge pages (THP and 1GB) and achieves similar or better performance to direct segments, but has none of its limitations. On average, RMM reduces the overhead of virtual memory to less than 1%.

For most workloads, the base page size (4 KB) incurs high overheads. For example, mcf, cactusADM, and graph500 spend 42%, 39% and 29% of execution time in page-walks due to TLB misses. Even the applications with smaller working sets, such as astar, omnetpp, and mummer, still suffer substantial paging overheads using 4 KB pages.

Clustered TLB (CTLB) only offers limited reductions in overhead and only for small-memory workloads. CTLB performs better than 4 KB pages on small-memory workloads, such as cactusADM, canneal, and omnetpp. However, CTLB provides little benefit on big-memory workloads and performs worse than THP overall.

Huge pages (THP and 1 GB) reduce virtual memory overheads for all workloads but still leave room for improvement. The limited hardware support for huge pages (e.g., few TLB entries), poor application memory locality, and the mismatch of their sizes with the virtual memory contiguity all contribute to the remaining overheads.

Direct segments achieve negligible overheads on big-memory workloads and some small-memory workloads. But, direct segments poorly serve workloads that require multiple ranges, such as omnetpp, canneal, or those that use memory-mapped files such as mummer. Compared to direct segments, RMM is a better choice because it achieves similar or better performance on all workloads.

Redundant Memory Mappings achieve negligible overhead—essentially eliminating virtual memory overheads for many workloads. Only one workload has greater than 2% overhead, GUPS. As our sensitivity analysis in the next section shows,

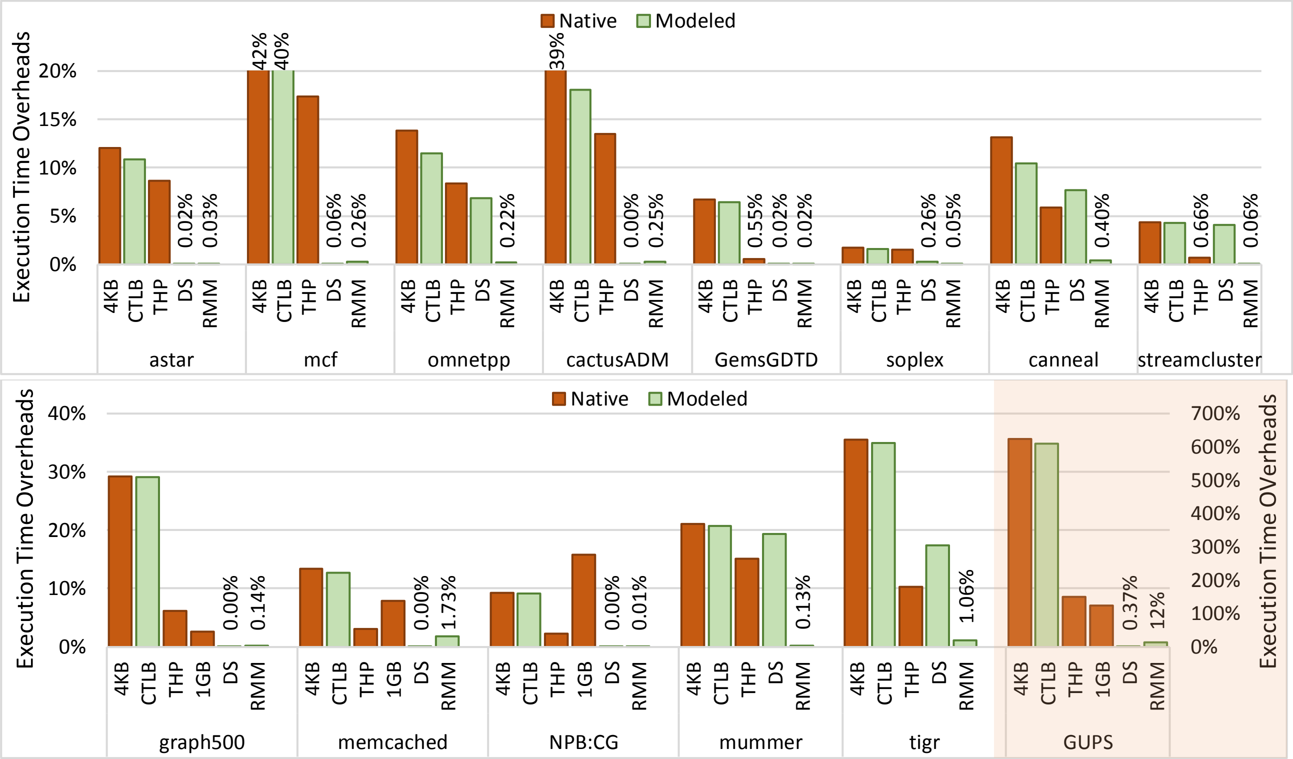

Figure 6: Execution time overheads due to page-walks for SPEC 2006 and PARSEC (top) big-memory and BioBench (bottom) workloads. GUPS uses the right y-axis and thus shaded separately. 1GB pages are only applicable to big-memory workloads.

GUPS requires at least a 64-entry range TLB to achieve less than 1% overhead. Overall, RMM performs consistently better than the alternatives and in many cases eliminates the performance cost of address translation.

8.2. Range TLB sensitivity analysis

To achieve high performance, the range TLB must be large enough to satisfy most L1 TLB misses. Figure 7 shows the range TLB miss ratio as a function of the numbers of entries. We observe that a handful of workloads, such as cactusADM, memcached, tigr, and GUPS, suffer from high miss ratios with a 16-entry range TLB. Overall, a 32-entry range TLB eliminates more than 99% of misses for most workloads (97.9% on average), delivering a good trade-off of performance for the required area and power.

We also note that a single-entry range TLB is insufficient to eliminate virtual memory overheads. Most applications require multiple range table entries, especially those with large working sets, such as cactusADM, GemsFDTD and GUPS, and those with large numbers of ranges, such as memcached, mummer, and tigr. However, the single-entry results illustrate that the optional MRU Pointer would be effective at saving dynamic energy and latency in many cases. It reduces accesses to the range TLB by more than 50% for astar, omnetpp, canneal, streamcluster, and graph500.

8.3. Impact of eager paging

Eager paging increases range size by instantiating physical pages when the application allocates memory, rather than when the application first writes or reads a page. Table 7 shows the effect of eager paging on the number and size of range, and on time and memory overheads, compared to default demand paging. Default demand paging includes forming THPs, which we translate to ranges.

The first two sections of Table 7 (demand paging and eager paging) compare the number of ranges, the percentage of the memory footprint covered by ranges with a contiguity threshold of 8 pages, and the range sizes (median, average, maximum) in terms of pages, created by demand and eager paging. Eager paging (i) lowers the median range size for small-memory workloads because it allocates fewer medium-sized ranges (the median for demand paging is usually 512, i.e., 2 MB regions, due to THP), (ii) increases the median range for big-memory workloads because it allocates fewer small and medium-sized ranges, and (iii) increases the average

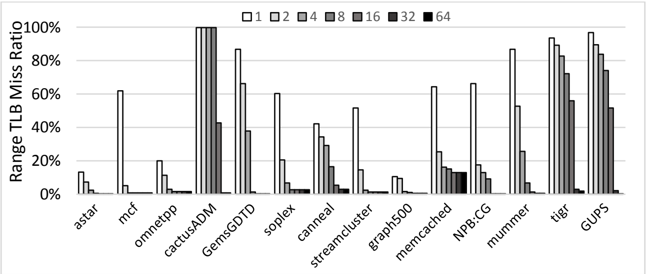

Figure 7: Range TLB miss ratio as a function of the number of range TLB entries.

	Demand Paging					Eager Paging						
	# ranges	% memory	range size in 4 KB pages			# ranges	% memory	range size in 4 KB pages			% time	% memory
Benchmark			median	average	max			median	average	max	overhead	overhead
astar	170	94.52	512	478	1024	33	99.69	32	2810	8192	-1.15	8.14
mcf	449	99.72	512	957	4608	28	99.94	24	15637	262143	-4.10	1.58
omnetpp	91	96.30	512	438	512	27	99.03	20	1617	8192	-0.50	6.34
cactusADM	311	99.50	512	549	1024	70	99.84	8192	5537	8192	0.85	125.90
GemsFDTD	326	98.76	512	651	2048	61	99.75	256	3613	16384	11.65	2.74
soplex	333	98.32	512	633	4096	54	99.85	128	4502	81919	-1.78	13.45
canneal	410	95.96	202	453	1024	46	99.82	189	4248	32767	1.15	0.99
streamcluster	65	95.73	512	439	512	32	99.18	21	1122	16383	-1.61	21.41
mummer	837	85.51	32	120	512	61	99.68	512	1940	32768	-1.55	0.87
tigr	1149	95.16	16	123	1536	167	99.51	32	889	16384	-1.97	0.01
Graph500	18574	99.97	512	984	524288	32	99.99	2048	187236	524288	2.56	0.27
Memcached	1540	99.97	1024	29629	524288	86	99.99	2048	216857	524288	-3.95	0.17
NPB:CG	22746	99.98	512	586	1536	95	99.99	4096	146861	524288	0.87	4.56
GUPS	705	99.99	512	23823	524288	62	99.99	524288	271039	524288	-0.61	0.05

Table 7: Impact of eager paging on ranges, time, and memory compared to demand paging (with Transparent Huge Pages).

and maximum range size for all workloads because it allocates larger blocks from the buddy allocator. Overall eager paging generates orders of magnitude fewer ranges that cover a larger percentage of memory for all applications compared to demand paging. Thus eager paging assists in achieving high range TLB hit ratio with few entries.

Eager paging alters execution by changing when and how pages, even used pages, are allocated to physical memory. We measure execution overhead due to eager paging by running applications with the eager paging operating system support, but without the hardware emulation. Table 7 shows that the execution time for most applications is relatively unchanged. A few get faster: mcf and memcached improve by 4.1% and 3.9%. However, GemsFDTD degrades by 11%. In this case, the changes in physical page allocation affect cache indexing, increasing cache conflicts. Various orthogonal mechanisms address this problem [19, 43].

Eager paging anticipates that the application will use the requested memory regions and may thus increase the memory footprint. The last column of Table 7 reports the memory footprint increase with eager paging. Eager paging increases memory by a small amount for three of the big-memory workloads, and by less than 10% for 7 of the remaining 10 workloads. Eager paging increases memory substantially on cactusADM and NPB:CG (the percentage is low, but totals 2.3 GB), mainly because of instantiating memory that these applications request but never use, and because of modifying TCMalloc to increase contiguity. Thus RMM trades increased memory for better performance, a common tradeoff when memory is cheap and plentiful. Note that the OS can convert a range to pages or abandon ranges altogether under memory pressure as discussed in Section 6.

8.4. Energy

The primary RMM effect on energy is executing the application faster, which improves static energy of system. According to our performance model, RMM improves performance by 2-84% and thus saves a similar ratio of static energy.

Secondary effects include the static and dynamic energy of the additional RMM hardware. The system accesses the range TLB in parallel with the L2 TLB, consuming dynamic energy on a L1 TLB miss. The dynamic energy of a 32-entry range TLB is relatively small with respect to the entire chip, and lower than of a fully-associative 128-entry L1 TLB (e.g., SPARC M7 [40]). Furthermore, replacing misses in the L2 TLB with hits in the range TLB saves dynamic energy by avoiding a page-walk that performs up to four memory operations. The OS can identify workloads for which the range TLB provides little benefit and disable the range TLB (see Section 6), eliminating its dynamic energy.

To further explore power and energy impact of the range TLB on the address translation path, we implemented a 32-entry range TLB and a 512-entry L2 page TLB with search latency of six cycles in Bluespec. We then synthesized both designs with the Cadence RTL Compiler using 45nm technology (tsmc45gs standard cell library) at 3.49GHz under typical conditions. We specified that timing should be prioritized over area and power.* This analysis shows that the range TLB adds power that is less than half (39.6%) of L2 TLB's power. Moreover, the range TLB area is only 13% of the L2 TLB area. These results and the high range TLB hit ratio indicate that simply increasing the number of entries in the L2 TLB, which would also incur a cycle penalty on the critical path, at the same power and area budget will not be as effective as the RMM design.

9. Related Work

Virtual memory remains an active area of research. Previous work shows that limited TLB reach results in costly page-walks that degrade application performance, often substan-

*Due to license limitations, we synthesized memory cells of both structures with D flip-flops instead of SRAM cells.

tially [10, 13, 14, 23, 29, 31]. Section 2 described the qualitative differences between RMM and the most closely related work on multipage mappings (sub-blocked TLBs [47], CoLT [39], Clustered TLBs [38]), huge pages [1, 6, 36], and direct segments [10, 23], and Section 8 showed quantitatively that RMM substantially improves over them. Below we discuss other mechanisms that help reduce the overhead of TLB misses, and how they relate to RMM.

One common way to reduce the cost of a TLB miss is through accelerating the page-walks. Commodity processors cache Page Table Entries (PTEs) in data caches to accelerate page-walks [28]. Software-defined TLB structures, such as TSBs in SPARC [46] and software-managed sections of TLB in Intel Itanium [3], pin entries in the TLB to improve performance. MMU caches also reduce latency of page-walks by caching intermediate levels of the page table, skipping one or more memory references during the page-walk [8, 12, 27]. RMM is orthogonal to these approaches since it eliminates some page-walks altogether. When page-walks are required in RMM, these mechanisms can accelerate them.

Virtual memory overhead can also be reduced by lowering the number of TLB misses. For instance, the hardware can prefetch PTEs in to the TLB in advance of their use [14, 30, 42]. However, the effectiveness of prefetching is limited by the predictability of the memory access patterns. Alternatively, Barr *et al.* [9] proposed speculative translation based on huge pages. Similar to prefetching, this mechanism depends on the TLB behavior and favors sequential patterns. Last-level shared TLBs [13, 34] and cooperative TLBs [45] increase the TLB reach and reduce the number of page-walks. Similarly, Papadopoulou *et al.* [37] proposed a prediction mechanism that allows all page sizes to share a single set-associative TLB. In addition, Du *et al.* [20] proposed mechanisms to allow huge pages to be formed even in the presence of retired physical pages. However, the total TLB reach is still limited for memory intensive applications since each TLB entry maps a single page unless ranges are used [31]. In contrast to these approaches, RMM generates and caches translations for arbitrarily large ranges. Thus RMM is less susceptible to irregularities in the application's access patterns and improves address translation for large memories.

Commercial processors have also used segmentation to implement virtual memory. The Burroughs B5000 [33] was an early user of pure segments. The 8086 [2] and iAPX 432 [26] processors also supported pure segmentation without paging. Later IA-32 processors provided segments on top of paging [29], but without any translation benefits for segments. In contrast to previous segmentation approaches, RMM combines the flexibility and robustness of paging while enjoying the translation performance of segmentation.

Prior work also proposes virtual caches to reduce the performance and energy overheads of the TLB by only translating after a cache miss [11, 29, 50]. However for those workloads that suffer many TLB misses due to poor locality, virtual caches just shift the translation to a lower level of the cache hierarchy while increasing the complexity of the system.

Finally, our proposed architecture resembles prior works in fine-grained memory protection [24, 48, 49], in the sense that both exploit range behavior. However, instead of exploiting only the contiguity of fine-grained protection rights across memory regions, RMM enhances and exploits the contiguity in memory allocation to accelerate address translation.

10. Summary

We propose Redundant Memory Mappings, a novel and robust translation mechanism, that improves performance by reducing the cost of virtual memory across all our workloads. RMM efficiently represents ranges of arbitrarily-many pages that are virtually and physically contiguous and layers this representation and its hardware redundantly to page tables and paging hardware. RMM requires only modest changes to existing hardware and operating systems. The resulting system delivers a virtual memory system that is high performance, flexible, and completely transparent to applications.

Acknowledgements

We thank our anonymous reviewers and Dan Gibson for their insightful comments and feedback on the paper. We thank Wisconsin Computer Architecture Affiliates for their feedback on an early version of the work. We thank Oriol Arcas and Ivan Ratkovic for the Bluespec implementation and the synthesis results of the range TLB.

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 Hill). Furkan Ayar's contribution to the paper occurred while on internship at Barcelona Supercomputing Center. Vasilis Karakostas is also supported by an FPU research grant from the Spanish MEC. Hill has a significant financial interest in AMD.

References

[1] "Huge Pages Part 1 (Introduction)," http://lwn.net/Articles/374424/.

[2] "Intel 8086 - Wikipedia," http://en.wikipedia.org/wiki/Intel_8086.

[3] "Intel® itanium® architecture developer's manual, vol. 2," http://www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-s-oftware-developer-rev-2-3-vol-2-manual.html.

[4] "perf: Linux profiling with performance counters ," https://perf.wiki.kernel.org/index.php/Main_Page.

[5] "TCMalloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[6] "Transparent Huge Pages in 2.6.38," http://lwn.net/Articles/423584/.

[7] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pp. 2–9, 2005.

[8] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don'T Walk (the Page Table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 48–59, 2010.

[9] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pp. 307–318, 2011.

[10] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 237–248, 2013.

[11] A. Basu, M. D. Hill, and M. M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pp. 297–308, 2012.

[12] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 383–394, 2013.

[13] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, pp. 62–63, 2011.

[14] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pp. 29–40, 2009.

[15] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[16] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, "Translation Lookaside Buffer Consistency: A Software Approach," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 113–122, 1989.

[17] S. M. Blackburn and K. S. McKinley, "Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 22–32, 2008.

[18] N. Cohen and E. Petrank, "Limitations of partial compaction: Towards practical bounds," *SIGPLAN Not.*, vol. 48, no. 6, pp. 309–320, 2013.

[19] C. Ding and K. Kennedy, "Inter-array Data Regrouping," in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pp. 149–163, 2000.

[20] Y. Du, M. Zhou, B. Childers, D. Mosse, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, pp. 223–234, Feb 2015.

[21] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 37–48, 2012.

[22] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "BadgerTrap: A Tool to Instrument x86-64 TLB Misses," *SIGARCH Comput. Archit. News*, vol. 42, no. 2, pp. 20–23, Sep. 2014.

[23] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *MICRO-47: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 178–189, 2014.

[24] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin, "A Case for Unlimited Watchpoints," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 159–172, 2012.

[25] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[26] Intel Corporation, "Introduction to the iAPX 432 Architecture," 1981, no. 171821-001.

[27] Intel Corporation, "TLBs, Paging-Structure Caches and their Invalidation," 2008, no. 317080-003.

[28] Intel Corporation, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," April 2012, no. 248966-026.

[29] B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro*, vol. 18, no. 4, pp. 60–75, Jul. 1998.

[30] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 195–206, 2002.

[31] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift, "Performance Analysis of the Memory Management Unit under Scale-out Workloads," in *Proceedings of the 2014 IEEE International Symposium on Workload Characterization*, pp. 1–12, 2014.

[32] J.-Y. Kim and H.-J. Yoo, "Bitwise Competition Logic for Compact Digital Comparator," in *Proceedings of the 2007 IEEE Asian Solid-State Circuits Conference*, 2007.

[33] W. Lonehgan and P. King, "Design of the b 5000 system," *Datamation*, vol. 7, no. 5, May 1961.

[34] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 2:1–2:38, Apr. 2013.

[35] MIPS Technologies, Incorporated, "MIPS32 Architecture for Programmers Volume iii: The MIPS Privileged Resource Architecture," 2001, no. MD00090, Revision 0.95.

[36] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, pp. 89–104, 2002.

[37] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, pp. 210–222, Feb 2015.

[38] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*, pp. 558–567, 2014.

[39] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 258–269, 2012.

[40] S. Phillips, "M7: Next Generation SPARC," in *Hot Chips: A Symposium on High Performance Chips*, 2014.

[41] D. Quintero, S. Chabrolles, C. H. Chen, M. Dhandapani, T. Holloway, C. Jadhav, S. K. Kim, S. Kurian, B. Raj, R. Resende, B. Roden, N. Srinivasan, R. Wale, W. Zanatta, and Z. Zhang, "IBM Power Systems Performance Guide Implementing and Optimizing," 2013.

[42] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB Preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 117–127, 2000.

[43] A. Seznec, "A Case for Two-way Skewed-associative Caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 169–178, 1993.

[44] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, "Sparc T4: A Dynamically Threaded Server-on-a-Chip," *IEEE Micro*, vol. 32, no. 2, pp. 8–19, Mar. 2012.

[45] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 313–324, 2010.

[46] Sun Microsystems, "UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007."

[47] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 171–182, 1994.

[48] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood, "A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 94–105, 2008.

[49] E. Witchel, J. Cates, and K. Asanović, "Mondrian Memory Protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 304–316, 2002.

[50] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton, "An In-cache Address Translation Mechanism," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 358–365, 1986.