Memory-Efficient Hashed Page Tables

Jovan Stojkovic, Namrata Mantri, †Dimitrios Skarlatos, Tianyin Xu, Josep Torrellas
University of Illinois at Urbana-Champaign †*Carnegie Mellon University*
{jovans2, nmantri2}@illinois.edu, dskarlat@cs.cmu.edu, {tyxu, torrella}@illinois.edu

***Abstract*—Conventional radix-tree page tables have scalability challenges, as address translation following a TLB miss potentially requires multiple memory accesses *in sequence*. An alternative is hashed page tables (HPTs) where, conceptually, address translation needs only one memory access. Traditionally, HPTs have been shunned due to high costs of handling conflicts and other limitations. However, recent advances have made HPTs compelling. Still, a major issue in HPT designs is their requirement for substantial *contiguous* physical memory.**

This paper addresses this problem. To minimize HPTs' contiguous memory needs, it introduces the *Logical to Physical (L2P) Table* and the use of *Dynamically-Changing Chunk Sizes*. These techniques break down the HPT into discontiguous physical-memory chunks. In addition, the paper also introduces two techniques that minimize HPTs' total memory needs and, indirectly, reduce the memory contiguity requirements. These techniques are *In-place Page Table Resizing* and *Per-way Resizing*. We call our complete design *Memory-Efficient HPTs* (ME-HPTs). Compared to state-of-the-art HPTs, ME-HPTs: (i) reduce the contiguous memory allocation needs by 92% on average, and (ii) improve the performance by 8.9% on average. For the two most demanding workloads, the contiguous memory requirements decrease from 64MB to 1MB. In addition, compared to state-of-the-art radix-tree page tables, ME-HPTs achieve an average speedup of 1.23× (without huge pages) and 1.28× (with huge pages).

***Index Terms*—Virtual memory, Page tables, Hashed page tables**

I. Introduction

State-of-the-art page tables use the radix-tree organization [5], [42]. On a TLB miss, address translation proceeds by walking a tree of pages that progressively direct the search to the leaf that contains the desired translation. This approach uses memory efficiently and has been highly optimized with multiple caching structures over decades. However, it is hardly scalable. The reason is that, to obtain the correct translation, the system potentially needs to perform up to four memory accesses *in sequence*. Each access uses as its address the value returned by the previous access. This process can be slow and does not leverage the memory-level parallelism afforded by modern processors. Furthermore, this process is getting slower, as another level is being added to the translation tree to cover the larger memory needs of emerging applications [40], [41].

An alternative to radix-tree page tables is hashed page tables (HPTs) [18], [24], [26], [33], [36], [37], [39], [44], [45], [77], [83], [88]. Here, translations are kept in a table. On a TLB miss, address translation proceeds by hashing the virtual page number and using the hash key to index the table to retrieve the physical page number. Assuming that there are no hash collisions, this approach only needs one memory access for address translation.

Traditionally, HPTs have been out of favor for at least three reasons pointed out by Barr et al. [9]. First, accesses to the page table lack spatial locality. This is because hashing scatters the translations of contiguous virtual pages. Second, the need to associate a hash tag (i.e., the virtual page number) with each HPT entry consumes space. Finally, handling hash collisions is expensive: either sophisticated hardware or the OS need to walk over the colliding entries [9], [39], [88].

An additional problem is that the theoretical idea of a single global HPT that holds page table entries for all the active processes in the machine does not work [24], [83], [88]. The reason is that, to support page sharing across processes and multiple page sizes, one needs additional levels of indirection in the translation. Further, on process termination, the HPT has to be sequentially searched to remove obsolete entries and fix-up conflicts. The alternative is to have per-process HPTs. However, this approach is challenging [77], as it is unclear how to size the per-process HPTs. Allocating a large HPT for each process risks exhausting memory.

Recent advances have made HPTs more compelling. For example, to improve HPT locality, Yaniv and Tsafrir [88] place multiple contiguous page table entries together in a single cache line. Further, they encode the hash tag using unused bits in each entry. Also, Skarlatos et al. [77] use Cuckoo hashing to effectively handle hash conflicts. In addition, they support per-process HPTs by assigning small HPTs when processes are created, and dynamically growing the HPTs relatively cheaply with Elastic Cuckoo Hashing.

Still, an important drawback in HPT designs is their need for substantial *contiguous* physical memory. This need comes from the apparent requirement to place the HPT (or, more precisely, individual HPT ways) in contiguous physical memory. In our measurements, an HPT way can reach 64MB. In practice, allocating a large chunk of contiguous memory in a busy machine is often time-consuming and, under some conditions, may cause program failure. In contrast, in radix-tree page tables, finding contiguous memory is not a concern because memory is allocated one page at a time.

This paper addresses this HPT shortcoming. To minimize HPT's contiguous memory needs, we introduce two techniques: the *Logical to Physical (L2P) Table* and the use of *Dynamically-Changing Chunk Sizes*. These techniques break down the HPT into memory-efficient, discontiguous physical-memory chunks. Moreover, we also introduce two more techniques that minimize HPTs' total memory needs and, indirectly, reduce the memory contiguity requirements. These techniques are *In-place Page Table Resizing* and *Per-way*

Resizing. We call our resulting design *Memory-Efficient HPTs* (ME-HPTs).

We evaluate ME-HPTs with full-system simulations running a set of memory-intensive workloads. Compared to state-of-the-art HPTs, ME-HPTs: (i) reduce the contiguous memory allocation needs by 92% on average, and (ii) improve the performance by 8.9% on average. For the two most demanding workloads, the contiguous memory requirements decrease from 64MB to 1MB. In addition, compared to state-of-the-art radix-tree page tables, ME-HPTs speed-up the workloads by an average of 1.23× (without huge pages) and 1.28× (with huge pages).

This paper's contributions are:

• Memory-Efficient HPTs (ME-HPTs), which introduce four new techniques that, directly or indirectly, minimize the contiguous physical memory needed by HPTs.

• An evaluation of the ME-HPT techniques, which shows that they solve the memory contiguity limitation of HPTs.

II. Background

A. Limitations of Radix-Tree Page Tables

Current processors overwhelmingly use radix-tree page tables, which organize the page tables in a tree. On a TLB miss, the hardware sequentially walks over each level of the tree. Figure 1 shows the process for the x86-64 translation, as it searches for the Physical Address (PA) corresponding to a virtual address (VA). In the process, the hardware accesses four page tables *in sequence*: PGD, PUD, PMD, and PTE.

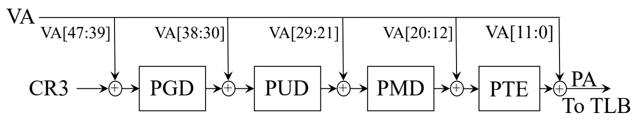


Fig. 1: Address translation in the x86-64 architecture.

Modern processors cache the intermediate page table entries in small Page Walk Caches (PWCs) [2], [9], [12], [13] to avoid accessing memory in each step. However, many emerging applications overflow such caches, introducing multiple sequential memory accesses in the critical path. Furthermore, manufacturers are increasing the number of levels in the translation tree. For example, the new Intel Sunny Cove [40], [41] adds a fifth level to the tree. Unfortunately, as very large memories are likely to appear with the arrival of non-volatile memory, this approach is hardly scalable.

B. Hashed Page Tables

Hashed page tables (HPTs) [18], [24], [26], [33], [36], [37], [39], [44], [45], [77], [79], [83], [88] use a different design. On a TLB miss, address translation consists of hashing the virtual page number (VPN) and using the hash key to index the page table. Assuming that there are no hash collisions, only one memory access is needed for address translation. HPTs have been implemented in the IBM PowerPC [38], HP PA-RISC [37], and Intel Itanium [39] architectures.

Challenges of Hashed Page Tables. HPTs have several limitations that have resulted in industry disfavoring them [9]. One is the loss of spatial locality in the accesses to the HPT. This is caused by hashing, which scatters the HPT entries of contiguous virtual pages. In addition, there is the need to associate a hash tag with each page table entry, which consumes memory space. Most importantly, there are hash collisions, which lead to more memory accesses, as the system walks collision chains [9]. Strategies such as collision chaining [39] and open addressing [88] introduce expensive memory references needed to walk over the colliding entries.

On top of all this, it is not possible to have a straightforward design with a single global HPT that contains page table entries from all the active processes in the machine [24], [88]. The reason is that supporting multiple page sizes (e.g., huge pages) or page sharing between processes requires additional complexity. For example, to support these two features, the IBM PowerPC architecture uses a *two-level translation* procedure for each memory reference [38]. In addition, when a process is killed, the system needs to perform a linear scan of the entire HPT to find and delete the process entries. Sadly, deleting an entry is costly: it may require a long hash table look-up (for open addressing) or a collision chain walk. Further, deleting a page table entry in open addressing may affect the collision probes in future look-ups.

Making Hashed Page Tables Compelling. Recent works have solved some of the HPT limitations. For example, Yaniv and Tsafrir [88] propose an HPT design that uses Page Table Entry Clustering, where multiple contiguous page table entries are placed together in a single HPT entry with a size equal to a cache line. Also, they propose Page Table Entry Compaction, where unused upper bits of multiple contiguous page table entries are repurposed to store the hash tag.

To simplify conflict handling, Skarlatos et al. [77], [79] propose using cuckoo hashing [64] in HPTs. The HPT of a given page size is made *W*-way set-associative, and each way uses a different hash function. To insert an element p, one way is selected and p is inserted in its hashing location. If the selected entry was used, the current occupant q is kicked out, and re-inserted in another way, in q's hashing location there. If that location was used, the ocupant is kicked out and the process repeated. The process may repeat multiple times. By carefully setting the allowed maximum HPT occupancy, one can pick a reasonable maximum number of re-insertion iterations allowed that makes the probability of a final HPT entry eviction very small. In cuckoo hashing, an element lookup needs to check all *W* ways (in parallel).

The design of Skarlatos et al. [77], [79] employs *process-private* HPTs. Rather than using a default per-process HPT size that could exhaust memory in a highly-used machine, the HPT starts small and *dynamically resizes*. The scheme is called Elastic Cuckoo Page Tables (ECPTs). An *upsize* operation is triggered when HPT occupancy reaches a high threshold; a *downsize* operation is triggered when HPT occupancy reaches a low threshold. These upsize/downsize operations do not stop program execution: they are efficiently overlapped with program execution.

Mechanics of an HPT Upsize. When the occupancy of a W-way HPT reaches a high threshold, a new, double-sized W-way HPT is allocated [77]. From then on, every time that the OS is invoked to insert an element into the HPT, the OS uses the opportunity to *rehash* (i.e., move) one element from the old HPT to the new HPT. To perform rehashes efficiently, a *Rehashing Pointer* P_i is added to each way i of the old HPT. In each way, P_i initially points to the HPT base. On a rehash of an element from way i, the OS takes the element pointed to by P_i, inserts it into way i of the new HPT, and increments P_i. At any time, P_i divides way i of the old HPT into two regions: the entries at indices lower than P_i (*Migrated Region*) and those at indices equal or higher than P_i (*Live Region*). As gradual rehashing proceeds, the migrated regions of the ways in the old HPT keep growing. Eventually, when the migrated regions completely cover all the ways, the old HPT is deallocated.

During resizing, the insertion of an element p in a W-way HPT proceeds as follows. The system randomly picks one way from the old HPT and uses its hash function to hash p. If the hash key falls in the live region of the way, the element is inserted in the old HPT; otherwise, p is hashed with the hash function of the same way of the *new* HPT, and the element is inserted in the new HPT. With this design, a lookup operation for an element p during resizing only needs W probes. Indeed, p is hashed using all the hash functions of the *old* HPT ways. For each way i, if the hash key falls in the live region, the *old* HPT way is probed; otherwise, p is hashed with the hash function of the same way in the *new* HPT, and the *new* HPT way is probed.

III. Motivation for Improving HPTs

While the aforementioned advances have made HPTs competitive, HPTs still have an important limitation: they require the allocation of substantial *contiguous* physical memory. This is shown in Table I, which lists some characteristics of the applications we analyze in this work. The applications will be discussed in Section VI.

App.	Data Mem (GB)	Page Table Contig. Mem. (KB)		Page Table Total Memory (MB)			
		No THP		No THP		THP	
		Tree	ECPT	Tree	ECPT	Tree	ECPT
BC	17.3	4	8192	17.12	36.0	16.9	36.0
BFS	9.3	4	16384	28.2	72.0	28.2	72.0
CC	9.3	4	16384	32.0	72.0	28.1	72.0
DC	9.3	4	16384	31.5	72.0	29.3	72.0
DFS	9.0	4	16384	28.1	72.0	28.1	72.0
GUPS	64.0	4	65536	140.0	288.0	0.4	0.8
MUMmer	6.9	4	1024	1.4	4.5	1.2	2.2
PR	9.3	4	16384	28.0	72.0	27.4	72.0
SSSP	9.3	4	16384	28.4	72.0	28.3	72.0
SysBench	64.0	4	65536	140.0	288.0	0.4	0.8
TC	11.9	4	2048	4.1	9.0	3.8	9.0
GeoMean	13.9	4.0	12697.6	23.5	56.0	7.9	18.0

TABLE I: Memory consumption of our applications.

Column 2 shows the maximum memory consumed by the applications' data. Columns 3 and 4 show the maximum *contiguous* memory allocated by the page tables using the radix-tree and ECPT organizations. In the radix-tree organization, the contiguous memory is always 4KB, which is the size of a page. In the ECPT organization, the numbers shown are the maximum size of an HPT way. *Intuitively*, each way of a set-associative HPT needs to be allocated in a contiguous chunk of physical memory. The ECPT numbers correspond to an environment without transparent huge pages (THP) [85] for application data, which is the most unfavorable case. From the table, we see that an HPT way uses 64MB of contiguous memory in two applications.

Allocating large contiguous chunks of memory in a busy server with highly-fragmented memory is expensive. We conduct experiments on a Linux-based server with different fragmentation levels using an open-source fragmentation tool [1]. We measure that, on average, allocating and zeroing out a 4KB, 8KB, 1MB, 8MB, and 64MB chunk takes 4K, 5K, 750K, 13M, and 120M cycles, respectively, at 2 GHz and 0.7 fragmentation (i.e., high) in the FMFI metric [32], [49]. These numbers are consistent with prior measurements [49]. As the chunk size increases, the overhead increases faster. *More importantly*, when we increase the memory fragmentation over 0.7 in the FMFI metric, the system is unable to allocate 64MB of contiguous memory and returns an error. Consequently, the ECPT runs are unable to finish.

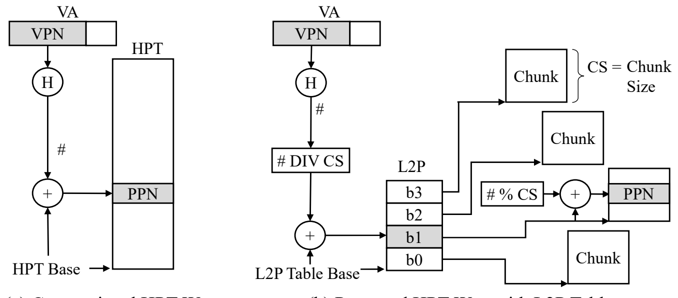
Columns 5-8 show the maximum memory consumed by the page tables using the radix-tree and ECPT organizations without and with THP. We can see that ECPT uses, on average, 138% and 128% more page table memory than radix tree without and with THP, respectively. Compared to the memory consumed by the applications' data, this higher amount of memory consumed by page tables is not significant. However, as we will show later, reducing the total memory consumed by HPTs can also help to minimize their contiguity requirements.

IV. Designing Memory-Efficient HPTs

To solve the HPT contiguity problem, we propose four novel hardware-assisted primitives for memory-efficient HPTs. Two of them directly minimize the contiguous physical memory needed by HPTs: (i) *Logical to Physical (L2P) Table* and (ii) *Dynamically-Changing Chunk Sizes*. The other two indirectly minimize the contiguous physical memory needed by HPTs by reducing the total physical memory needed by HPTs: (i) *In-place Page Table Resizing* and (ii) *Per-way Resizing*.

A. Logical to Physical (L2P) Table

As shown in Figure 2a, a conventional HPT way needs to be allocated in a contiguous memory region. This is because, on a TLB miss, the Virtual Page Number (VPN) is hashed, and the resulting key is added to the base of the HPT to obtain the entry with the corresponding Physical Page Number (PPN). This design does not admit discontiguities in the table—unlike in radix page tables. In our experiments, we find applications whose HPTs need up to 64MB per way. Finding 64MB of contiguous memory is often time-consuming. Further, in a machine with highly-fragmented memory, we find that it causes the program to crash.



(a) Conventional HPT Way (b) Proposed HPT Way with L2P Table

Fig. 2: Reducing the HPT requirements for contiguous memory allocation using the L2P table.

To solve this problem, we propose a design that breaks each way of the HPT into multiple fixed-sized *Chunks* that do not need to be contiguous. Then, we add a small indirection table in the Memory Management Unit (MMU) called the *Logical to Physical* (L2P) table that redirects the accesses to the chunks. The mechanism is transparent to software, and the hardware is highly localized. With a judicious selection of parameters, each chunk requires only a modest amount of contiguous memory, and the L2P table is small enough to have a tolerable access time.

The design is shown in Figure 2b. The HPT way is broken down into a set of chunks of size *ChunkSize* (CS). On a TLB miss, the hash of the VPN is divided by CS and added to the base of the L2P table. The contents of that location (e.g., *b1* in Figure 2b) is a pointer to the base of the chunk that includes the desired PPN. Then, the hash of the VPN modulo CS is added to the base of the chunk to get to the entry with the PPN.

At any time, the MMU only contains the L2P table of the currently-running process. The OS saves and restores the L2P table on context switch. This operation has low overhead. The reason is that, as will see, the L2P table is not very large, and we only need to save and restore the L2P table entries in use—which are, on average, a small fraction of all the entries.

Because CS is a power of two, the division and modulo operations in Figure 2b are *simply a shift and a mask operation*. Consequently, we estimate that an MMU-resident L2P table, as it is accessed in hardware, adds about a couple of cycles to a baseline page walk like in Figure 2a. Such overhead is perfectly tolerable, at least for the Elastic Cuckoo Page Table (ECPT) implementation of HPTs. This is because this overhead can, in most cases, be hidden by overlapping it with the access to the Cuckoo Walk Cache (CWC) of ECPTs. We discuss the details in Section V-D.

B. Dynamically Changing Chunk Sizes

To minimize the L2P table access time, the L2P table has to be small and, therefore, can at most hold pointers to several tens of chunks. As a result, we need to set CS to a value so that all these several tens of chunks combined together are able to hold all the entries of one HPT way.

In practice, applications have widely-varying behavior. On the one hand, big-data applications allocate vast amounts of memory and may require tens or hundreds of MB per HPT way. On the other hand, many system services, functions, and microservices need little memory, and may only need a few KB per HPT way. For the first class of applications to hold all the page mappings in several tens of chunks, each chunk has to be in the MB range. However, the second class of applications would waste substantial memory with large chunks. Note that, in all cases, a chunk is composed of one or multiple contiguous physical pages allocated in one shot.

To address this problem, we propose to *dynamically* change the chunk size assigned to an application, based on the behavior of the application over time. We select a set of chunk sizes that range from small to large. When an application starts, it uses the smallest chunk size. As the application increases its HPT way requirements, it may change its chunk size. With this support, HPTs of both small- and large-memory applications use non-contiguous memory *efficiently*.

Section V lists and justifies the L2P table sizes and different chunk sizes that we use. However, to understand the operation, we now show an example that uses a 64-entry L2P table and 8KB and 1MB chunks. Figure 3a shows an application that starts by needing only 4KB for its HPT way. In this case, the OS uses the small chunk size (8KB) and allocates only one chunk. Only one entry of the L2P table is used, and the HPT way uses half of the 8KB chunk. If the application doubles its HPT, the OS simply fills-up the second half of the chunk, while still using a single entry in the L2P table (Figure 3b).

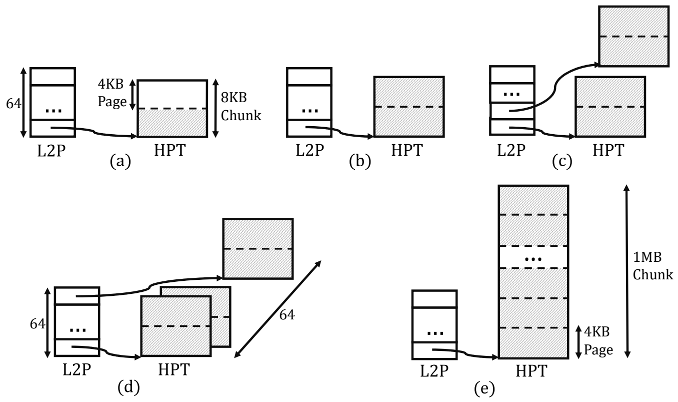


Fig. 3: Dynamically changing chunk sizes for memory efficiency in non-contiguous HPTs. Shaded pages are in use.

If the application further doubles its HPT to 16KB, the OS allocates a second small chunk and uses a second entry in the L2P table (Figure 3c). The process repeats until all 64 entries in the L2P table are used, for a total of a 512KB HPT way (Figure 3d). If another doubling of the HPT occurs, it triggers the transition of chunk size, which involves allocating a new single 1MB chunk, rehashing the entries from the old chunks into the new one, and freeing-up the old chunks. After the resize, we use a single entry of the L2P table, pointing to the 1MB chunk (Figure 3e). Further upsizes allocate more 1MB chunks and hence use more entries of the L2P table. Once all 64 L2P table entries point to full 1MB chunks, another HPT upsize requires the allocation of a chunk of the next size up.

Overall, both small- and large-sized applications use HPT memory efficiently. The hardware uses some bits in the MMU to record which chunk size the HPT is currently using, and whether an upsizing will entail a change in chunk size.

C. *In-place Page Table Resizing*

This technique and the next one reduce the total physical memory needed by HPTs; we will see later that, indirectly, they also minimize the contiguous physical memory needed.

To understand this technique, recall that our state-of-the art baseline scheme (i.e., ECPT) resizes the HPT dynamically while the program runs and that, during resizing, both the old and the new HPT co-exist in memory. This approach can consume sizable memory. For example, some of our applications need an HPT of 192MB, organized in three 64MB ways. Therefore, during resizing to 192MB, the new and old HPT tables consume 192 + 96 = 288MB. In addition, a resizing operation takes time, and during most of a program's execution time, there are two HPTs (old+new) co-existing in memory; we measure that, on average, this is the case for 87.3% of the total execution time. With multiple processes running in the machine, each with one HPT per page size, there may potentially be several HPT resizings occurring concurrently, consuming substantial memory.

The OS needs to rehash all the old HPT entries. However, it rehashes a single entry (or a small group of them) only when a new HPT entry is inserted. The reason for waiting to rehash entries until a new entry is inserted is to reuse the OS invocation that the insertion triggers. Of course, one could allocate another OS thread to perform all the rehashes in the background, but that would add overhead.

To reduce the memory used, in this paper, we propose *In-Place* HPT resizing. The idea is for the new and old HPTs to share the same memory space. This ensures that, at any given time, the memory used by the two HPTs is equal to the bigger of the two, rather than the sum of them. To see it pictorially, Figure 4 shows how to expand an HPT (a) out of place and (b) in place. For simplicity, the picture shows only one way of the HPT. Moreover, it depicts the HPT way as contiguous memory. *In reality*, as per Sections IV-A and IV-B, *the HPT ways (old and new) are composed of a set of non-contiguous chunks*.

To be consistent with the original ECPT paper [77], Figure 4 assumes that addresses increase from the top of the figure to the bottom. Recall that in HPTs undergoing conventional out-of-place resizing (Figure 4a), there are three regions: Migrated and Live regions in the old HPT, and the new HPT. Insertions place entries into the new HPT or into the Live region. Rehashes expand the Migrated region downward by taking entries from the Live region and inserting them into the new HPT. No entry is placed in the Migrated region.

In our proposed in-place resizing (Figure 4b), both insertions and rehashes may place entries into the Migrated region—since it overlaps with the new HPT. To prevent these new entries from causing confusion, we propose a simple

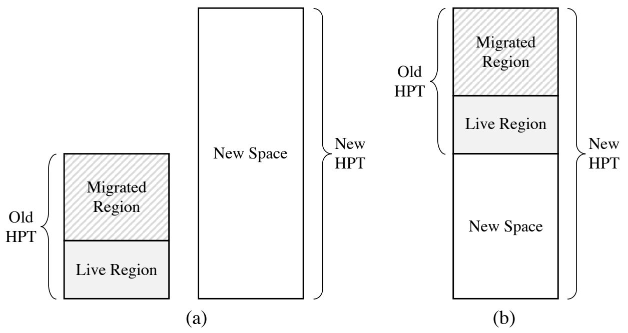


Fig. 4: Out-of-place and (a) in-place (b) resizing of an HPT way. For simplicity, the picture depicts the HPT ways (old and new) as contiguous memory. In reality, they are both composed of a set of non-contiguous chunks.

technique: the hashing function used for the new HPT is kept the same as in the old HPT; however, to index the new HPT, we use one additional bit of the resulting hash key (in an upsize) or one fewer bit (in a downsize). The bit added is the one beyond the most significant bit (MSB) of the hash key; the one removed is the MSB of the hash key. In addition, we only upsize or downsize each way of the HPT to the next higher/lower *power of two*.

Detailed Rehash Algorithm. During an HPT resizing, a rehash operation moves an element from way i of the old HPT (H_i) to the same way in the new HPT (H'_i). The element moved is the one at the top of the Live region of H_i, and is pointed to by the Rehashing pointer of the way (P_i) (Section II-B). After the move, P_i is incremented. To describe the algorithm, we first consider an HPT upsize, and then an HPT downsize.

In an HPT upsize, assume that the entry to rehash is *OldEntry* in Figure 5a. We hash its VPN using the original hash function but use one additional bit of the resulting hash key. At this point, two outcomes are possible: if the additional bit is zero, the entry is kept in place (*NewEntry* in Figure 5b); if it is one, the entry is moved to the second half of the new HPT, at the same offset as in the old HPT (*NewEntry* in Figure 5c).

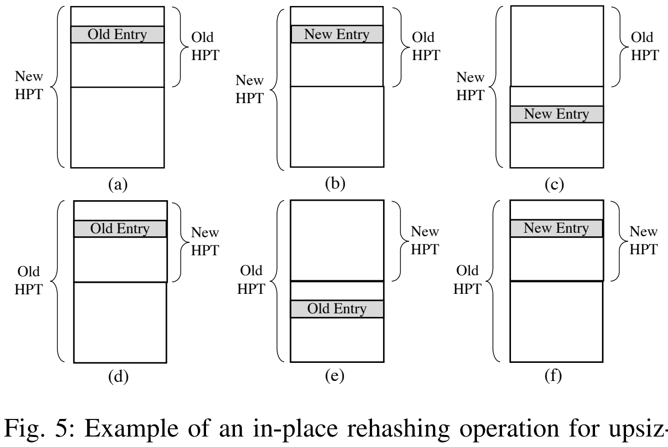


Fig. 5: Example of an in-place rehashing operation for upsizing (a to c) and downsizing (d to f).

In an HPT downsize, the opposite happens: two entries from the old HPT may be rehashed into the same position in the new HPT—and, therefore, one will be cuckooed into a different way. As an example, the two *OldEntry* entries of the old HPT in Figures 5d and 5e will be rehashed into the same position of the new HPT, given by *NewEntry* in Figure 5f.

Other Operations. The other operations performed on the HPTs during resizing use the same algorithm as described by Elastic Cuckoo Hashing [77]. In the following, we assume we are in an HPT upsize. In a lookup, for each way *i*, the hash key created using the old function is compared to the Rehashing Pointer (P_i). If the hash key is higher than or equal to P_i, the old hash key is used to index the HPT; otherwise, the new hash key is used to index the HPT. The new hash key is the same as the old one except that it includes the bit beyond the MSB. Only a single access per HPT way is needed.

A delete follows the lookup procedure and, on finding the element, clears the entry. Finally, an insert involves randomly picking a way i, using the old function to create a hash key, and comparing the hash key to P_i. If the hash key is larger than or equal to P_i, the element is inserted in the entry pointed to by the old hash key; otherwise, it is inserted in the entry pointed to by the new hash key. In case of a conflict, the existing entry is cuckooed into a different way.

Interaction between Resizing and Changing Chunk Sizes. All the upsizing and downsizing HPT operations that do not involve a change in chunk size perform in-place HPT resizing. All the upsizing and downsizing HPT operations that need a change in chunk size require, by construction, out-of-place HPT resizing.

D. Per-way Resizing

State-of-the-art proposals organize the HPT in a set associative manner to reduce conflicts and, on a resize, upsize or downsize all *W* ways. This approach is potentially memory inefficient: the HPT doubles/halves in size every time, while it may be that the HPT only needs a little more/less space.

To reduce waste in HPT memory consumption, we propose *Per-Way Resizing*. The idea is to upsize or downsize only one way at a time. With this proposal, on an upsize, one needs to allocate only *1/W* of the memory that would otherwise be added for the new HPT. As in the previous section, this proposal is applied to HPT ways that are composed of multiple non-contiguous chunks.

While this improvement saves memory, it introduces two new issues. Specifically, we need to determine which way to pick to upsize or downsize while avoiding way imbalance. We also need to determine what algorithm to use to insert an item into the HPT. We consider these issues next.

Deciding Which Way to Upsize or Downsize. In the conventional all-way resizing, the OS keeps a counter with the whole HPT occupancy; when the counter reaches a threshold, the HPT is resized. With per-way resizing, the OS tracks the occupancy of each individual way with per-way counters. When one of the counters reaches a threshold, the corresponding way is resized.

One needs to avoid repeatedly upsizing (or downsizing) the same way at the expense of other ways. To keep the different ways somewhat balanced, we add one additional condition before allowing the resizing of a way. Specifically, the candidate way cannot already be larger than another way (in an upsize) or smaller than another way (in a downsize). With this constraint, a way will never be more than double (or less than half) the size of another way.

Deciding Where to Insert an Element. In a conventional design where all the ways are equally sized, one can randomly choose a way during insertion, and naturally maintain a balanced occupancy of the ways. Such random insertion is not appropriate with per-way resizing: the upsized way would not be utilized according to its full capacity while the other ways would continue to be highly occupied. The result would be frequent conflicts and re-insertions.

To avoid this problem, we propose a *weighted* random insertion algorithm. Specifically, since the OS knows the occupancy and size of each way, it also knows how many free slots each way has. Hence, our algorithm sets the probability of inserting the element in way *i* to be the ratio between the number of free slots in way *i* and the total number of free slots. In other words, we generate a random number between 0 and 1, and give a weight $FREE_i/FREE_{sum}$ to each way i. Moreover, if way i is larger than other ways and its occupancy has already reached the predefined threshold to upsize, we set its weight to zero—effectively preventing insertions in i.

With this algorithm, we get the desired behavior. First, after the upsize of a way, most of the insertions pick that way—delaying the upsize of other ways. Second, once the occupancy of a large way reaches the threshold value, insertions do not pick that way before upsizing smaller ways.

E. Reducing HPT Size Reduces Memory Contiguity

The last two techniques were software strategies to reduce the size of the HPT. However, they can also indirectly reduce the memory contiguity requirements of the HPT. The reason is that a smaller HPT may be able to operate with smaller-sized memory chunks than a larger HPT. For example, we will see in the Evaluation Section that, thanks to these techniques, two of our applications can build their HPTs with 1MB memory chunks rather than requiring chunks of the next larger size.

V. ME-HPT Implementation Aspects

In this section, we discuss several aspects of ME-HPTs: L2P table entry stealing, the chosen chunk sizes, the scalability of L2P tables, and hiding the access to the L2P table.

A. L2P Table Entry Stealing

To minimize access time, we size the L2P table of an application to have 32 entries for each page size supported and HPT way. Recall that we support three page sizes (i.e., 4KB, 2MB, and 1GB). If we assume three HPT ways, then we have nine subtables in the L2P table for an application.

Consider now way i's three subtables for 4KB, 2MB, and 1GB page sizes. It is very unlikely that all three subtables will be highly utilized. Most likely, one or more of them will have few entries. Consequently, we allow the subtable of one page size to steal entries from the subtables of other page sizes if it needs them. The result is a better utilization of the L2P entries.

Figure 6a shows our proposed design. In the MMU, we lay out the three subtables of the same way i contiguously. The 1GB one is placed in the middle, since it is the least likely to be used. The 4KB and 2MB L2P subtables grow in opposite directions. Figure 6a shows the case where the 4KB and 2MB subtables use two entries each.

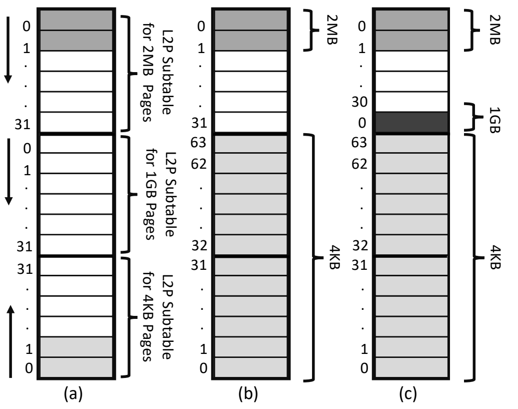

L2P Subtable for 2MB Pages
L2P Subtable for 1GB Pages
L2P Subtable for 4KB Pages
2MB
1GB
4KB
(a)
(b)
(c)

Fig. 6: Stealing entries across the L2P subtables of different page sizes and same way. The non-white entries are in use.

Assume now that the 4KB subtable uses all its 32 entries and needs to be upsized. If no entry in the 1GB subtable is in use, we let the 4KB subtable take all the entries from the 1GB subtable, and grow to 64 entries (Figure 6b). The chunk size is kept the same. If later, the 4KB subtable wants to upsize again, it can only do it by allocating a chunk of the next bigger size and rehashing all its entries there, as we saw before.

If, after the 4KB subtable takes all the entries from the 1GB subtable, an entry is needed in the 1GB subtable, the system uses the most significant entry of the 2MB subtable (Figure 6c).

B. Chosen Chunk Sizes

With this support, we choose the sizes of chunks to be, from smaller to larger, 8KB, 1MB, 8MB, and 64MB—although, for our applications, we only need 8KB and 1MB chunks. Specifically, let us we start with 8KB, the smallest chunk size. When one of the subtables, using 8KB chunks, reaches 64 entries (Figure 3d) its HPT way extends over 8KB $\times$ 64 = 512KB. If the HPT way now needs to be upsized, the OS changes the chunk size to 1MB, allocates a single chunk, and uses a single entry in the L2P table (Figure 3e).

If the application fills up all 64 entries with 1MB chunks, it uses 64MB per way. If the HPT way now needs to be upsized to 128MB, the OS uses the next chunk size. Since this case is rare and we do not want to allocate large contiguous chunks, we set the next chunk size to 8MB. In this case, the OS allocates 16 chunks and uses 16 entries in the L2P table. If the application fills up all 64 entries and the HPT way needs to be upsized, the next chunk size (very rarely used) is 64MB.

The size of the L2P table of an application is modest. Consider the worst-case scenario of using 8KB chunks. With a physical address of 46 bits, the base address of an 8KB chunk is 33 bits followed by 13 zeros. We only need to store the 33 bits. Then, the overall size of the L2P table is 32 entries $\times$ 3 ways $\times$ 3 page sizes $\times$ 33 bits = 1.16KB.

Note that the chunk sizes do not need to be limited to those listed above. Instead, the OS could make per-process and per-system state decisions. To determine the next chunk size, the OS could dynamically use heuristics based on the current level of fragmentation and the expected final HPT way size. We consider this topic future work.

C. Scalability of L2P Tables with Changing Chunks

With L2P tables and changing chunk sizes, ME-HPT provides a scalable solution for HPTs. As indicated above, the MMU contains the L2P table of only the running process. Including all the page sizes and ways, the L2P table for a process has 288 entries and uses 1.16KB of memory. In a context switch, the OS only saves and restores the valid entries in the L2P table, which are clustered at the extremes of the table. Applications typically use only a few entries of the L2P table. Section VII-E4 will show that, on average, they only use 53 entries. Hence, the overhead of saving and restoring the L2P table is modest. In addition, in a virtualized system, the overhead is even smaller for two reasons. First, there are no guest L2P tables because guest HPTs are not contiguous, as they are spread in host pages. Second, on a guest context switch, the host L2P table is not saved or restored.

For the HPT, the largest contiguous memory needed is that of one chunk (i.e., 8KB or 1MB in our applications). Table II shows, for the different chunk sizes, what is the maximum size of the HPT way that can be created and, for the resulting total HPT (i.e., all three ways), what is the maximum size of the physical memory space that can be mapped with 4KB pages and with huge (2MB) pages. We see that, with 8KB chunks, ME-HPT can build a 64x8KB=512KB HPT way. If the HPT is for 4KB pages, the resulting 3-way HPT can map 768MB of application data; if the HPT is for 2MB pages, the resulting 3-way HPT can map 384GB of application data.

Chunk Size	Max HPT Way Size	Max Total HPT Mapping Space with 4KB Pages	Max Total HPT Mapping Space with 2MB Pages
8KB	512KB	768MB	384GB
1MB	64MB	96GB	48TB
8MB	512MB	768GB	384TB
64MB	4GB	6TB	3PB

TABLE II: Maximum HPT way sizes and maximum total HPT mapping space for different chunk sizes. For our applications, we only need 8KB and 1MB chunk sizes.

With 1MB chunks, ME-HPT can build a 64x1MB=64MB HPT way. As before, if the HPT is for 4KB pages, the 3-way HPT can map 96GB of application data; if the HPT is for huge pages, the 3-way HPT can map 48TB of data.

With 8MB chunks, ME-HPT can build a 64x8MB=512MB HPT way. If the HPT is for 4KB pages, the 3-way HPT can map 768GB of application data; if the HPT is for huge pages, the resulting HPT can map 384TB of application data.

If the application is even bigger, we can switch to 64MB chunks. In this case, the ME-HPT can build a 64x64MB=4GB HPT way. If the HPT is for 4KB pages, the 3-way HPT can map 6TB of application data; if the HPT is for huge pages, the resulting HPT can map 3PB of application data.

When the OS upsizes an HPT way, it allocates one or multiple additional chunks. These chunks are *neither contiguous with each other nor contiguous with the chunks of the current HPT*. An HPT way is always a collection of non-contiguous chunks. While, for our applications, the chunk size is 8KB or 1MB, larger applications may need larger chunks. To find space for large chunks in a highly-fragmented machine, the OS may perform memory compaction or swap-out pages, as is ordinarily done to allocate huge pages. An upsizing cannot fail unless it requests a chunk of a size so big that the OS is unable to provide so much contiguous memory.

D. Hiding the Access to the L2P Table

The extra latency added by the access to the L2P table (Section IV-A) does not noticeably slow down a page walk in an ECPT design. The reason is that this extra latency can be overlapped with the access to the Cuckoo Walk Cache (CWC) hardware structure in ECPTs.

Figure 7 shows the design. On a TLB miss, the translation for the missing virtual address can be present in any of the ways of the HPT of any page size. To trim the number of memory locations to check, the ECPT hardware first accesses the CWC hardware caches. The outcome determines which way of the HPT of which page size should be accessed. In parallel, if a resize is in progress, the rehash pointers are checked to decide whether the old or the new HPT should be accessed.

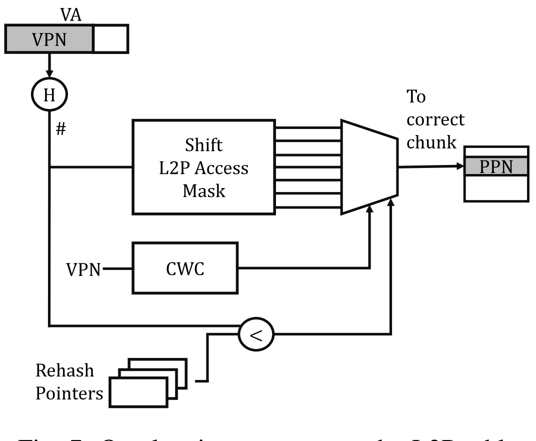


Fig. 7: Overlapping accesses to the L2P table.

In the meantime, we propose that the hardware accesses the L2P table of the process in the MMU, and generates the potential memory addresses to access. After the CWC and rehash pointer checks finish, they select which memory access(es) to perform. Hence, the latency of the L2P table access is hidden.

The only time when the L2P table access is not hidden is in a cuckoo re-insertion due to HPT conflicts. In this case, the CWC is not accessed, and the latency of the L2P table access is on the critical path. Fortunately, in this case, the few-cycle access latency is noise. The reason is that an element insertion or cuckoo rehash is performed by the OS rather than by hardware, and invoking the OS has much higher overhead.

VI. Experimental Methodology

Modeled Architectures. We use full-system cycle-level simulations to model a server architecture with 8 cores and 64 GB of main memory. Our Baseline architecture models Elastic Cuckoo Page Tables (ECPTs) [77]. We enhance the baseline architecture with our design (ME-HPT). We compare the two architectures (i) with only 4KB pages and (ii) with multiple page sizes by enabling Transparent Huge Pages (THP) in the Linux kernel [85]. We use 3-way HPTs. We also model a system with state-of-the-art radix-tree page tables. Table III shows the architectural parameters. Recall that PTE, PMD, and PUD are the names of the structures associated with pages of size 4KB, 2MB, and 1GB, respectively. An HPT is upsized when its occupancy gets above 0.6, and downsized when it drops below 0.2. An HPT way always starts with 8KB.

Processor Parameters	
Processor	8 OoO cores, 256-entry ROB, 2GHz
Private L1D+L1I caches	32KB, 8-way, 2 cycles RT, 64B line
Private L2 cache	512KB, 8-way, 16 cycles RT
Shared L3 cache	2MB per core, 16-way, 56 avg cyc RT
L1 DTLB (4KB pages)	64 entries, 4-way, 2 cycles RT
L1 DTLB (2MB pages)	32 entries, 4-way, 2 cycles RT
L1 DTLB (1GB pages)	4 entries, 2 cycles RT
L2 DTLB (4KB pages)	1024 entries, 12-way, 12 cycles RT
L2 DTLB (2MB pages)	1024 entries, 12-way, 12 cycles RT
L2 DTLB (1GB pages)	16 entries, 4-way, 12 cycles RT
# of page walkers	4 (both in ME-HPT and radix tree)
PWC for radix tree	3 levels, 32 entries/level, 4 cycles RT, fully associative, 0.75KB total size
Memory Parameters	
Capacity; #Channels; #Banks	64GB; 4; 8
Memory access latency	200 cycles RT (Average)
Frequency; Data rate	1GHz; DDR
ME-HPT Parameters	
Initial HPT for each page size	128 entries $\times$ 3 ways each
Initial PMD/PUD CWT [77]	128 entries $\times$ 2 ways each
PMD-CWC; PUD-CWC [77]	16 entr, 4 cyc RT; 2 entr, 4 cyc RT
Hash functions: CRC	Latency: 2 cyc; Area: $1.9*10^{-3}mm^2$
L2P table size	32entr x 3ways x 3pagesizes (1.16KB)
Shift + L2P access + Mask	4-cycle latency
Chunk sizes	Used: 8KB,1MB; Unused: 8MB,64MB
HPT occupancy thresholds	0.6 for upsize, 0.2 for downsize
Memory fragmentation	0.7 FMFI

TABLE III: Architectural parameters used in the evaluation. RT stands for round trip from the core.

Modeling Infrastructure. We integrate the Simics [58] full-system simulator with the SST framework [7], [72] and the DRAMSim2 [74] memory simulator. We use Intel SAE [17] on Simics for OS instrumentation. We model and evaluate the hardware components of ME-HPT in detail using SST.

Simics intercepts on-the-fly all the instructions executed and all the virtual memory operations performed. These operations are then processed by our back-end cycle-level processor/memory simulator modeling ME-HPTs. Simics provides the actual page table and memory contents for every memory address. With this infrastructure, we are able to implement any page table organization in the simulator, while using the actual page table entry (PTE) contents. Hence, the sequence of PTEs used in the simulated ME-HPT and the radix-tree page tables is the same, although the page table organizations and memory allocations are different. We account for all the HPT overheads. For the allocation overheads, we use real system measurements on a system fragmented with an open-source tool [1]. In the evaluation, we use a high memory fragmentation of 0.7 FMFI [49].

Applications. We evaluate eleven applications from domains ranging from graph processing to high-performance computing (HPC), bioinformatics, and systems. We evaluate eight graph-processing applications from the GraphBIG benchmark suite [60] with input graphs of 1M nodes. They are: Betweenness Centrality (BC), Breadth-First Search (BFS), Connected Components (CC), Degree Centrality (DC), Depth-First Search (DFS), PageRank (PR), Shortest Path (SSSP), and Triangle Count (TC). In the HPC domain, we use GUPS from the HPC Challenge benchmark [57]. GUPS is a random access benchmark that measures the rate of integer random memory updates. In the bioinformatics domain, we use MUMmer from the BioBench suite [4], which performs genome alignment. Lastly, we select the Memory benchmark from the SysBench suite [81], which stresses the memory subsystem. For each application, we measure the first 550M instructions per thread.

VII. Evaluation

A. Memory Contiguity Savings

Figure 8 shows the maximum size of the contiguous memory allocated by the baseline ECPT and by ME-HPT. The data corresponds to the HPTs for 4KB pages, which is the worst case. For each application, the figure shows bars for ECPT, ECPT with THP, ME-HPT, and ME-HPT with THP.

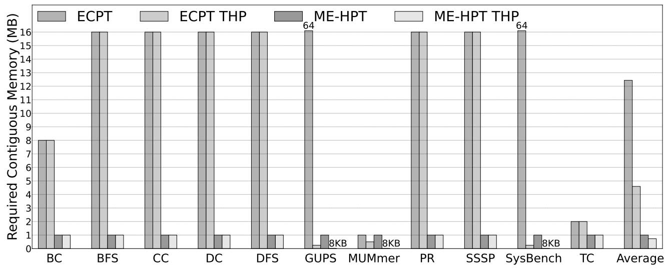


Fig. 8: Maximum size of the contiguous memory allocated for the HPTs for 4KB pages, without and with THP.

With ECPT, one needs to allocate a large contiguous region, equal to the size of one HPT way. This size is 16MB for most graph applications, and 64MB for GUPS and SysBench without THP.

ME-HPT needs much less contiguous memory allocation. Thanks to the L2P table and dynamically-changing chunk sizes, the maximum contiguous allocation size is equal to either 8KB or 1MB—which are the only two chunk sizes needed in the evaluated applications. In most cases, ME-HPTs end up using 1MB chunks. As shown in Figure 8, on average, ME-HPT reduces the maximum size of the contiguous memory allocated by 92% and 84% without and with THP, respectively. For the two most demanding workloads (GUPS and SysBench), the contiguous memory requirements decrease from 64MB to 1MB.

B. Application Performance

Figure 9 shows the speedup of ME-HPT, ECPT, and Radix (with and without THP) over Radix without THP. ME-HPT delivers 1.09x and 1.06x average speed-ups over ECPT without and with THP, respectively; ME-HPT delivers 1.23x and 1.28x average speed-ups over a system with radix-tree page tables, without and with THP, respectively. ME-HPT is faster than ECTP in practically all the cases. The speed-ups result from the lower cost of memory allocation in ME-HPT. Indeed, ME-HPT only allocates chunks of size 8KB and 1MB, while ECPT allocates contiguous regions of up to 64MB.

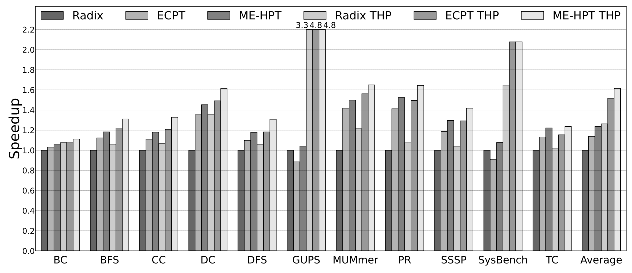


Fig. 9: Speedup of ME-HPT, ECPT, and Radix, without and with THP, over Radix without THP.

If we increase the memory fragmentation over 0.7 FMFI, the system is unable to allocate 64MB of contiguous memory and returns an error. Consequently, the ECPT runs do not finish for the GUPS and SysBench applications.

C. Memory Savings

Figure 10 shows, for each application, the reduction in page table memory by ME-HPT over the ECPT baseline. For each application, we show bars without and with THP. Each bar is divided into the contributions of our two techniques: in-place resizing (Section IV-C) and per-way resizing (Section IV-D). In addition, the numbers on top of the bars are the absolute memory reductions in Mbytes.

On average, ME-HPT saves 43% and 41% of the page table memory used by the baseline ECPT design, without and with THP, respectively. The numbers on top of the bars show that the average memory savings per application are 37MB and

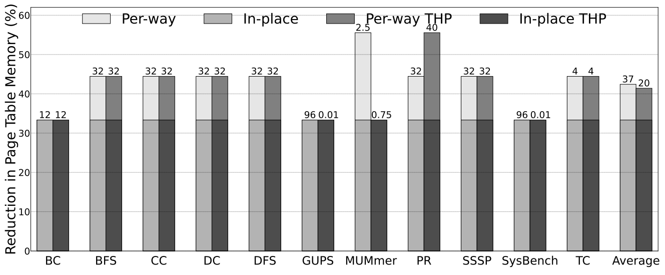


Fig. 10: Reduction in page table memory attained by ME-HPT over the ECPT baseline. The number on top of each bar is the absolute reduction in Mbytes.

20MB, without and with THP. Further, both in-place resizing and per-way resizing contribute to the savings. On average, in-place resizing contributes with 75–80% of the savings, while per-way resizing contributes with 25–20% of the savings.

D. Why Reducing the Page Table Size Matters

The reduction in page table size attained by ME-HPT may look unremarkable in terms of absolute Mbytes. However, such reduction helps decrease the maximum size of the contiguous memory allocation needed for the page tables. The reason is that it helps the L2P table work better and enables the use of smaller contiguous chunks.

Specifically, Section VII-E4 will show that, under ME-HPT without THP, GUPS and SysBench use 192 L2P entries each. Each of these entries points to a 1 MB chunk, as can be deduced from Figure 8. Without the two optimizations that reduce HPT size (in-place resizing and per-way resizing), the page tables of these applications would need 96 Mbytes more memory (Figure 10). That would bring the total number of L2P entries needed to 192 + 96 = 288 entries. Unfortunately, this amount would overflow the capacity of the L2P. As shown in Figure 6, L2P's capacity for a given page size is 192 entries—i.e., 64 entries per page size (thanks to stealing) times 3 ways. As a result, we would have to use the next bigger chunk size. This would require the allocation of larger contiguous memory chunks—in our case, 8MB chunks as shown in Table III. Overall, thanks to the two optimizations that reduce HPT size, the maximum contiguous memory allocation needed by GUPS and SysBench reduces from 8MB to 1MB.

Note that although Table I shows that the maximum contiguous allocation for GUPS and Sysbench under ECPT no THP is 64MB, these applications need, per way, a total of 64MB + 32MB = 96MB of page table memory under ECPT when resizing happens. This is why, without our optimizations, an L2P pointing to 64 1MB chunks cannot map the whole HPT way.

It can be shown that, for the MUMmer application under ME-HPT without THP, with our reduced total HPT size, the L2P table points to multiple 8KB chunks in two ways, and to one 1MB chunk in the other way. Without our space-reducing optimizations, all ways point to two 1MB chunks. Again, the L2P table works better with our reduced total HPT size.

E. ME-HPT Characterization

We now characterize different aspects of ME-HPT.

1) Resizing Operations: Figure 11 shows the number of upsizing operations per way in ME-HPT per application for 4KB pages, without and with THP. Each way starts with 8KB, and every upsizing doubles the size of one way only. There are no downsizes. On average, ways 0, 1, and 2 are upsized 10.6, 10.5, and 9.9 times without THP. Our algorithm load-balances the upsizes across the ways. The highest number of upsizes per way is 13 for GUPS and SysBench without THP. GUPS and SysBench with THP never upsize their 4KB ME-HPTs.

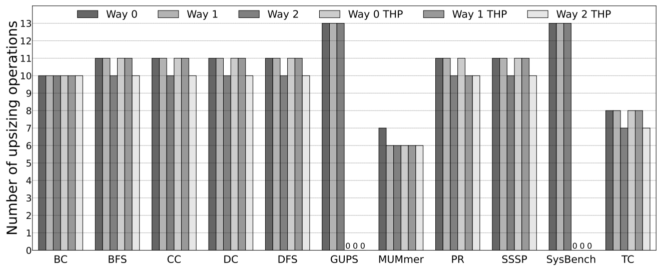


Fig. 11: Number of upsizing operations per way in the ME-HPT for 4KB pages, without and with THP.

We also characterize the ME-HPTs for 2MB pages. The upsizing operations occur only in GUPS and SysBench with THP. They both have 5 upsizing events per way. There is no upsizing event for the ME-HPTs with 1GB pages.

Most of the upsizes happen during the warm-up phase. During steady state, we observe, on average, only 1.8 and 1.6 upsizes per application without and with THP, respectively. For all the applications, there is at most one chunk size switch (from 8KB to 1MB) throughout the whole execution, and thus only one out-of-place resize.

2) Effectiveness of Per-Way Resizing: Figure 12 shows the final size of each of the ways of ME-HPT for 4KB pages. Not all the ways of ME-HPT have the same size. This confirms the effectiveness of per-way resizing.

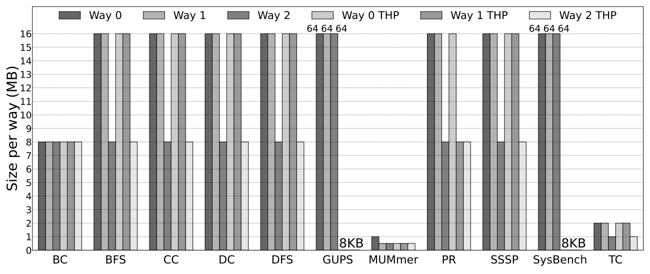


Fig. 12: Size of each of the ways of ME-HPT for 4KB pages.

Without THP, GUPS and SysBench allocate the largest ME-HPTs: 64MB per way. On the other hand, some applications require substantially smaller ME-HPTs—e.g., 0.5MB for most of the ways in MUMmer. We also observe that enabling THP does not necessarily lead to smaller ME-HPTs. The reason is that, even with THP, some applications do not use huge pages, and most of the ME-HPT entries for 4KB pages are still used.

For the applications that heavily utilize huge pages with THP (GUPS and SysBench), the size of ME-HPTs for 4KB pages retains the initial, smallest size (8KB).

3) Data Movement Reduction: Intuitively, ME-HPT's in-place resizing (Section IV-C) should reduce the number of page table entries moved in an HPT upsize by 50% for each HPT way. This is because about 50% of the entries should remain in place. Figure 13 shows the actual average fraction of page table entries moved in an upsize of the 4KB page tables for each application. As shown in the figure, on average, the measured fraction is close to the expected 0.5. GUPS and SysBench with THP do not have upsizings for the 4KB page tables, and are not included in the average. Hence, in-place resizing provides savings in data movement and reduces the number of memory accesses. This results in improved bandwidth and reduced energy consumption.

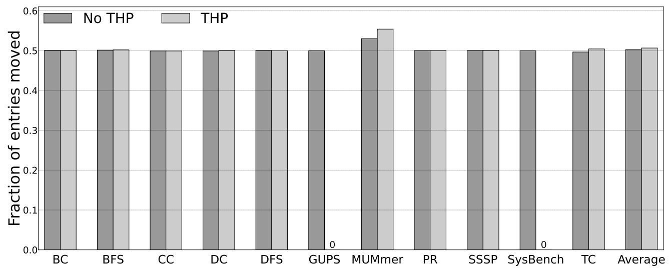


Fig. 13: Fraction of page table entries moved in an upsize of the 4KB page tables with ME-HPT.

4) Number of L2P Table Entries Used: The L2P table has 32 entries $\times$ 3 page sizes $\times$ 3 ways = 288 entries for all page sizes (Section V-A). Most applications use only a small fraction of these entries. Figure 14 shows the actual number of entries that each application uses, without and with THP. The number ranges from 11 (TC) to 195 (MUMmer). On average, only 52.5 entries are utilized.

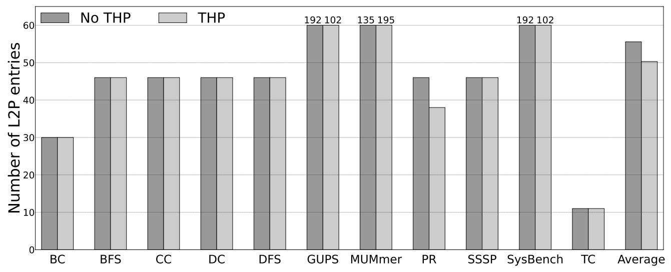


Fig. 14: Number of L2P table entries used per application.

5) Benefits for Small-Size Applications: For applications that only need a small page table, ME-HPT's use of variable-sized chunks is very beneficial. To see this effect, we consider our eight graph applications and reduce their input graph to 1K, 10K, and 100K nodes—rather than the standard 1000K nodes. Figure 15 compares the size of an HPT way for 4KB pages for two ME-HPT designs: one that uses only 1MB chunks (ME-HPT$_{1MB}$) and the default one that uses both 8KB and 1MB chunks (ME-HPT$_{1MB+8KB}$). The data corresponds to the average of all the graph applications. Note that using THP or not does not make a difference for these graph applications due to their irregular memory access patterns.

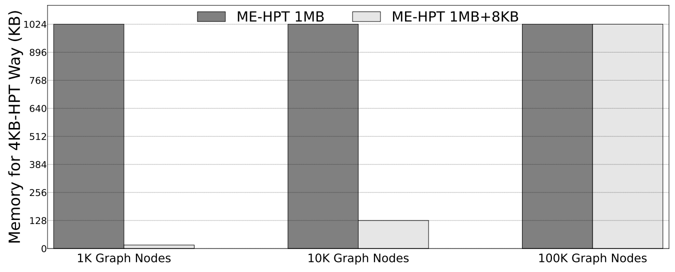


Fig. 15: Average size of an HPT way for 4KB pages for: ME-HPT with only 1MB chunks (ME-HPT$_{1MB}$) and ME-HPT with both 8KB and 1MB chunks (ME-HPT$_{1MB+8KB}$).

For graphs with 100K nodes, the average HPT way needs nearly 1MB and, therefore, both designs use the same memory. However, for graphs with 10K and 1K nodes, the average HPT way only needs about 128KB and 16KB, respectively. As a result, the ME-HPT$_{1MB+8KB}$ design can use 8KB chunks and only allocate as much memory as it is needed. On the other hand, the ME-HPT$_{1MB}$ can only allocate 1MB at a time, wasting substantial memory and reducing performance.

6) Cuckoo Re-Insertions: Figure 16 shows, for our default ME-HPT design, the distribution of the average number of cuckoo re-insertions due to conflicts after an insertion or a reshash. We can see that, with 0.64 probability, no re-insertion is necessary. On average, ME-HPT requires 0.7 re-insertions per insertion or reshash. Even if ME-HPT cannot completely hide the few cycles of an L2P table access in a re-insertion, the aggregate overhead visible to the application is negligible.

Fig. 16: Number of cuckoo re-insertions per ME-HPT insertion or reshash.

VIII. Application of Design to Other Problems

The hashing optimizations introduced in this paper are generically applicable to many of today's hash table designs and use cases, beyond HPTs. They are applicable to various multi-way and set-associative hash table designs, and can benefit use cases where resizing is beneficial and memory efficiency, in terms of contiguity or total memory, is desired.

Scalable Secure Directories. Directories are set-associative hardware structures for cache coherence [25]. Hash-based directory design has been proposed to develop scalable, secure directories with high effective associativity [28], [87]. For example, SecDir [87] proposes per-core private directories using cuckoo hashing, in a similar vein as per-process HPTs. Efficient resizing techniques are useful for hash-based directory designs. Our in-place resizing and per-way resizing techniques can be directly applied to directory designs. Further, directories can be disaggregated with one level of indirection using our L2P table technique. The latency of the L2P table access may potentially be hidden with the access to a large last-level cache.

Memory Indexing. Hash tables are commonly used to implement the memory indices of databases, file systems, and other storage systems for both DRAM and persistent memory [22], [23], [50], [56], [61], [63], [86], [91]. Dynamic resizing is a key operation for memory indices, whose sizes need to be constantly adjusted based on the workloads. Most of the existing hash-based memory indices perform out-of-place resizing, which is expensive in both memory use and data movement. The ME-HPT design supports in-place resizing and can be applied to memory indices to reduce contiguity requirements. Recently, a scheme called Level Hashing [20], [91] has been proposed that supports in-place resizing for persistent memory indices. We compare ME-HPT hashing to level hashing in Section IX.

Key-Value Stores. Hash tables are the key building blocks of key-value stores, which essentially provide a hash-table interface [6], [8], [16], [27], [43], [51], [53], [54], [80], [86], [89]. The ideas developed in ME-HPTs can be applied to many existing key-value stores, which require dynamic resizing—in many use cases, one cannot know the proper size of the key-value store in advance, since no fixed-size suits all system configurations and workloads, and the workloads may change at runtime. Recently, hardware-assisted key-value stores have been designed to further accelerate key-value store performance [48], [52], [55], [89], [90]. The ME-HPT designs can be integrated in hardware for key-value stores to improve their memory efficiency.

IX. Related Work

TLB Design. A large body of research has focused on improving TLB design [10], [14], [19], [21], [47], [68]–[70], [78]. Pham et al. [69] exploit the fact that contiguous virtual pages can be mapped to contiguous physical pages and, thus, TLB reach can by increased by coalescing adjacent entries. Guided by the same motivation, Karakostas et al. [47] introduced Redundant Memory Mappings to translate a range of virtually and physically contiguous pages with a single range table entry. As shown in this paper, creating very large contiguous physical address spaces in fragmented servers is hard.

Researchers also proposed schemes that use large in-memory TLBs, such as POM-TLB [75] and CSALT [59]. These schemes can eliminate a significant portion of page table walks. However, their effectiveness depends on the prediction accuracy. Also, an L2 TLB miss may be propagated to the in-DRAM TLB and, on a miss, still require a page walk.

Huge Pages. Huge pages can reduce the penalty of expensive page walks by reducing the number of page table levels and increasing TLB reach. However, the overheads of huge pages often exceed their benefits. Hence, researchers have proposed numerous schemes to improve the efficiency of huge pages [21], [30], [31], [34], [49], [62], [65]–[67], [71], [73], [82], [84]. Cox et al. [21] proposed MIX TLBs that concurrently support all page sizes by exploiting huge page allocation patterns. Panwar et al. [65] presented HawkEye, an OS support to reduce kernel overheads of huge pages. Guo et al. [34] developed a management policy to break a large page into many smaller ones and to combine small pages into a large page based on the available memory and page access patterns.

Other. To improve page walk performance, researchers have used standard architecture techniques such as prefetching [15], [46], [76] and caching [9], [12], [13]. Bhattacharjee et al. [15] developed Inter-Core Cooperative TLB prefetchers to exploit common TLB miss patterns among cores. Barr et al. [9] presented an extensive design space of page walk cache organizations. The DIY architecture [3] asked applications to manage address translation. Other work proposed direct segments [11], [29] and devirtualized memory [35].

Level Hashing. To our knowledge, Level Hashing [20], [91] is the only hashing scheme that supports a form of in-place resizing. However, it is specialized for non-volatile memory and, therefore, focuses on optimizing writes. It trades more memory accesses (4 per lookup) for less entry moves during resizing (only 1/3 of the old table entries are moved). In comparison, our in-place resizing reduces the moves to 50% with no additional memory references per lookup. Importantly, our algorithm is not specific to write-intensive workloads and is generically applicable to many existing hash table designs. In particular, it applies to page tables, which should be optimized for read-intensive accesses. Moreover, unlike level hashing, in an upsize, ME-HPT does not need to de-allocate part of the old table—the old table becomes a part of the new table. De-allocations tend to cause fragmentation.

X. Conclusion

A limitation of HPTs is their apparent need for substantial contiguous physical memory. This paper addressed this limitation with *Memory Efficient HPTs* (ME-HPTs), which introduces four new techniques that, directly or indirectly, minimize the contiguous physical memory needed by HPTs. These techniques are the L2P table, dynamically-changing chunk sizes, in-place HPT resizing and per-way resizing. Compared to state-of-the-art HPTs, ME-HPTs: (i) reduced the contiguous memory allocation needs by 92% on average, and (ii) improved the performance by 8.9% on average. For two workloads, the contiguous memory needs decreased from 64MB to 1MB. Also, compared to state-of-the-art radix-tree page tables, ME-HPTs sped-up the workloads by an average of 1.23× (without huge pages) and 1.28× (with huge pages).

Acknowledgments

This work was funded in part by NSF Grant No. CNS-1956007, CNS-2107307, and by Intel Corp. under the Transformative Server Architectures (TSA) Center.

References

[1] "Memory Fragmentation Tool," https://github.com/x-y-z/fragm.

[2] J. Ahn, S. Jin, and J. Huh, "Revisiting Hardware-assisted Page Walks for Virtualized Systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*, 2012.

[3] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-It-Yourself Virtual Memory Translation," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, 2017.

[4] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, 2005.

[5] AMD, "Architecture Programmer's Manual (Volume 2)," https://www.amd.com/system/files/TechDocs/24593.pdf, 2019.

[6] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and Large CAMs for High Performance Data-Intensive Networked Systems," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*, 2010.

[7] A. Awad, S. D. Hammond, G. R. Voskuilen, and R. J. Hoekstra, "Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework," Sandia National Laboratories, Tech. Rep. SAND2017-0002, 2017.

[8] A. Badam, K. Park, V. S. Pai, and L. L. Peterson, "HashCache: Cache Storage for the Next Billion," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

[9] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," in *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*, 2010.

[10] 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 (ISCA'11)*, 2011.

[11] 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 (ISCA'13)*, 2013.

[12] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, 2008.

[13] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013.

[14] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, 2011.

[15] A. Bhattacharjee and M. Martonosi, "Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*, 2010.

[16] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, "Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing," in *Proceedings of the 2016 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'16)*, 2016.

[17] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and Analysis Engine for Scale-Out Workloads," in *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*, 2016.

[18] A. Chang and M. F. Mergen, "801 storage: Architecture and programming," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, 1988. [Online]. Available: https://doi.org/10.1145/35037.42270

[19] J. B. Chen, A. Borg, and N. P. Jouppi, "A Simulation Based Study of TLB Performance," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, 1992.

[20] Z. Chen, Y. Hua, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/chen

[21] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017.

[22] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, 2015. [Online]. Available: https://doi.org/10.1145/2694344.2694359

[23] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting Hash Table Design for Phase Change Memory," in *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW'15)*, 2015.

[24] C. Dougan, P. Mackerras, and V. Yodaiken, "Optimizing the Idle Task and Other MMU Tricks," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, 1999.

[25] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*. Cambridge University Press, 2012.

[26] S. Eranian and D. Mosberger, "The Linux/ia64 Project: Kernel Design and Status Update," HP Labs, Tech. Rep. HPL-2000-85, 2000.

[27] B. Fan, M. Kaminsky, and D. G. Andersen, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.

[28] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

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

[30] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large Pages May Be Harmful on NUMA Systems," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*, 2014.

[31] M. Gorman and P. Healy, "Performance Characteristics of Explicit Superpage Support," in *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*, 2010.

[32] M. Gorman and A. Whitcroft, "The what, the why and the where to of anti-fragmentation," in *Linux Symposium*, 2005.

[33] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser, "Itanium — A System Implementor's Tale," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*, 2005.

[34] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, "Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi," in *Proceedings of the 11th ACM International Conference on Virtual Execution Environments (VEE'15)*, 2015.

[35] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing Memory in Heterogeneous Systems," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, 2018.

[36] G. Hoang, C. Bae, J. Lange, L. Zhang, P. Dinda, and R. Joseph, "A case for alternative nested paging models for virtualized systems," *Computer Architecture Letters*, vol. 9, 2010.

[37] J. Huck and J. Hays, "Architectural Support for Translation Table Management in Large Address Space Machines," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, 1993.

[38] IBM, "PowerPC Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors," https://wiki.alcf.anl.gov/images/f/fb/PowerPC_-_Assembly_-_IBM_Programming_Environment_2.3.pdf, 2005.

[39] Intel, "Itanium Architecture Software Developer's Manual (Volume 2)," https://www.intel.com/content/www/us/en/products/docs/processors/itanium/itanium-architecture-vol-1-2-3-4-reference-set-manual.html, 2010.

[40] Intel, "5-Level Paging and 5-Level EPT (White Paper)," https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2015.

[41] Intel, "Sunny Cove Microarchitecture," https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove, 2018.

[42] Intel, "64 and IA-32 Architectures Software Developer's Manual," 2019.

[43] Z. István, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10GBPS key-value stores on FPGAS," in *23rd International Conference on Field programmable Logic and Applications*, 2013.

[44] B. L. Jacob and T. N. Mudge, “A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations,” in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, 1998.

[45] J. Jann, P. Mackerras, J. Ludden, M. Gschwind, W. Ouren, S. Jacobs, B. F. Veale, and D. Edelsohn, “IBM POWER9 system software,” *IBM Journal of Research and Development*, vol. 62, no. 4/5, 2018.

[46] G. B. Kandiraju and A. Sivasubramaniam, “Going the Distance for TLB Prefetching: An Application-driven Study,” in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA’02)*, 2002.

[47] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, “Redundant Memory Mappings for Fast Access to Large Memories,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA’15)*, 2015.

[48] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, “Meet the Walkers: Accelerating Index Traversals for in-Memory Databases,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013. [Online]. Available: https://doi.org/10.1145/2540708.2540748

[49] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, “Coordinated and Efficient Huge Page Management with Ingens,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI’16)*, 2016.

[50] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, “RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP’19)*, 2019. [Online]. Available: https://doi.org/10.1145/3341301.3359635

[51] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, “KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC,” in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP’17)*, 2017.

[52] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey, “Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-Value Store Server Platform,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA’15)*, 2015.

[53] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, “Be Fast, Cheap and in Control with SwitchKV,” in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI’16)*, 2016.

[54] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, “SILT: A Memory-Efficient, High-Performance Key-Value Store,” in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP’11)*, 2011.

[55] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, “Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached,” in *Proceedings of the 2013 International Conference on Computer Architecture (ISCA’13)*, 2013.

[56] B. Lu, X. Hao, T. Wang, and E. Lo, “Dash: Scalable Hashing on Persistent Memory,” *Proceedings of the VLDB Endowment*, vol. 13, no. 8, 2020. [Online]. Available: http://dx.doi.org/10.14778/3389133.3389134

[57] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, “The HPC Challenge (HPCC) Benchmark Suite,” in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC’06)*, 2006.

[58] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A Full System Simulation Platform,” *IEEE Computer*, 2002.

[59] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, “CSALT: Context Switch Aware Large TLB,” in *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*, 2017.

[60] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, “GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC’15)*, 2015.

[61] M. Nam, H. Cha, Y. Choi, S. H. Noh, and B. Nam, “Write-Optimized Dynamic Hashing for Persistent Memory,” in *Proceedings of 17th USENIX Conference on File and Storage Technologies (FAST’19)*, 2019. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/nam

[62] J. Navarro, S. Iyer, P. Druschel, and A. Cox, “Practical, Transparent Operating System Support for Superpages,” in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI’02)*, 2002.

[63] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, D. Chakrabarti, and M. Scott, “Dalí: A Periodically Persistent Hash Map,” in *International Symposium on Distributed Computing*, 2017.

[64] R. Pagh and F. F. Rodler, “Cuckoo Hashing,” *Journal of Algorithms*, vol. 51, no. 2, 2004.

[65] A. Panwar, S. Bansal, and K. Gopinath, “HawkEye: Efficient Fine-grained OS Support for Huge Pages,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’19)*, 2019.

[66] A. Panwar, A. Prasad, and K. Gopinath, “Making Huge Pages Actually Useful,” in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’18)*, 2018.

[67] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, “Prediction-Based Superpage-Friendly TLB Designs,” in *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA’15)*, 2015.

[68] C. H. Park, T. Heo, J. Jeong, and J. Huh, “Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA’17)*, 2017.

[69] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, “Increasing TLB Reach by Exploiting Clustering in Page Translations,” in *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA’14)*, 2014.

[70] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, “CoLT: Coalesced Large-Reach TLBs,” in *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*, 2012.

[71] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, “Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways?” in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, 2015.

[72] A. F. Rodrigues, J. Cook, E. Cooper-Balis, K. S. Hemmert, C. Kersey, R. Riesen, P. Rosenfeld, R. Oldfield, and M. Weston, “The Structural Simulation Toolkit,” in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC’10)*, 2006.

[73] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, “Reducing TLB and Memory Overhead Using Online Superpage Promotion,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA’95)*, 1995.

[74] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “DRAMSim2: A Cycle Accurate Memory System Simulator,” *IEEE Computer Architecture Letters*, 2011.

[75] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, “Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA’17)*, 2017.

[76] A. Saulsbury, F. Dahlgren, and P. Stenström, “Recency-Based TLB Preloading,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA’00)*, 2000.

[77] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, “Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’20)*, 2020.

[78] 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 (MICRO-43)*, 2010.

[79] J. Stojkovic, D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, “Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[80] Y. Sun, Y. Hua, S. Jiang, Q. Li, S. Cao, and P. Zuo, “SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems,” in *Proceedings of the 2019 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC’19)*, 2019.

[81] SysBench, “A modular, cross-platform and multi-threaded benchmark tool.” http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html, 2019.

[82] M. Talluri and M. D. Hill, “Surpassing the TLB Performance of Superpages with Less Operating System Support,” in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, 1994.

[83] M. Talluri, M. D. Hill, and Y. A. Khalidi, “A New Page Table for 64-bit Address Spaces,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP’95)*, 1995.

[84] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, “Tradeoffs in Supporting Two Page Sizes,” in *19th International Symposium on Computer Architecture (ISCA’92)*, 1992.

[85] The Linux Kernel Archives, “Transparent hugepage support,” https://www.kernel.org/doc/Documentation/vm/transhuge.txt, 2019.

[86] J. Triplett, P. E. McKenney, and J. Walpole, “Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming,” in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC’11)*, 2011.

[87] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, “SecDir: A Secure Directory to Defeat Directory Side-Channel Attacks,” in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA’19)*, 2019.

[88] I. Yaniv and D. Tsafrir, “Hash, Don’t Cache (the Page Table),” in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS’16)*, 2016.

[89] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. Solihin, “Hardware-Based Address-Centric Acceleration of Key-Value Store,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[90] G. Zhang and D. Sanchez, “Leveraging Caches to Accelerate Hash Tables and Memoization,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[91] P. Zuo, Y. Hua, and J. Wu, “Write-optimized and high-performance hashing index scheme for persistent memory,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/zuo